



# Intel® Math Kernel Library

---

## *Reference Manual*

Copyright © 1994-2001, Intel Corporation  
All Rights Reserved  
Issued in U.S.A.  
Document Number 630813-011

# *Intel® Math Kernel Library*

## *Reference Manual*

---

Document Number: 630813-011

World Wide Web: <http://developer.intel.com>

<b>Revision</b>	<b>Revision History</b>	<b>Date</b>
-001	Original Issue.	11/94
-002	Added functions crotg, zrotg. Documented versions of functions ?her2k, ?symm, ?syrk, and ?syr2k not previously described. Pagination revised.	5/95
-003	Changed the title; former title: "Intel BLAS Library for the Pentium® Processor Reference Manual." Added functions ?rotm,?rotmg and updated Appendix C.	1/96
-004	Documents Math Kernel library release 2.0 with the parallelism capability. Information on parallelism has been added in Chapter 1 and in section "BLAS Level 3 Routines" in Chapter 2.	11/96
-005	Two-dimensional FFTs have been added. C interface has been added to both one- and two-dimensional FFTs.	8/97
-006	Documents Math Kernel Library release 2.1. Sparse BLAS section has been added in Chapter 2.	1/98
-007	Documents Math Kernel Library release 3.0. Descriptions of LAPACK routines (Chapters 4 and 5) and CBLAS interface (Appendix C) have been added. Quick Reference has been excluded from the manual; MKL 3.0 Quick Reference is now available in HTML format.	1/99
-008	Documents Math Kernel Library release 3.2. Description of FFT routines have been revised. In Chapters 4 and 5 NAG names for LAPACK routines have been excluded.	6/99
-009	New LAPACK routines for eigenvalue problems have been added in chapter 5.	11/99
-010	Documents Math Kernel Library release 4.0. Chapter 6 describing the VML functions has been added.	06/00
-011	Documents Math Kernel Library release 5.1. LAPACK section has been extended to include the full list of computational and driver routines .	04/01

This manual as well as the software described in it is furnished under license and may only be used or copied in accordance with the terms of the license. The information in this manual is furnished for informational use only, is subject to change without notice, and should not be construed as a commitment by Intel Corporation.

Intel Corporation assumes no responsibility or liability for any errors or inaccuracies that may appear in this document or any software that may be provided in association with this document. Except as permitted by such license, no part of this document may be reproduced, stored in a retrieval system, or transmitted in any form or by any means without the express written consent of Intel Corporation.

Information in this document is provided in connection with Intel® products. No license, express or implied, by estoppel or otherwise, to any intellectual property rights is granted by this document.

Except as provided in Intel's Terms and Conditions of Sale for such products, Intel assumes no liability whatsoever, and Intel disclaims any express or implied warranty, relating to sale and/or use of Intel products including liability or warranties relating to fitness for a particular purpose, merchantability, or infringement of any patent, copyright or other intellectual property right.

Intel products are not intended for use in medical, life saving, or life sustaining applications.

Intel may make changes to specifications and product descriptions at any time, without notice.

Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined." Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them. Processors may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available upon request.

Intel, the Intel logo, Pentium, and MMX are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

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

Copyright © 1994-2001, Intel Corporation. All Rights Reserved.

Chapters 4 and 5 include derivative work portions that have been copyrighted:  
© 1991, 1992, and 1998 by The Numerical Algorithms Group, Ltd.

# *Contents*

---

## **Chapter 1 Overview**

About This Software .....	1-1
BLAS Routines.....	1-2
Sparse BLAS Routines .....	1-2
Fast Fourier Transforms .....	1-2
LAPACK Routines .....	1-2
VML Functions .....	1-3
Performance Enhancements .....	1-3
Parallelism.....	1-3
Platforms Supported.....	1-4
About This Manual.....	1-4
Audience for This Manual .....	1-5
Manual Organization.....	1-5
Notational Conventions .....	1-6
Routine Name Shorthand .....	1-6
Font Conventions .....	1-6
Related Publications .....	1-8

## **Chapter 2 BLAS and Sparse BLAS Routines**

Routine Naming Conventions .....	2-2
Matrix Storage Schemes .....	2-3
BLAS Level 1 Routines and Functions .....	2-4
?asum .....	2-5
?axpy .....	2-6

?copy .....	2-7
?dot .....	2-8
?dotc .....	2-9
?dotu .....	2-10
?nrm2 .....	2-11
?rot .....	2-12
?rotg .....	2-14
?rotm .....	2-15
?rotmg .....	2-17
?scal .....	2-18
?swap .....	2-20
i?amax .....	2-21
i?amin .....	2-22
BLAS Level 2 Routines .....	2-23
?gbmv .....	2-24
?gemv .....	2-27
?ger .....	2-30
?gerc .....	2-31
?geru .....	2-33
?hbmv .....	2-35
?hemv .....	2-38
?her .....	2-40
?her2 .....	2-42
?hpmv .....	2-44
?hpr .....	2-47
?hpr2 .....	2-49
?sbmv .....	2-51
?spmv .....	2-54
?spr .....	2-56
?spr2 .....	2-58
?symv .....	2-60
?syr .....	2-62

?syr2 .....	2-64
?tbmv .....	2-66
?tbsv .....	2-69
?tpmv .....	2-72
?tpsv .....	2-75
?trmv .....	2-77
?trsv .....	2-79
BLAS Level 3 Routines .....	2-82
Symmetric Multiprocessing Version of MKL .....	2-82
?gemm .....	2-83
?hemm .....	2-86
?herk .....	2-89
?her2k .....	2-92
?symm .....	2-96
?syrk .....	2-100
?syr2k .....	2-103
?trmm .....	2-107
?trsm .....	2-110
Sparse BLAS Routines and Functions .....	2-114
Vector Arguments in Sparse BLAS .....	2-114
Naming Conventions in Sparse BLAS .....	2-115
Routines and Data Types in Sparse BLAS .....	2-115
BLAS Routines That Can Work With Sparse Vectors .	2-116
?axpyi .....	2-116
?doti .....	2-118
?dotci .....	2-119
?dotui .....	2-120
?gthr .....	2-121
?gthrz .....	2-122
?roti .....	2-123
?sctr .....	2-124

## Chapter 3 Fast Fourier Transforms

One-dimensional FFTs .....	3-1
Data Storage Types .....	3-2
Data Structure Requirements .....	3-2
Complex-to-Complex One-dimensional FFTs .....	3-3
cfft1d/zfft1d .....	3-4
cfft1dc/zfft1dc .....	3-5
Real-to-Complex One-dimensional FFTs .....	3-6
scfft1d/dzfft1d .....	3-8
scfft1dc/dzfft1dc .....	3-10
Complex-to-Real One-dimensional FFTs .....	3-11
csfft1d/zdffft1d .....	3-13
csfft1dc/zdffft1dc .....	3-15
Two-dimensional FFTs .....	3-17
Complex-to-Complex Two-dimensional FFTs .....	3-18
cfft2d/zfft2d .....	3-19
cfft2dc/zfft2dc .....	3-20
Real-to-Complex Two-dimensional FFTs .....	3-21
scfft2d/dzfft2d .....	3-22
scfft2dc/dzfft2dc .....	3-24
Complex-to-Real Two-dimensional FFTs .....	3-27
csfft2d/zdffft2d .....	3-28
csfft2dc/zdffft2dc .....	3-29

## Chapter 4 LAPACK Routines: Linear Equations

Routine Naming Conventions .....	4-2
Matrix Storage Schemes .....	4-3
Mathematical Notation .....	4-3
Error Analysis .....	4-4
Computational Routines .....	4-5
Routines for Matrix Factorization .....	4-7
?getrf .....	4-7

?gbtrf .....	4-10
?gttrf .....	4-12
?potrf .....	4-14
?pptrf .....	4-16
?pbtrf .....	4-18
?pttrf .....	4-20
?sytrf .....	4-22
?hetrf .....	4-25
?sptrf .....	4-28
?hpstrf .....	4-31
Routines for Solving Systems of Linear Equations .....	4-33
?getrs .....	4-34
?gbtrs .....	4-36
?gttrs .....	4-38
?potrs .....	4-41
?pptrs .....	4-43
?pbtrs .....	4-45
?pttrs .....	4-47
?sytrs .....	4-49
?hetrs .....	4-51
?sptrs .....	4-53
?hptrs .....	4-55
?trtrs .....	4-57
?tptrs .....	4-59
?tbtrs .....	4-61
Routines for Estimating the Condition Number .....	4-63
?gecon .....	4-63
?gbcon .....	4-65
?gtcon .....	4-67
?pocon .....	4-70
?ppcon .....	4-72
?pbcon .....	4-74

?ptcon .....	4-76
?sycon .....	4-78
?hecon .....	4-80
?spcon .....	4-82
?hpcon .....	4-84
?trcon .....	4-86
?tpcon .....	4-88
?tbcon .....	4-90
Refining the Solution and Estimating Its Error .....	4-92
?gerfs .....	4-92
?gbrfs .....	4-95
?gtrfs .....	4-98
?porfs .....	4-101
?pprfs .....	4-104
?pbrfs .....	4-107
?ptrfs .....	4-110
?syrfs .....	4-113
?herfs .....	4-116
?sprfs .....	4-119
?hprfs .....	4-122
?trrfs .....	4-124
?tprfs .....	4-127
?tbrfs .....	4-130
Routines for Matrix Inversion .....	4-133
?getri .....	4-133
?potri .....	4-135
?pptri .....	4-137
?sytri .....	4-139
?hetri .....	4-141
?sptri .....	4-143
?hptri .....	4-145
?trtri .....	4-147

?tptri .....	4-148
Routines for Matrix Equilibration .....	4-150
?geequ.....	4-150
?gbequ.....	4-153
?poequ.....	4-155
?ppequ.....	4-157
?pbequ.....	4-159
Driver Routines .....	4-161
?gesv .....	4-162
?gesvx .....	4-163
?gbsv .....	4-170
?gbsvx .....	4-172
?gtsv .....	4-179
?gtsvx .....	4-181
?posv .....	4-186
?posvx .....	4-188
?ppsv .....	4-193
?ppsvx .....	4-195
?pbsv .....	4-200
?pbsvx .....	4-202
?ptsv .....	4-207
?ptsvx .....	4-209
?sysv.....	4-212
?sysvx.....	4-215
?hesvx .....	4-220
?hesv .....	4-224
?spsv .....	4-227
?spsvx.....	4-230
?hpsvx .....	4-235
?hpsv .....	4-239

## Chapter 5 LAPACK Routines: Least Squares and Eigenvalue Problems

Routine Naming Conventions .....	5-4
Matrix Storage Schemes .....	5-5
Mathematical Notation .....	5-5
Computational Routines .....	5-6
Orthogonal Factorizations .....	5-6
?geqrf .....	5-8
?geqpf .....	5-11
?geqp3 .....	5-14
?orgqr .....	5-17
?ormqr .....	5-19
?ungqr .....	5-21
?unmqr .....	5-23
?gelqf .....	5-25
?orglq .....	5-28
?ormlq .....	5-30
?unglq .....	5-32
?unmlq .....	5-34
?geqlf .....	5-36
?orgql .....	5-38
?ungql .....	5-40
?ormql .....	5-42
?unmql .....	5-45
?gerqf .....	5-48
?orgrq .....	5-50
?ungrq .....	5-52
?ormrq .....	5-54
?unmrq .....	5-57
?tzrzf .....	5-60
?ormrz .....	5-62
?unmrz .....	5-65

?ggqrft .....	5-68
?ggrqft .....	5-71
Singular Value Decomposition .....	5-74
?gebrd .....	5-76
?gbbrd .....	5-79
?orgbr .....	5-82
?ormbr .....	5-85
?ungbr .....	5-88
?unmbr .....	5-91
?bdsqr .....	5-94
?bdsdc .....	5-98
Symmetric Eigenvalue Problems .....	5-101
?syrtd .....	5-105
?orgtr .....	5-107
?ormtr .....	5-109
?hetrd .....	5-111
?ungtr .....	5-113
?unmtr .....	5-115
?sptrd .....	5-117
?opgtr .....	5-119
?opmtr .....	5-120
?hpstrd .....	5-122
?upgtr .....	5-124
?upmtr .....	5-125
?sbrtd .....	5-128
?hbtrd .....	5-130
?sterf .....	5-132
?steqr .....	5-134
?stedc .....	5-137
?stegr .....	5-141
?pteqr .....	5-146
?stebz .....	5-149
?stein .....	5-152

?disna .....	5-154
Generalized Symmetric-Definite Eigenvalue Problems.	5-157
?systg .....	5-158
?hegst .....	5-160
?spgst .....	5-162
?hpgst .....	5-164
?sbgst .....	5-166
?hbgst .....	5-169
?pbstf .....	5-172
Nonsymmetric Eigenvalue Problems .....	5-174
?gehrd .....	5-178
?orghr .....	5-180
?ormhr .....	5-182
?unghr .....	5-185
?unmhr .....	5-187
?gebal .....	5-190
?gebak .....	5-193
?hseqr .....	5-195
?hsein .....	5-199
?trevc .....	5-205
?trsna .....	5-209
?trexc .....	5-214
?trsen .....	5-216
?trsyl .....	5-221
Generalized Nonsymmetric Eigenvalue Problems .....	5-224
?gghrd .....	5-225
?ggbal .....	5-228
?ggbak .....	5-231
?hgeqz .....	5-233
?tgevc .....	5-239
?tgexc .....	5-244
?tgsen .....	5-247

?tgsyl .....	5-253
?tgsna .....	5-258
Generalized Singular Value Decomposition .....	5-263
?ggsvp .....	5-264
?tgsja .....	5-268
Driver Routines .....	5-275
Linear Least Squares (LLS) Problems .....	5-275
?gels .....	5-276
?gelsy .....	5-279
?gelss .....	5-283
?gelsd .....	5-286
Generalized LLS Problems.....	5-290
?gglse .....	5-290
?ggglm .....	5-293
Symmetric Eigenproblems.....	5-295
?syev .....	5-296
?heev .....	5-298
?syevd .....	5-300
?heevd .....	5-303
?syevx .....	5-306
?heevx .....	5-310
?syevr .....	5-314
?heevr .....	5-319
?spev .....	5-324
?hpev .....	5-326
?spevd .....	5-328
?hpevd .....	5-331
?spevx .....	5-335
?hpevx .....	5-339
?sbev .....	5-343
?hbev .....	5-345
?sbevd .....	5-347

?hbvd .....	5-350
?sbevx .....	5-354
?hbevx .....	5-358
?stev .....	5-362
?stevd .....	5-364
?stevx .....	5-367
?stevr .....	5-371
Nonsymmetric Eigenproblems .....	5-376
?gees .....	5-376
?geesx .....	5-381
?geev .....	5-387
?geevx .....	5-391
Singular Value Decomposition .....	5-397
?gesvd .....	5-397
?gesdd .....	5-402
?ggsvd .....	5-406
Generalized Symmetric Definite Eigenproblems .....	5-411
?sygv .....	5-412
?hegv .....	5-415
?sygvd .....	5-418
?hegvd .....	5-421
?sygvx .....	5-425
?hegvx .....	5-430
?spgv .....	5-435
?hpgv .....	5-438
?spgvd .....	5-441
?hpgvd .....	5-444
?spgvx .....	5-448
?hpgvx .....	5-452
?sbgv .....	5-456
?hbgv .....	5-459
?sbgvd .....	5-462

?hbgvd .....	5-465
?sbgvx .....	5-469
?hbgvx .....	5-473
Generalized Nonsymmetric Eigenproblems .....	5-478
?gges .....	5-478
?ggesx .....	5-485
?ggev .....	5-493
?ggevx .....	5-498
References.....	5-505

## Chapter 6 Vector Mathematical Functions

Data Types and Accuracy Modes .....	6-2
Function Naming Conventions .....	6-2
Functions Interface .....	6-3
Vector Indexing Methods .....	6-6
Error Diagnostics .....	6-6
VML Mathematical Functions .....	6-8
Inv .....	6-10
Div .....	6-11
Sqrt .....	6-12
InvSqrt .....	6-13
Cbrt .....	6-15
InvCbrt .....	6-16
Pow .....	6-17
Exp .....	6-18
Ln .....	6-20
Log10 .....	6-21
Cos .....	6-22
Sin .....	6-23
SinCos .....	6-24
Tan .....	6-25
Acos .....	6-26
Asin .....	6-27

Atan .....	6-28
Atan2 .....	6-29
Cosh .....	6-30
Sinh .....	6-31
Tanh .....	6-33
Acosh .....	6-34
Asinh .....	6-35
Atanh .....	6-36
VML Pack/Unpack Functions .....	6-37
Pack .....	6-37
Unpack .....	6-39
VML Service Functions .....	6-42
SetMode .....	6-42
GetMode .....	6-45
SetErrStatus .....	6-46
GetErrStatus .....	6-48
ClearErrStatus .....	6-48
SetErrorCallBack .....	6-49
GetErrorCallBack .....	6-51
ClearErrorCallBack .....	6-52

## **Appendix A Routine and Function Arguments**

Vector Arguments in BLAS .....	A-1
Vector Arguments in VML .....	A-3
Matrix Arguments .....	A-4

## **Appendix B Code Examples**

## **Appendix C CBLAS Interface to the BLAS**

CBLAS Arguments .....	C-1
Enumerated Types .....	C-2
Level 1 CBLAS .....	C-3
Level 2 CBLAS .....	C-6
Level 3 CBLAS .....	C-14

## **Glossary**

## **Index**

# Overview

1

The Intel® Math Kernel Library (MKL) provides Fortran routines and functions that perform a wide variety of operations on vectors and matrices.

The library also includes fast Fourier transform functions and vector mathematical functions with Fortran and C interfaces. The MKL enhances the performance of the programs that use it because the library has been optimized for Intel® processors.

This chapter introduces the Math Kernel Library and provides information about the organization of this manual.

## About This Software

The Math Kernel Library includes the following groups of routines:

- Basic Linear Algebra Subprograms (BLAS):
  - vector operations
  - matrix-vector operations
  - matrix-matrix operations
- Sparse BLAS (basic vector operations on sparse vectors)
- Fast Fourier transform routines (with Fortran and C interfaces)
- LAPACK routines for solving systems of linear equations
- LAPACK routines for solving least-squares problems, eigenvalue and singular value problems, and Sylvester's equations
- Vector Mathematical Library (VML) functions for computing core mathematical functions on vector arguments (with Fortran and C interfaces).

For specific issues on using the library, please refer to the *Release Notes*.

## BLAS Routines

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

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

## Sparse BLAS Routines

[Sparse BLAS Routines and Functions](#) operate on sparse vectors (that is, vectors in which most of the elements are zeros). These routines perform vector operations similar to BLAS Level 1 routines. Sparse BLAS routines take advantage of vectors' sparsity: they allow you to store only non-zero elements of vectors.

## Fast Fourier Transforms

[Fast Fourier Transforms](#) (FFTs) are used in digital signal processing and image processing and in partial differential equation (PDE) solvers.

Combined with the BLAS routines, the FFTs contribute to the portability of the programs and provide a simplified interface between your program and the available library.

## LAPACK Routines

The Math Kernel Library coveres the full set of the LAPACK computational and driver routines. These routines can be divided into the following groups according to the operations they perform:

- Routines for solving systems of linear equations, factoring and inverting matrices, and estimating condition numbers (see [Chapter4](#)).
- Routines for solving least-squares problems, eigenvalue and singular value problems, and Sylvester's equations (see [Chapter5](#)).

## VML Functions

VML functions (see [Chapter 6](#)) include a set of highly optimized implementations of certain computationally expensive core mathematical functions (power, trigonometric, exponential, hyperbolic etc.) that operate on real vector arguments.

## Performance Enhancements

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

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

The major optimization techniques used throughout the library include:

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

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

## Parallelism

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

- One way is based on user-managed threads in the program and further distribution of the operations over the threads based on data decomposition, domain decomposition, control decomposition, or some other parallelizing technique. Each thread can use any of the MKL functions because the library has been designed to be thread-safe.

- Another method is to use the FFT and BLAS level 3 routines. They have been parallelized and require no alterations of your application to gain the performance enhancements of multiprocessing. Performance using multiple processors on the level 3 BLAS shows excellent scaling. Since the threads are called and managed within the library, the application does not need to be recompiled thread-safe (see also [BLAS Level 3 Routines](#) in Chapter 2).
- Yet another method is to use *tuned LAPACK routines*. Currently these include the single- and double precision flavors of routines for *QR* factorization of general matrices, triangular factorization of general and symmetric positive-definite matrices, solving systems of equations with such matrices, as well as solving symmetric eigenvalue problems.

For instructions on setting the number of available processors for the BLAS level 3 and LAPACK routines, see the *Release Notes*.

## Platforms Supported

The Math Kernel Library includes Fortran routines and functions optimized for Intel® processor-based computers running operating systems that support multiprocessing. In addition to the Fortran interface, the MKL includes a C-language interface for the fast Fourier transform functions and for the Vector Mathematical Library functions.

## About This Manual

This manual describes the routines of the Math Kernel Library. Each reference section describes a routine group consisting of routines used with four basic data types: single-precision real, double-precision real, single-precision complex, and double-precision complex.

Each routine group is introduced by its name, a short description of its purpose, and the calling sequence for each type of data with which each routine of the group is used. The following sections are also included:

Discussion	Describes the operation performed by routines of the group based on one or more equations. The data types of the arguments are defined in general terms for the group.
------------	--

Input Parameters	Defines the data type for each parameter on entry, for example: <code>a REAL for saxpy</code> <code>DOUBLE PRECISION for daxpy</code>
Output Parameters	Lists resultant parameters on exit.

## Audience for This Manual

The manual addresses programmers proficient in computational linear algebra and assumes a working knowledge of linear algebra and Fourier transform principles and vocabulary.

## Manual Organization

The manual contains the following chapters and appendixes:

- |           |   |
|-----------|---|
| Chapter 1 | <a href="#">Overview</a> . Introduces the Math Kernel Library software, provides information on manual organization, and explains notational conventions.   |
| Chapter 2 | <a href="#">BLAS and Sparse BLAS Routines</a> . Provides descriptions of BLAS and Sparse BLAS functions and routines.   |
| Chapter 3 | <a href="#">Fast Fourier Transforms</a> . Provides descriptions of fast Fourier transforms (FFT).   |
| Chapter 4 | <a href="#">LAPACK Routines: Linear Equations</a> . Provides descriptions of LAPACK routines for solving systems of linear equations and performing a number of related computational tasks: triangular factorization, matrix inversion, estimating the condition number of matrices. |
| Chapter 5 | <a href="#">LAPACK Routines: Least Squares and Eigenvalue Problems</a> . Provides descriptions of LAPACK routines for solving least-squares problems, standard and generalized eigenvalue problems, singular value problems, and Sylvester's equations.                               |

Chapter 6	<a href="#">Vector Mathematical Functions</a> . Provides descriptions of VML functions for computing elementary mathematical functions on vector arguments.
Appendix A	<a href="#">Routine and Function Arguments</a> . Describes the major arguments of the BLAS routines and VML functions: vector and matrix arguments.
Appendix B	<a href="#">Code Examples</a> . Provides code examples of calling BLAS functions and routines.
Appendix C	<a href="#">CBLAS Interface to the BLAS</a> . Provides the C interface to the BLAS.

The manual also includes a [Glossary](#) and an [Index](#).

## Notational Conventions

This manual uses the following notational conventions:

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

### Routine Name Shorthand

For shorthand, character codes are represented by a question mark “?” in names of routine groups. The question mark is used to indicate any or all possible varieties of a function; for example:

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

### Font Conventions

The following font conventions are used:

<code>UPPERCASE COURIER</code>	Data type used in the discussion of input and output parameters for Fortran interface. For example, <code>CHARACTER*1</code> .
<code>lowercase courier</code>	Code examples: <code>a(k+i,j) = matrix(i,j)</code> and data types for C interface, for example, <code>const float*</code>

*lowercase courier mixed  
with UpperCase courier*

Function names for C interface,  
for example, `vmlSetMode`

*lowercase courier italic*

Variables in arguments and parameters  
discussion. For example, `incx`.

\*

Used as a multiplication symbol in code  
examples and equations and where  
required by the Fortran syntax.

## Related Publications

For more information about the BLAS, Sparse BLAS, LAPACK and VML routines, refer to the following publications:

- BLAS Level 1  
C. Lawson, R. Hanson, D. Kincaid, and F. Krough. *Basic Linear Algebra Subprograms for Fortran Usage*, ACM Transactions on Mathematical Software, Vol.5, No.3 (September 1979) 308-325.
- BLAS Level 2  
J. Dongarra, J. Du Croz, S. Hammarling, and R. Hanson. *An Extended Set of Fortran Basic Linear Algebra Subprograms*, ACM Transactions on Mathematical Software, Vol.14, No.1 (March 1988) 1-32.
- BLAS Level 3  
J. Dongarra, J. DuCroz, I. Duff, and S. Hammarling. *A Set of Level 3 Basic Linear Algebra Subprograms*, ACM Transactions on Mathematical Software (December 1989).
- Sparse BLAS  
D. Dodson, R. Grimes, and J. Lewis. *Sparse Extensions to the FORTRAN Basic Linear Algebra Subprograms*, ACM Transactions on Mathematical Software, Vol.17, No.2 (June 1991).  
D. Dodson, R. Grimes, and J. Lewis. *Algorithm 692: Model Implementation and Test Package for the Sparse Basic Linear Algebra Subprograms*, ACM Transactions on Mathematical Software, Vol.17, No.2 (June 1991).
- LAPACK  
E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Donagarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*, Third Edition, Society for Industrial and Applied Mathematics (SIAM), 1999.  
G. Golub and C. Van Loan. *Matrix Computations*, Johns Hopkins University Press, 1989.
- VML  
J.M.Muller. *Elementary functions: algorithms and implementation*, Birkhauser Boston, 1997.  
IEEE Standard for Binary Floating-Point Arithmetic. ANSI/IEEE Std 754-1985.

For a reference implementation of BLAS, sparse BLAS, and LAPACK packages (without platform-specific optimizations) visit [www.netlib.org](http://www.netlib.org).

# *BLAS and Sparse BLAS Routines*

2

This chapter contains descriptions of the BLAS and Sparse BLAS routines of the Intel® Math Kernel Library. The routine descriptions are arranged in four sections according to the BLAS level of operation:

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

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

When BLAS routines encounter an error, they call the error reporting routine `XERBLA`. To be able to view error reports, you must include `XERBLA` in your code. A copy of the source code for `XERBLA` is included in the library.

In BLAS Level 1 groups `i?amax` and `i?amin`, an “i” is placed before the character code and corresponds to the index of an element in the vector. These groups are placed in the end of the BLAS Level 1 section.

## Routine Naming Conventions

BLAS routine names have the following structure:

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

The `<character code>` is a character that indicates the data type:

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

Some routines and functions can have combined character codes, such as `sc` or `dz`. For example, the function `scasum` uses a complex input array and returns a real value.

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

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

<code>ge</code>	general matrix
<code>gb</code>	general band matrix
<code>sy</code>	symmetric matrix
<code>sp</code>	symmetric matrix (packed storage)
<code>sb</code>	symmetric band matrix
<code>he</code>	Hermitian matrix
<code>hp</code>	Hermitian matrix (packed storage)
<code>hb</code>	Hermitian band matrix
<code>tr</code>	triangular matrix
<code>tp</code>	triangular matrix (packed storage)
<code>tb</code>	triangular band matrix.

The `<mod>` field, if present, provides additional details of the operation.

BLAS level 1 names can have the following characters in the `<mod>` field:

<code>c</code>	conjugated vector
<code>u</code>	unconjugated vector
<code>g</code>	Givens rotation.

BLAS level 2 names can have the following characters in the `<mod>` field:

<code>mv</code>	matrix-vector product
<code>sv</code>	solving a system of linear equations with matrix-vector operations
<code>r</code>	rank-1 update of a matrix
<code>r2</code>	rank-2 update of a matrix.

BLAS level 3 names can have the following characters in the `<mod>` field:

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

The examples below illustrate how to interpret BLAS routine names:

- `<d> <dot>` `ddot`: double-precision real vector-vector dot product
- `<c> <dot> <c>` `cdotc`: complex vector-vector dot product, conjugated
- `<sc> <asum>` `scasum`: sum of magnitudes of vector elements, single precision real output and single precision complex input
- `<c> <dot> <u>` `cdotu`: vector-vector dot product, unconjugated, complex
- `<s> <ge> <mv>` `sgemv`: matrix-vector product, general matrix, single precision
- `<z> <tr> <mm>` `ztrmm`: matrix-matrix product, triangular matrix, double-precision complex.

Sparse BLAS naming conventions are similar to those of BLAS level 1.

For more information, see *Naming conventions in Sparse BLAS*.

## Matrix Storage Schemes

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

- *Full storage*: a matrix  $A$  is stored in a two-dimensional array `a`, with the matrix element  $a_{ij}$  stored in the array element `a(i, j)`.
- *Packed storage* scheme allows you to store symmetric, Hermitian, or triangular matrices more compactly: the upper or lower triangle of the matrix is packed by columns in a one-dimensional array.
- *Band storage*: a band matrix is stored compactly in a two-dimensional array: columns of the matrix are stored in the corresponding columns of the array, and *diagonals* of the matrix are stored in rows of the array.

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

## BLAS Level 1 Routines and Functions

BLAS Level 1 includes routines and functions, which perform vector-vector operations. Table 2-1 lists the BLAS Level 1 routine and function groups and the data types associated with them.

**Table 2-1 BLAS Level 1 Routine Groups and Their Data Types**

Routine or Function Group	Data Types	Description
<a href="#"><u>?asum</u></a>	s, d, sc, dz	Sum of vector magnitudes (functions)
<a href="#"><u>?axpy</u></a>	s, d, c, z	Scalar-vector product (routines)
<a href="#"><u>?copy</u></a>	s, d, c, z	Copy vector (routines)
<a href="#"><u>?dot</u></a>	s, d	Dot product (functions)
<a href="#"><u>?dotc</u></a>	c, z	Dot product conjugated (functions)
<a href="#"><u>?dotu</u></a>	c, z	Dot product unconjugated (functions)
<a href="#"><u>?nrm2</u></a>	s, d, sc, dz	Vector 2-norm (Euclidean norm) a normal or null vector (functions)
<a href="#"><u>?rot</u></a>	s, d, cs, zd	Plane rotation of points (routines)
<a href="#"><u>?rotg</u></a>	s, d, c, z	Givens rotation of points (routines)
<a href="#"><u>?rotm</u></a>	s, d	Modified plane rotation of points
<a href="#"><u>?rotmg</u></a>	s, d	Givens modified plane rotation of points
<a href="#"><u>?scal</u></a>	s, d, c, z, cs, zd	Vector scaling (routines)
<a href="#"><u>?swap</u></a>	s, d, c, z	Vector-vector swap (routines)
<a href="#"><u>i?amax</u></a>	s, d, c, z	Vector maximum value, absolute largest element of a vector where <i>i</i> is an index to this value in the vector array (functions)
<a href="#"><u>i?amin</u></a>	s, d, c, z	Vector minimum value, absolute smallest element of a vector where <i>i</i> is an index to this value in the vector array (functions)

## ?asum

*Computes the sum of magnitudes of the vector elements.*

---

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

### Discussion

Given a vector *x*, ?asum functions compute the sum of the magnitudes of its elements or, for complex vectors, the sum of magnitudes of the elements' real parts plus magnitudes of their imaginary parts:

$$res = |\text{Re}x(1)| + |\text{Im}x(1)| + |\text{Re}x(2)| + |\text{Im}x(2)| + \dots + |\text{Re}x(n)| + |\text{Im}x(n)|$$

where *x* is a vector of order *n*.

### Input Parameters

<i>n</i>	INTEGER. Specifies the order of vector <i>x</i> .
<i>x</i>	REAL for sasum DOUBLE PRECISION for dasum COMPLEX for scasum DOUBLE COMPLEX for dzasum
	Array, DIMENSION at least $(1 + (n-1)*\text{abs}(incx))$ .
<i>incx</i>	INTEGER. Specifies the increment for the elements of <i>x</i> .

### Output Parameters

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

---

## ?axpy

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

---

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

### Discussion

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

$$y := a*x + y$$

where:

*a* is a scalar

*x* and *y* are vectors of order *n*.

### Input Parameters

*n* INTEGER. Specifies the order of vectors *x* and *y*.

*a* REAL for *saxpy*  
DOUBLE PRECISION for *daxpy*  
COMPLEX for *caxpy*  
DOUBLE COMPLEX for *zaxpy*

Specifies the scalar *a*.

*x* REAL for *saxpy*  
DOUBLE PRECISION for *daxpy*  
COMPLEX for *caxpy*  
DOUBLE COMPLEX for *zaxpy*

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

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

<i>y</i>	REAL for <code>saxy</code> DOUBLE PRECISION for <code>daxy</code> COMPLEX for <code>caxy</code> DOUBLE COMPLEX for <code>zaxy</code>
	Array, DIMENSION at least $(1 + (n-1) * \text{abs}(incy))$ .
<i>incy</i>	INTEGER. Specifies the increment for the elements of <i>y</i> .

### Output Parameters

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

---

## ?copy

Copies vector to another vector.

---

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

### Discussion

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

*y* = *x*

where *x* and *y* are vectors.

### Input Parameters

<i>n</i>	INTEGER. Specifies the order of vectors <i>x</i> and <i>y</i>
<i>x</i>	REAL for <code>scopy</code> DOUBLE PRECISION for <code>dcopy</code> COMPLEX for <code>ccopy</code> DOUBLE COMPLEX for <code>zcopy</code>
	Array, DIMENSION at least $(1 + (n-1) * \text{abs}(incx))$ .
<i>incx</i>	INTEGER. Specifies the increment for the elements of <i>x</i> .

<i>y</i>	REAL for <b>scopy</b> DOUBLE PRECISION for <b>dcopy</b> COMPLEX for <b>ccopy</b> DOUBLE COMPLEX for <b>zcopy</b>
<i>incy</i>	Array, DIMENSION at least $(1 + (n-1)*\text{abs}(incy))$ .
	INTEGER. Specifies the increment for the elements of <i>x</i> .

### Output Parameters

<i>y</i>	Contains a copy of the vector <i>x</i> if <i>n</i> is positive. Otherwise, parameters are unaltered.
----------	---

---

## ?dot

*Computes a vector-vector dot product.*

---

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

### Discussion

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

$$res = \sum(x^*y)$$

where *x* and *y* are vectors.

### Input Parameters

<i>n</i>	INTEGER. Specifies the order of vectors <i>x</i> and <i>y</i> .
<i>x</i>	REAL for <b>sdot</b> DOUBLE PRECISION for <b>ddot</b>
<i>incx</i>	Array, DIMENSION at least $(1+(n-1)*\text{abs}(incx))$ .
	INTEGER. Specifies the increment for the elements of <i>x</i> .

*y*                   REAL for `sdot`  
                       DOUBLE PRECISION for `ddot`  
                       Array, DIMENSION at least  $(1 + (n-1) * \text{abs}(incy))$ .  
*incy*               INTEGER. Specifies the increment for the elements of *y*.

### Output Parameters

*res*               REAL for `sdot`  
                       DOUBLE PRECISION for `ddot`  
                       Contains the result of the dot product of *x* and *y*, if *n* is positive. Otherwise, *res* contains 0.

---

## ?dotc

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

---

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

### Discussion

The ?dotc functions perform a vector-vector operation defined as

$$res = \sum (\text{conjg}(x)^* y)$$

where *x* and *y* are *n*-element vectors.

### Input Parameters

*n*                   INTEGER. Specifies the order of vectors *x* and *y*.  
*x*                   COMPLEX for `cdotc`  
                       DOUBLE COMPLEX for `zdotc`  
                       Array, DIMENSION at least  $(1 + (n-1) * \text{abs}(incx))$ .  
*incx*               INTEGER. Specifies the increment for the elements of *x*.

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

### Output Parameters

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

---

## ?dotu

Computes a vector-vector dot product.

---

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

### Discussion

The `?dotu` functions perform a vector-vector reduction operation defined as  $\text{res} = \sum k * y$  where *x* and *y* are *n*-element vectors.

### Input Parameters

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

<i>y</i>	COMPLEX for <code>cdotu</code> DOUBLE COMPLEX for <code>zdotu</code>
Array, DIMENSION at least $(1 + (n-1) * \text{abs}(incy))$ .	
<i>incy</i>	INTEGER. Specifies the increment for the elements of <i>y</i> .

### Output Parameters

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

---

## ?nrm2

Computes the Euclidean norm of a vector.

---

```

res = snrm2 ( n, x, incx )
res = dnrm2 ( n, x, incx )
res = scnrm2 ( n, x, incx )
res = dznrm2 ( n, x, incx )

```

### Discussion

The ?nrm2 functions perform a vector reduction operation defined as

*res* =  $\| \mathbf{x} \|$

where:

*x* is a vector

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

**Input Parameters**

<i>n</i>	INTEGER. Specifies the order of vector <i>x</i> .
<i>x</i>	REAL for <i>snrm2</i> DOUBLE PRECISION for <i>dnrm2</i> COMPLEX for <i>scnrm2</i> DOUBLE COMPLEX for <i>dznrm2</i>
	Array, DIMENSION at least $(1 + (n-1) * \text{abs}(\text{incx}))$ .
<i>incx</i>	INTEGER. Specifies the increment for the elements of <i>x</i> .

**Output Parameters**

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

---

**?rot***Performs rotation of points in the plane.*

---

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

**Discussion**

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

$$\begin{aligned}x(i) &= c*x(i) + s*y(i) \\y(i) &= c*y(i) - s*x(i)\end{aligned}$$

### Input Parameters

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

### Output Parameters

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

---

## ?rotg

*Computes the parameters for a Givens rotation.*

---

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

### Discussion

Given the cartesian coordinates (*a*, *b*) of a point *p*, these routines return the parameters *a*, *b*, *c*, and *s* associated with the Givens rotation that zeros the *y*-coordinate of the point.

### Input Parameters

<i>a</i>	REAL for <b>srotg</b> DOUBLE PRECISION for <b>drotg</b> COMPLEX for <b>crotg</b> DOUBLE COMPLEX for <b>zrotg</b>
	Provides the <i>x</i> -coordinate of the point <i>p</i> .
<i>b</i>	REAL for <b>srotg</b> DOUBLE PRECISION for <b>drotg</b> COMPLEX for <b>crotg</b> DOUBLE COMPLEX for <b>zrotg</b>
	Provides the <i>y</i> -coordinate of the point <i>p</i> .

### Output Parameters

<i>a</i>	Contains the parameter <i>r</i> associated with the Givens rotation.
<i>b</i>	Contains the parameter <i>z</i> associated with the Givens rotation.

---

<i>c</i>	REAL for <code>srotg</code> DOUBLE PRECISION for <code>drotg</code> REAL for <code>crotg</code> DOUBLE PRECISION for <code>zrotg</code>
	Contains the parameter <i>c</i> associated with the Givens rotation.
<i>s</i>	REAL for <code>srotg</code> DOUBLE PRECISION for <code>drotg</code> COMPLEX for <code>crotg</code> DOUBLE COMPLEX for <code>zrotg</code>
	Contains the parameter <i>s</i> associated with the Givens rotation.

---

## ?rotm

Performs rotation of points in the modified plane.

---

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

### Discussion

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

$$\begin{aligned}x(i) &= H*x(i) + H*y(i) \\y(i) &= H*y(i) - H*x(i)\end{aligned}$$

where:

*H* is a modified Givens transformation matrix whose values are stored in the *param(2)* through *param(5)* array. See discussion on the *param* argument.

## Input Parameters

<i>n</i>	INTEGER. Specifies the order of vectors <i>x</i> and <i>y</i> .
<i>x</i>	REAL for <b>srotm</b> DOUBLE PRECISION for <b>drotm</b> Array, DIMENSION at least $(1 + (n-1) * \text{abs}(incx))$ .
<i>incx</i>	INTEGER. Specifies the increment for the elements of <i>x</i> .
<i>y</i>	REAL for <b>srotm</b> DOUBLE PRECISION for <b>drotm</b> Array, DIMENSION at least $(1 + (n-1) * \text{abs}(incy))$ .
<i>incy</i>	INTEGER. Specifies the increment for the elements of <i>y</i> .
<i>param</i>	REAL for <b>srotm</b> DOUBLE PRECISION for <b>drotm</b> Array, DIMENSION 5.

The elements of the *param* array are:

*param(1)* contains a switch, *flag*.

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

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

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

$$flag = 0.: H = \begin{bmatrix} 1. & h12 \\ h21 & 1. \end{bmatrix}$$

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

$$flag = -2.: H = \begin{bmatrix} 1. & 0. \\ 0. & 1. \end{bmatrix}$$

In the above cases, the matrix entries of 1., -1., and 0. are assumed based on the last three values of *flag* and are not actually loaded into the *param* vector.

### Output Parameters

- $x$       Each element is replaced by  $h11*x + h12*y$ .
- $y$       Each element is replaced by  $h21*x + h22*y$ .
- $H$       Givens transformation matrix updated.

## ?rotmg

Computes the modified parameters for a Givens rotation.

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

### Discussion

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

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

### Input Parameters

- $d1$       REAL for `srotmg`  
DOUBLE PRECISION for `drotmg`  
Provides the scaling factor for the  $x$ -coordinate of the input vector ( $\sqrt(d1)x1$ ).
- $d2$       REAL for `srotmg`  
DOUBLE PRECISION for `drotmg`  
Provides the scaling factor for the  $y$ -coordinate of the input vector ( $\sqrt(d2)y1$ ).
- $x1$       REAL for `srotmg`  
DOUBLE PRECISION for `drotmg`  
Provides the  $x$ -coordinate of the input vector.

*y1*                    REAL for `srotmg`  
DOUBLE PRECISION for `drotmg`  
 Provides the *y*-coordinate of the input vector.

### Output Parameters

*param*                    REAL for `srotmg`  
DOUBLE PRECISION for `drotmg`  
 Array, DIMENSION 5.

The elements of the *param* array are:

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

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

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

$$\text{flag} = 0.: H = \begin{bmatrix} 1. & h_{12} \\ h_{21} & 1. \end{bmatrix}$$

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

$$\text{flag} = -2.: H = \begin{bmatrix} 1. & 0. \\ 0. & 1. \end{bmatrix}$$

In the above cases, the matrix entries of 1., -1., and 0. are assumed based on the last three values of *flag* and are not actually loaded into the *param* vector.

---

## ?scal

*Computes a vector by a scalar product.*

---

`call sscal ( n, a, x, incx )`

---

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

### Discussion

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

$x = a \cdot x$

where:

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

### Input Parameters

$n$             INTEGER. Specifies the order of vector  $x$ .  
 $a$             REAL for `sscal` and `csscal`  
                DOUBLE PRECISION for `dscal` and `zdscal`  
                REAL for `cscal`  
                DOUBLE PRECISION for `zscal`  
                Specifies the scalar  $a$ .  
 $x$             REAL for `sscal`  
                DOUBLE PRECISION for `dscal`  
                COMPLEX for `cscal` and `csscal`  
                DOUBLE COMPLEX for `zscal` and `csscal`  
                Array, DIMENSION at least  $(1 + (n-1) * \text{abs}(incx))$ .  
 $incx$         INTEGER. Specifies the increment for the elements of  $x$ .

### Output Parameters

$x$             Overwritten by the updated vector  $x$ .

---

## ?swap

*Swaps a vector with another vector.*

---

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

### Discussion

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

### Input Parameters

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

### Output Parameters

<i>x</i>	Contains the resultant vector <i>x</i> .
<i>y</i>	Contains the resultant vector <i>y</i> .

---

## i?amax

*Finds the element of a vector that has the largest absolute value.*

---

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

### Discussion

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

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

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

### Input Parameters

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

### Output Parameters

<i>index</i>	INTEGER. Contains the position of vector element <i>x</i> that has the largest absolute value.
--------------	--

## i?amin

*Finds the element of a vector that has the smallest absolute value.*

---

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

### Discussion

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

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

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

### Input Parameters

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

### Output Parameters

$index$	<code>INTEGER</code> . Contains the position of vector element $x$ that has the smallest absolute value.
---------	--

## BLAS Level 2 Routines

This section describes BLAS Level 2 routines, which perform matrix-vector operations. Table 2-2 lists the BLAS Level 2 routine groups and the data types associated with them.

**Table 2-2 BLAS Level 2 Routine Groups and Their Data Types**

Routine Groups	Data Types	Description
?gbmv	s, d, c, z	Matrix-vector product using a general band matrix
?gemv	s, d, c, z	Matrix-vector product using a general matrix
?ger	s, d	Rank-1 update of a general matrix
?gerc	c, z	Rank-1 update of a conjugated general matrix
?geru	c, z	Rank-1 update of a general matrix, unconjugated
?hbmv	c, z	Matrix-vector product using a Hermitian band matrix
?hemv	c, z	Matrix-vector product using a Hermitian matrix
?her	c, z	Rank-1 update of a Hermitian matrix
?her2	c, z	Rank-2 update of a Hermitian matrix
?hpmv	c, z	Matrix-vector product using a Hermitian packed matrix
?hpr	c, z	Rank-1 update of a Hermitian packed matrix
?hpr2	c, z	Rank-2 update of a Hermitian packed matrix
?sbmv	s, d	Matrix-vector product using symmetric band matrix
?spmv	s, d	Matrix-vector product using a symmetric packed matrix
?spr	s, d	Rank-1 update of a symmetric packed matrix
?spr2	s, d	Rank-2 update of a symmetric packed matrix
?symv	s, d	Matrix-vector product using a symmetric matrix
?syr	s, d	Rank-1 update of a symmetric matrix
?syr2	s, d	Rank-2 update of a symmetric matrix

continued \*

**Table 2-2 BLAS Level 2 Routine Groups and Their Data Types (continued)**

Routine Groups	Data Types	Description
<a href="#">?tbmv</a>	s, d, c, z	Matrix-vector product using a triangular band matrix
<a href="#">?tbsv</a>	s, d, c, z	Linear solution of a triangular band matrix
<a href="#">?tpmv</a>	s, d, c, z	Matrix-vector product using a triangular packed matrix
<a href="#">?tpsv</a>	s, d, c, z	Linear solution of a triangular packed matrix
<a href="#">?trmv</a>	s, d, c, z	Matrix-vector product using a triangular matrix
<a href="#">?trsv</a>	s, d, c, z	Linear solution of a triangular matrix

**?gbmv**

*Computes a matrix-vector product using a general band matrix*

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

**Discussion**

The [?gbmv](#) routines perform a matrix-vector operation defined as

$y := \alpha * a * x + \beta * y$

or

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

or

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

where:

*alpha* and *beta* are scalars

*x* and *y* are vectors

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

### Input Parameters

*trans*                    CHARACTER\*1. Specifies the operation to be performed, as follows:

<i>trans</i> value	Operation to be Performed
N or n	<i>y</i> := <i>alpha</i> * <i>a</i> * <i>x</i> + <i>beta</i> * <i>y</i>
T or t	<i>y</i> := <i>alpha</i> * <i>a</i> '* <i>x</i> + <i>beta</i> * <i>y</i>
C or c	<i>y</i> := <i>alpha</i> *conjg( <i>a</i> ')* <i>x</i> + <i>beta</i> * <i>y</i>

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

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

*kl*                    INTEGER. Specifies the number of sub-diagonals of the matrix *a*. The value of *kl* must satisfy  $0 \leq kl$ .

*ku*                    INTEGER. Specifies the number of super-diagonals of the matrix *a*. The value of *ku* must satisfy  $0 \leq ku$ .

*alpha*                REAL for sgbmv  
DOUBLE PRECISION for dgbmv  
COMPLEX for cgbmv  
DOUBLE COMPLEX for zgbmv

Specifies the scalar *alpha*.

*a*                    REAL for sgbmv  
DOUBLE PRECISION for dgbmv  
COMPLEX for cgbmv  
DOUBLE COMPLEX for zgbmv

Array, **DIMENSION** (*lda*, *n*). Before entry, the leading (*k1 + ku + 1*) by *n* part of the array *a* must contain the matrix of coefficients. This matrix must be supplied column-by-column, with the leading diagonal of the matrix in row (*ku + 1*) of the array, the first super-diagonal starting at position 2 in row *ku*, the first sub-diagonal starting at position 1 in row (*ku + 2*), and so on. Elements in the array *a* that do not correspond to elements in the band matrix (such as the top left *ku* by *ku* triangle) are not referenced.

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

```
do 20, j = 1, n
  k = ku + 1 - j
  do 10, i = max(1, j-ku), min(m, j+k1)
    a(k+i, j) = matrix(i,j)
  10 continue
20 continue
```

*lda* **INTEGER**. Specifies the first dimension of *a* as declared in the calling (sub)program. The value of *lda* must be at least (*k1 + ku + 1*).

*x* **REAL** for **sgbmv**  
**DOUBLE PRECISION** for **dgbmv**  
**COMPLEX** for **cgbmv**  
**DOUBLE COMPLEX** for **zgbmv**

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

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

*beta* **REAL** for **sgbmv**  
**DOUBLE PRECISION** for **dgbmv**  
**COMPLEX** for **cgbmv**  
**DOUBLE COMPLEX** for **zgbmv**

---

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

*y*

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

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

*incy*

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

### Output Parameters

*y*

Overwritten by the updated vector *y*.

---

## ?gemv

Computes a matrix-vector product using  
a general matrix

---

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

### Discussion

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

*y* := *alpha*\**a*\**x* + *beta*\**y*,

or

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

or

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

where:

$\alpha$  and  $\beta$  are scalars

$x$  and  $y$  are vectors

$a$  is an  $m$  by  $n$  matrix.

## Input Parameters

$trans$       CHARACTER\*1. Specifies the operation to be performed, as follows:

$trans$ value	Operation to be Performed
N or n	$y := \alpha * a * x + \beta * y$
T or t	$y := \alpha * a' * x + \beta * y$
C or c	$y := \alpha * \text{conj}(a') * x + \beta * y$

$m$       INTEGER. Specifies the number of rows of the matrix  $a$ .  $m$  must be at least zero.

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

$\alpha$       REAL for sgemv  
DOUBLE PRECISION for dgemv  
COMPLEX for cgemv  
DOUBLE COMPLEX for zgemv

Specifies the scalar  $\alpha$ .

$a$       REAL for sgemv  
DOUBLE PRECISION for dgemv  
COMPLEX for cgemv  
DOUBLE COMPLEX for zgemv

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

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

*x* **REAL** for **sgemv**  
**DOUBLE PRECISION** for **dgemv**  
**COMPLEX** for **cgemv**  
**DOUBLE COMPLEX** for **zgemv**

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

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

*beta* **REAL** for **sgemv**  
**DOUBLE PRECISION** for **dgemv**  
**COMPLEX** for **cgemv**  
**DOUBLE COMPLEX** for **zgemv**

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

*y* **REAL** for **sgemv**  
**DOUBLE PRECISION** for **dgemv**  
**COMPLEX** for **cgemv**  
**DOUBLE COMPLEX** for **zgemv**

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

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

## Output Parameters

**y** Overwritten by the updated vector **y**.

---

## ?ger

*Performs a rank-1 update of a general matrix.*

---

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

### Discussion

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

$$\alpha := \alpha * \mathbf{x} * \mathbf{y}^T + \mathbf{a},$$

where:

**alpha** is a scalar

**x** is an **m**-element vector

**y** is an **n**-element vector

**a** is an **m** by **n** matrix.

## Input Parameters

<b>m</b>	<b>INTEGER</b> . Specifies the number of rows of the matrix <b>a</b> . The value of <b>m</b> must be at least zero.
<b>n</b>	<b>INTEGER</b> . Specifies the number of columns of the matrix <b>a</b> . The value of <b>n</b> must be at least zero.
<b>alpha</b>	<b>REAL</b> for <b>sger</b> <b>DOUBLE PRECISION</b> for <b>dger</b>  Specifies the scalar <b>alpha</b> .
<b>x</b>	<b>REAL</b> for <b>sger</b> <b>DOUBLE PRECISION</b> for <b>dger</b>

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

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

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

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

**a** **REAL** for **sger**  
**DOUBLE PRECISION** for **dger**  
Array, **DIMENSION** (**lda**, **n**). Before entry, the leading **m** by **n** part of the array **a** must contain the matrix of coefficients.

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

### Output Parameters

**a** Overwritten by the updated matrix.

## ?gerc

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

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

## Discussion

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

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

where:

`alpha` is a scalar

`x` is an  $m$ -element vector

`y` is an  $n$ -element vector

`a` is an  $m$  by  $n$  matrix.

## Input Parameters

<code>m</code>	<code>INTEGER</code> . Specifies the number of rows of the matrix <code>a</code> . The value of <code>m</code> must be at least zero.
<code>n</code>	<code>INTEGER</code> . Specifies the number of columns of the matrix <code>a</code> . The value of <code>n</code> must be at least zero.
<code>alpha</code>	<code>SINGLE PRECISION COMPLEX</code> for <code>cgerc</code> <code>DOUBLE PRECISION COMPLEX</code> for <code>zgerc</code>  Specifies the scalar <code>alpha</code> .
<code>x</code>	<code>SINGLE PRECISION COMPLEX</code> for <code>cgerc</code> <code>DOUBLE PRECISION COMPLEX</code> for <code>zgerc</code>  Array, <code>DIMENSION</code> at least $(1 + (m - 1) * \text{abs}(incx))$ . Before entry, the incremented array <code>x</code> must contain the $m$ -element vector <code>x</code> .
<code>incx</code>	<code>INTEGER</code> . Specifies the increment for the elements of <code>x</code> . The value of <code>incx</code> must not be zero.
<code>y</code>	<code>COMPLEX</code> for <code>cgerc</code> <code>DOUBLE COMPLEX</code> for <code>zgerc</code>  Array, <code>DIMENSION</code> at least $(1 + (n - 1) * \text{abs}(incy))$ . Before entry, the incremented array <code>y</code> must contain the $n$ -element vector <code>y</code> .
<code>incy</code>	<code>INTEGER</code> . Specifies the increment for the elements of <code>y</code> . The value of <code>incy</code> must not be zero.

---

$a$	COMPLEX for <code>cgerc</code> DOUBLE COMPLEX for <code>zgerc</code>
	Array, DIMENSION ( $lda, n$ ). Before entry, the leading $m$ by $n$ part of the array $a$ must contain the matrix of coefficients.
$lda$	INTEGER. Specifies the first dimension of $a$ as declared in the calling (sub)program. The value of $lda$ must be at least $\max(1, m)$ .

### Output Parameters

$a$	Overwritten by the updated matrix.
-----	------------------------------------

---

## ?geru

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

---

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

### Discussion

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

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

where:

$\alpha$  is a scalar

$x$  is an  $m$ -element vector

$y$  is an  $n$ -element vector

$a$  is an  $m$  by  $n$  matrix.

## Input Parameters

<i>m</i>	<b>INTEGER.</b> Specifies the number of rows of the matrix <i>a</i> . The value of <i>m</i> must be at least zero.
<i>n</i>	<b>INTEGER.</b> Specifies the number of columns of the matrix <i>a</i> . The value of <i>n</i> must be at least zero.
<i>alpha</i>	<b>COMPLEX</b> for <i>cgeru</i> <b>DOUBLE COMPLEX</b> for <i>zgeru</i>  Specifies the scalar <i>alpha</i> .
<i>x</i>	<b>COMPLEX</b> for <i>cgeru</i> <b>DOUBLE COMPLEX</b> for <i>zgeru</i>  Array, <b>DIMENSION</b> at least $(1 + (m - 1) * \text{abs}(incx))$ . Before entry, the incremented array <i>x</i> must contain the <i>m</i> -element vector <i>x</i> .
<i>incx</i>	<b>INTEGER.</b> Specifies the increment for the elements of <i>x</i> . The value of <i>incx</i> must not be zero.
<i>y</i>	<b>COMPLEX</b> for <i>cgeru</i> <b>DOUBLE COMPLEX</b> for <i>zgeru</i>  Array, <b>DIMENSION</b> at least $(1 + (n - 1) * \text{abs}(incy))$ . Before entry, the incremented array <i>y</i> must contain the <i>n</i> -element vector <i>y</i> .
<i>incy</i>	<b>INTEGER.</b> Specifies the increment for the elements of <i>y</i> . The value of <i>incy</i> must not be zero.
<i>a</i>	<b>COMPLEX</b> for <i>cgeru</i> <b>DOUBLE COMPLEX</b> for <i>zgeru</i>  Array, <b>DIMENSION</b> ( <i>lda</i> , <i>n</i> ). Before entry, the leading <i>m</i> by <i>n</i> part of the array <i>a</i> must contain the matrix of coefficients.
<i>lda</i>	<b>INTEGER.</b> Specifies the first dimension of <i>a</i> as declared in the calling (sub)program. The value of <i>lda</i> must be at least $\max(1, m)$ .

## Output Parameters

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

---

## ?hbmv

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

---

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

### Discussion

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

$$y := \alpha \cdot a \cdot x + \beta \cdot y,$$

where:

$\alpha$  and  $\beta$  are scalars

$x$  and  $y$  are  $n$ -element vectors

$a$  is an  $n$  by  $n$  Hermitian band matrix, with  $k$  super-diagonals.

### Input Parameters

*uplo*      CHARACTER\*1. Specifies whether the upper or lower triangular part of the band matrix *a* is being supplied, as follows:

<i>uplo</i> value	Part of Matrix <i>a</i> Supplied
U or u	The upper triangular part of matrix <i>a</i> is being supplied.
L or l	The lower triangular part of matrix <i>a</i> is being supplied.

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

*k*      INTEGER. Specifies the number of super-diagonals of the matrix *a*. The value of *k* must satisfy  $0 \leq k$ .

*alpha*COMPLEX for `chbmv`DOUBLE COMPLEX for `zhbmv`Specifies the scalar *alpha*.*a*COMPLEX for `chbmv`DOUBLE COMPLEX for `zhbmv`

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

The following program segment transfers the upper triangular part of a Hermitian band matrix from conventional full matrix storage to band storage:

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

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

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

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

```
a( m + i, j ) = matrix( i, j )
10 continue
20 continue
```

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

*lda*

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

*x*

**COMPLEX** for *chbmv*
**DOUBLE COMPLEX** for *zhbmv*

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

*incx*

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

*beta*

**COMPLEX** for *chbmv*
**DOUBLE COMPLEX** for *zhbmv*

Specifies the scalar *beta*.

*y*

**COMPLEX** for *chbmv*
**DOUBLE COMPLEX** for *zhbmv*

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

*incy*

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

## Output Parameters

*y*

Overwritten by the updated vector *y*.

## ?hemv

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

---

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

### Discussion

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

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

where:

$\alpha$  and  $\beta$  are scalars

$x$  and  $y$  are  $n$ -element vectors

$a$  is an  $n$  by  $n$  Hermitian matrix.

### Input Parameters

*uplo*      CHARACTER\*1. Specifies whether the upper or lower triangular part of the array *a* is to be referenced, as follows:

<i>uplo</i> value	Part of Array <i>a</i> To Be Referenced
<i>U</i> or <i>u</i>	The upper triangular part of array <i>a</i> is to be referenced.
<i>L</i> or <i>l</i>	The lower triangular part of array <i>a</i> is to be referenced.

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

---

<i>alpha</i>	COMPLEX for <code>chemv</code> DOUBLE COMPLEX for <code>zhemv</code>
	Specifies the scalar <i>alpha</i> .
<i>a</i>	COMPLEX for <code>chemv</code> DOUBLE COMPLEX for <code>zhemv</code>
	Array, DIMENSION ( <i>lda</i> , <i>n</i> ). Before entry with <i>uplo</i> = 'U' or 'u', the leading <i>n</i> by <i>n</i> upper triangular part of the array <i>a</i> must contain the upper triangular part of the Hermitian matrix and the strictly lower triangular part of <i>a</i> is not referenced. Before entry with <i>uplo</i> = 'L' or 'l', the leading <i>n</i> by <i>n</i> lower triangular part of the array <i>a</i> must contain the lower triangular part of the Hermitian matrix and the strictly upper triangular part of <i>a</i> is not referenced.
	The imaginary parts of the diagonal elements need not be set and are assumed to be zero.
<i>lda</i>	INTEGER. Specifies the first dimension of <i>a</i> as declared in the calling (sub)program. The value of <i>lda</i> must be at least <code>max(1, n)</code> .
<i>x</i>	COMPLEX for <code>chemv</code> DOUBLE COMPLEX for <code>zhemv</code>
	Array, DIMENSION at least <code>(1 + (n - 1)*abs(incx))</code> . Before entry, the incremented array <i>x</i> must contain the <i>n</i> -element vector <i>x</i> .
<i>incx</i>	INTEGER. Specifies the increment for the elements of <i>x</i> . The value of <i>incx</i> must not be zero.
<i>beta</i>	COMPLEX for <code>chemv</code> DOUBLE COMPLEX for <code>zhemv</code>
	Specifies the scalar <i>beta</i> . When <i>beta</i> is supplied as zero then <i>y</i> need not be set on input.

<i>y</i>	COMPLEX for <code>cher</code> DOUBLE COMPLEX for <code>zher</code>
	Array, DIMENSION at least $(1 + (n - 1) * \text{abs}(incy))$ . Before entry, the incremented array <i>y</i> must contain the <i>n</i> -element vector <i>y</i> .
<i>incy</i>	INTEGER. Specifies the increment for the elements of <i>y</i> . The value of <i>incy</i> must not be zero.

### Output Parameters

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

---

## ?her

Performs a rank-1 update of a Hermitian matrix.

---

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

### Discussion

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

*a* := *alpha*\**x*\*`conjg`(*x*) + *a*,

where:

*alpha* is a real scalar

*x* is an *n*-element vector

*a* is an *n* by *n* Hermitian matrix.

## Input Parameters

*uplo* **CHARACTER\***1. Specifies whether the upper or lower triangular part of the array *a* is to be referenced, as follows:

<i>uplo</i> value	Part of Array <i>a</i> To Be Referenced
<b>U</b> or <b>u</b>	The upper triangular part of array <i>a</i> is to be referenced.
<b>L</b> or <b>l</b>	The lower triangular part of array <i>a</i> is to be referenced.

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

*alpha* **REAL** for **cher**  
**DOUBLE PRECISION** for **zher**

Specifies the scalar *alpha*.

*x* **COMPLEX** for **cher**  
**DOUBLE COMPLEX** for **zher**

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

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

*a* **COMPLEX** for **cher**  
**DOUBLE COMPLEX** for **zher**

Array, **DIMENSION ( lda, n )**. Before entry with *uplo = 'U'* or *'u'*, the leading *n* by *n* upper triangular part of the array *a* must contain the upper triangular part of the Hermitian matrix and the strictly lower triangular part of *a* is not referenced.

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

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

*lda*

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

### Output Parameters

*a*

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

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

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

---

## ?her2

Performs a rank-2 update of a Hermitian matrix.

---

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

### Discussion

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

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

where:

*alpha* is a scalar

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

*a* is an *n* by *n* Hermitian matrix.

## Input Parameters

*uplo*      CHARACTER\*1. Specifies whether the upper or lower triangular part of the array *a* is to be referenced, as follows:

<i>uplo</i> value	Part of Array <i>a</i> To Be Referenced
<i>U</i> or <i>u</i>	The upper triangular part of array <i>a</i> is to be referenced.
<i>L</i> or <i>l</i>	The lower triangular part of array <i>a</i> is to be referenced.

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

*alpha*      COMPLEX for *cher2*  
DOUBLE COMPLEX for *zher2*

Specifies the scalar *alpha*.

*x*      COMPLEX for *cher2*  
DOUBLE COMPLEX for *zher2*

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

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

*y*      COMPLEX for *cher2*  
DOUBLE COMPLEX for *zher2*

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

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

*a*      COMPLEX for *cher2*  
DOUBLE COMPLEX for *zher2*

Array, **DIMENSION ( lda, n )**. Before entry with  $\text{uplo} = \text{'U'}$  or  $\text{'u'}$ , the leading  $n$  by  $n$  upper triangular part of the array  $a$  must contain the upper triangular part of the Hermitian matrix and the strictly lower triangular part of  $a$  is not referenced.

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

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

**lda**

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

### Output Parameters

**a**

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

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

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

---

## ?hpmv

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

---

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

## Discussion

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

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

where:

`alpha` and `beta` are scalars

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

`a` is an  $n$  by  $n$  Hermitian matrix, supplied in packed form.

## Input Parameters

`uplo` **CHARACTER\*1**. Specifies whether the upper or lower triangular part of the matrix `a` is supplied in the packed array `ap`, as follows:

<code>uplo</code> value	Part of Matrix <code>a</code> Supplied
<code>U</code> or <code>u</code>	The upper triangular part of matrix <code>a</code> is supplied in <code>ap</code> .
<code>L</code> or <code>l</code>	The lower triangular part of matrix <code>a</code> is supplied in <code>ap</code> .

`n` **INTEGER**. Specifies the order of the matrix `a`. The value of `n` must be at least zero.

`alpha` **COMPLEX** for `chpmv`  
**DOUBLE COMPLEX** for `zhpmv`

Specifies the scalar `alpha`.

`ap` **COMPLEX** for `chpmv`  
**DOUBLE COMPLEX** for `zhpmv`

Array, **DIMENSION** at least  $((n*(n+1))/2)$ . Before entry with `uplo = 'U'` or `'u'`, the array `ap` must contain the upper triangular part of the Hermitian matrix packed sequentially, column-by-column, so that `ap(1)` contains `a(1, 1)`, `ap(2)` and `ap(3)` contain `a(1, 2)` and `a(2, 2)` respectively, and so on. Before entry with `uplo = 'L'` or `'l'`, the array `ap` must contain the lower triangular part of the Hermitian matrix packed

sequentially, column-by-column, so that  $\text{ap}(1)$  contains  $a(1, 1)$ ,  $\text{ap}(2)$  and  $\text{ap}(3)$  contain  $a(2, 1)$  and  $a(3, 1)$  respectively, and so on.

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

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

## Output Parameters

$y$	Overwritten by the updated vector $y$
-----	---------------------------------------

---

## ?hpr

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

---

```
call chpr ( uplo, n, alpha, x, incx, ap )
call zhpr ( uplo, n, alpha, x, incx, ap )
```

### Discussion

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

$$\alpha := \alpha * x * \text{conj}(x') + \alpha,$$

where:

$\alpha$  is a real scalar

$x$  is an  $n$ -element vector

$\alpha$  is an  $n$  by  $n$  Hermitian matrix, supplied in packed form.

### Input Parameters

$uplo$  **CHARACTER\*1**. Specifies whether the upper or lower triangular part of the matrix  $\alpha$  is supplied in the packed array  $ap$ , as follows:

$uplo$ value	Part of Matrix $\alpha$ Supplied
$U$ or $u$	The upper triangular part of matrix $\alpha$ is supplied in $ap$ .
$L$ or $l$	The lower triangular part of matrix $\alpha$ is supplied in $ap$ .

$n$  **INTEGER**. Specifies the order of the matrix  $\alpha$ . The value of  $n$  must be at least zero.

$\alpha$  **REAL** for **chpr**  
**DOUBLE PRECISION** for **zhpr**  
Specifies the scalar  $\alpha$ .

<i>x</i>	COMPLEX for <code>chpr</code> DOUBLE COMPLEX for <code>zhpr</code>
	Array, DIMENSION at least $(1 + (n - 1) * \text{abs}(incx))$ . Before entry, the incremented array <i>x</i> must contain the <i>n</i> -element vector <i>x</i> .
<i>incx</i>	INTEGER. Specifies the increment for the elements of <i>x</i> . <i>incx</i> must not be zero.
<i>ap</i>	COMPLEX for <code>chpr</code> DOUBLE COMPLEX for <code>zhpr</code>
	Array, DIMENSION at least $((n * (n + 1)) / 2)$ . Before entry with <i>uplo</i> = 'U' or 'u', the array <i>ap</i> must contain the upper triangular part of the Hermitian matrix packed sequentially, column-by-column, so that <i>ap</i> (1) contains <i>a</i> (1, 1), <i>ap</i> (2) and <i>ap</i> (3) contain <i>a</i> (1, 2) and <i>a</i> (2, 2) respectively, and so on. Before entry with <i>uplo</i> = 'L' or 'l', the array <i>ap</i> must contain the lower triangular part of the Hermitian matrix packed sequentially, column-by-column, so that <i>ap</i> (1) contains <i>a</i> (1, 1), <i>ap</i> (2) and <i>ap</i> (3) contain <i>a</i> (2, 1) and <i>a</i> (3, 1) respectively, and so on. The imaginary parts of the diagonal elements need not be set and are assumed to be zero.

## Output Parameters

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

## ?hpr2

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

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

### Discussion

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

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

where:

$\text{alpha}$  is a scalar

$\mathbf{x}$  and  $\mathbf{y}$  are  $n$ -element vectors

$\alpha$  is an  $n$  by  $n$  Hermitian matrix, supplied in packed form.

### Input Parameters

$\text{uplo}$       CHARACTER\*1. Specifies whether the upper or lower triangular part of the matrix  $\alpha$  is supplied in the packed array  $\text{ap}$ , as follows

$\text{uplo}$ value	Part of Matrix $\alpha$ Supplied
$\text{U}$ or $\text{u}$	The upper triangular part of matrix $\alpha$ is supplied in $\text{ap}$ .
$\text{L}$ or $\text{l}$	The lower triangular part of matrix $\alpha$ is supplied in $\text{ap}$ .

$n$       INTEGER. Specifies the order of the matrix  $\alpha$ . The value of  $n$  must be at least zero.

$\text{alpha}$       COMPLEX for `chpr2`  
DOUBLE COMPLEX for `zhpr2`  
Specifies the scalar  $\text{alpha}$ .

<i>x</i>	COMPLEX for <code>chpr2</code> DOUBLE COMPLEX for <code>zhpr2</code>
	Array, dimension at least $(1 + (n - 1) * \text{abs}(incx))$ . Before entry, the incremented array <i>x</i> must contain the <i>n</i> -element vector <i>x</i> .
<i>incx</i>	INTEGER. Specifies the increment for the elements of <i>x</i> . The value of <i>incx</i> must not be zero.
<i>y</i>	COMPLEX for <code>chpr2</code> DOUBLE COMPLEX for <code>zhpr2</code>
	Array, DIMENSION at least $(1 + (n - 1) * \text{abs}(incy))$ . Before entry, the incremented array <i>y</i> must contain the <i>n</i> -element vector <i>y</i> .
<i>incy</i>	INTEGER. Specifies the increment for the elements of <i>y</i> . The value of <i>incy</i> must not be zero.
<i>ap</i>	COMPLEX for <code>chpr2</code> DOUBLE COMPLEX for <code>zhpr2</code>
	Array, DIMENSION at least $((n * (n + 1)) / 2)$ . Before entry with <i>uplo</i> = 'U' or 'u', the array <i>ap</i> must contain the upper triangular part of the Hermitian matrix packed sequentially, column-by-column, so that <i>ap</i> (1) contains <i>a</i> (1, 1), <i>ap</i> (2) and <i>ap</i> (3) contain <i>a</i> (1, 2) and <i>a</i> (2, 2) respectively, and so on. Before entry with <i>uplo</i> = 'L' or 'l', the array <i>ap</i> must contain the lower triangular part of the Hermitian matrix packed sequentially, column-by-column, so that <i>ap</i> (1) contains <i>a</i> (1, 1), <i>ap</i> (2) and <i>ap</i> (3) contain <i>a</i> (2, 1) and <i>a</i> (3, 1) respectively, and so on. The imaginary parts of the diagonal elements need not be set and are assumed to be zero.

## Output Parameters

*ap*

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

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

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

---

## ?sbmv

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

---

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

### Discussion

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

*y* := *alpha*\**a*\**x* + *beta*\**y*,

where:

*alpha* and *beta* are scalars

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

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

## Input Parameters

*uplo*      CHARACTER\*1. Specifies whether the upper or lower triangular part of the band matrix *a* is being supplied, as follows:

<i>uplo</i> value	Part of Matrix <i>a</i> Supplied
U or u	The upper triangular part of matrix <i>a</i> is supplied.
L or l	The lower triangular part of matrix <i>a</i> is supplied.

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

*k*      INTEGER. Specifies the number of super-diagonals of the matrix *a*. The value of *k* must satisfy  $0 \leq k$ .

*alpha*      REAL for ssbmv  
DOUBLE PRECISION for dsbmv

Specifies the scalar *alpha*.

*a*      REAL for ssbmv  
DOUBLE PRECISION for dsbmv

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

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

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

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

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

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

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

**`x`** **REAL** for `ssbmv`  
**DOUBLE PRECISION** for `dsbmv`

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

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

**`beta`** **REAL** for `ssbmv`  
**DOUBLE PRECISION** for `dsbmv`

Specifies the scalar `beta`.

**`y`** **REAL** for `ssbmv`  
**DOUBLE PRECISION** for `dsbmv`

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

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

### Output Parameters

*y*                    Overwritten by the updated vector *y*.

---

## ?spmv

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

---

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

### Discussion

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

*y* := *alpha*\**a*\**x* + *beta*\**y*,

where:

*alpha* and *beta* are scalars

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

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

### Input Parameters

*uplo*                    **CHARACTER\*1.** Specifies whether the upper or lower triangular part of the matrix *a* is supplied in the packed array *ap*, as follows:

---

<i>uplo</i> value	Part of Matrix <i>a</i> Supplied
<i>U</i> or <i>u</i>	The upper triangular part of matrix <i>a</i> is supplied in <i>ap</i> .
<i>L</i> or <i>l</i>	The lower triangular part of matrix <i>a</i> is supplied in <i>ap</i> .

---

---

<i>n</i>	<b>INTEGER.</b> Specifies the order of the matrix <i>a</i> . The value of <i>n</i> must be at least zero.
<i>alpha</i>	<b>REAL</b> for <b>sspmv</b> <b>DOUBLE PRECISION</b> for <b>dspmv</b> Specifies the scalar <i>alpha</i> .
<i>ap</i>	<b>REAL</b> for <b>sspmv</b> <b>DOUBLE PRECISION</b> for <b>dspmv</b> Array, <b>DIMENSION</b> at least $((n*(n + 1))/2)$ . Before entry with <i>uplo</i> = 'U' or 'u', the array <i>ap</i> must contain the upper triangular part of the symmetric matrix packed sequentially, column-by-column, so that <i>ap</i> (1) contains <i>a</i> (1, 1), <i>ap</i> (2) and <i>ap</i> (3) contain <i>a</i> (1, 2) and <i>a</i> (2, 2) respectively, and so on. Before entry with <i>uplo</i> = 'L' or 'l', the array <i>ap</i> must contain the lower triangular part of the symmetric matrix packed sequentially, column-by-column, so that <i>ap</i> (1) contains <i>a</i> (1, 1), <i>ap</i> (2) and <i>ap</i> (3) contain <i>a</i> (2, 1) and <i>a</i> (3, 1) respectively, and so on.
<i>x</i>	<b>REAL</b> for <b>sspmv</b> <b>DOUBLE PRECISION</b> for <b>dspmv</b> Array, <b>DIMENSION</b> at least $(1 + (n - 1)*\text{abs}(incx))$ . Before entry, the incremented array <i>x</i> must contain the <i>n</i> -element vector <i>x</i> .
<i>incx</i>	<b>INTEGER.</b> Specifies the increment for the elements of <i>x</i> . The value of <i>incx</i> must not be zero.
<i>beta</i>	<b>REAL</b> for <b>sspmv</b> <b>DOUBLE PRECISION</b> for <b>dspmv</b> Specifies the scalar <i>beta</i> . When <i>beta</i> is supplied as zero, then <i>y</i> need not be set on input.
<i>y</i>	<b>REAL</b> for <b>sspmv</b> <b>DOUBLE PRECISION</b> for <b>dspmv</b> Array, <b>DIMENSION</b> at least $(1 + (n - 1)*\text{abs}(incy))$ . Before entry, the incremented array <i>y</i> must contain the <i>n</i> -element vector <i>y</i> .

---

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

### Output Parameters

*y*                    Overwritten by the updated vector *y*.

---

## ?spr

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

---

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

### Discussion

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

*a* := *alpha*\**x*\**x'* + *a*,

where:

*alpha* is a real scalar

*x* is an *n*-element vector

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

### Input Parameters

*uplo*                    **CHARACTER\*1.** Specifies whether the upper or lower triangular part of the matrix *a* is supplied in the packed array *ap*, as follows:

<i>uplo</i> value	Part of Matrix <i>a</i> Supplied
<i>U</i> or <i>u</i>	The upper triangular part of matrix <i>a</i> is supplied in <i>ap</i> .
<i>L</i> or <i>l</i>	The lower triangular part of matrix <i>a</i> is supplied in <i>ap</i> .

---

---

<i>n</i>	<b>INTEGER.</b> Specifies the order of the matrix <i>a</i> . The value of <i>n</i> must be at least zero.
<i>alpha</i>	<b>REAL</b> for <b>sspr</b> <b>DOUBLE PRECISION</b> for <b>dspr</b>
	Specifies the scalar <i>alpha</i> .
<i>x</i>	<b>REAL</b> for <b>sspr</b> <b>DOUBLE PRECISION</b> for <b>dspr</b>
	Array, <b>DIMENSION</b> at least $(1 + (n - 1) * \text{abs}(incx))$ . Before entry, the incremented array <i>x</i> must contain the <i>n</i> -element vector <i>x</i> .
<i>incx</i>	<b>INTEGER.</b> Specifies the increment for the elements of <i>x</i> . The value of <i>incx</i> must not be zero.
<i>ap</i>	<b>REAL</b> for <b>sspr</b> <b>DOUBLE PRECISION</b> for <b>dspr</b>
	Array, <b>DIMENSION</b> at least $((n * (n + 1)) / 2)$ . Before entry with <i>uplo</i> = 'U' or 'u', the array <i>ap</i> must contain the upper triangular part of the symmetric matrix packed sequentially, column-by-column, so that <i>ap</i> (1) contains <i>a</i> (1,1), <i>ap</i> (2) and <i>ap</i> (3) contain <i>a</i> (1, 2) and <i>a</i> (2, 2) respectively, and so on.
	Before entry with <i>uplo</i> = 'L' or 'l', the array <i>ap</i> must contain the lower triangular part of the symmetric matrix packed sequentially, column-by-column, so that <i>ap</i> (1) contains <i>a</i> (1,1), <i>ap</i> (2) and <i>ap</i> (3) contain <i>a</i> (2, 1) and <i>a</i> (3, 1) respectively, and so on.

### Output Parameters

<i>ap</i>	With <i>uplo</i> = 'U' or 'u', overwritten by the upper triangular part of the updated matrix. With <i>uplo</i> = 'L' or 'l', overwritten by the lower triangular part of the updated matrix.
-----------	--

---

## ?spr2

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

---

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

### Discussion

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

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

where:

$\alpha$  is a scalar

$x$  and  $y$  are  $n$ -element vectors

$\alpha$  is an  $n$  by  $n$  symmetric matrix, supplied in packed form.

### Input Parameters

**uplo** CHARACTER\*1. Specifies whether the upper or lower triangular part of the matrix  $\alpha$  is supplied in the packed array  $ap$ , as follows:

<b>uplo value</b>	<b>Part of Matrix <math>\alpha</math> Supplied</b>
U or u	The upper triangular part of matrix $\alpha$ is supplied in $ap$ .
L or l	The lower triangular part of matrix $\alpha$ is supplied in $ap$ .

**n** INTEGER. Specifies the order of the matrix  $\alpha$ . The value of  $n$  must be at least zero.

**alpha** REAL for sspr2  
DOUBLE PRECISION for dspr2  
Specifies the scalar  $\alpha$ .

---

<i>x</i>	<small>REAL for sspr2 DOUBLE PRECISION for dspr2</small>
	Array, <small>DIMENSION at least <math>(1 + (n - 1) * \text{abs}(incx))</math>.</small> Before entry, the incremented array <i>x</i> must contain the <i>n</i> -element vector <i>x</i> .
<i>incx</i>	<small>INTEGER. Specifies the increment for the elements of <i>x</i>. The value of <i>incx</i> must not be zero.</small>
<i>y</i>	<small>REAL for sspr2 DOUBLE PRECISION for dspr2</small>
	Array, <small>DIMENSION at least <math>(1 + (n - 1) * \text{abs}(incy))</math>.</small> Before entry, the incremented array <i>y</i> must contain the <i>n</i> -element vector <i>y</i> .
<i>incy</i>	<small>INTEGER. Specifies the increment for the elements of <i>y</i>. The value of <i>incy</i> must not be zero.</small>
<i>ap</i>	<small>REAL for sspr2 DOUBLE PRECISION for dspr2</small>
	Array, <small>DIMENSION at least <math>((n * (n + 1)) / 2)</math>. Before entry with <i>uplo</i> = 'U' or 'u', the array <i>ap</i> must contain the upper triangular part of the symmetric matrix packed sequentially, column-by-column, so that <i>ap</i>(1) contains <i>a</i>(1,1), <i>ap</i>(2) and <i>ap</i>(3) contain <i>a</i>(1,2) and <i>a</i>(2,2) respectively, and so on.</small> Before entry with <i>uplo</i> = 'L' or 'l', the array <i>ap</i> must contain the lower triangular part of the symmetric matrix packed sequentially, column-by-column, so that <i>ap</i> (1) contains <i>a</i> (1,1), <i>ap</i> (2) and <i>ap</i> (3) contain <i>a</i> (2,1) and <i>a</i> (3,1) respectively, and so on.

### Output Parameters

<i>ap</i>	With <i>uplo</i> = 'U' or 'u', overwritten by the upper triangular part of the updated matrix. With <i>uplo</i> = 'L' or 'l', overwritten by the lower triangular part of the updated matrix.
-----------	--

---

## ?symv

*Computes a matrix-vector product for a symmetric matrix.*

---

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

### Discussion

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

$y := \alpha \cdot a \cdot x + \beta \cdot y$ ,

where:

$\alpha$  and  $\beta$  are scalars

$x$  and  $y$  are  $n$ -element vectors

$a$  is an  $n$  by  $n$  symmetric matrix.

### Input Parameters

**uplo**      CHARACTER\*1. Specifies whether the upper or lower triangular part of the array  $a$  is to be referenced, as follows:

<b>uplo</b> value	<b>Part of Array <math>a</math> To Be Referenced</b>
<b>U</b> or <b>u</b>	The upper triangular part of array $a$ is to be referenced.
<b>L</b> or <b>l</b>	The lower triangular part of array $a$ is to be referenced.

**n**      INTEGER. Specifies the order of the matrix  $a$ . The value of  $n$  must be at least zero.

---

<i>alpha</i>	<small>REAL for ssymv DOUBLE PRECISION for dsymv</small>
	Specifies the scalar <i>alpha</i> .
<i>a</i>	<small>REAL for ssymv DOUBLE PRECISION for dsymv</small>
	Array, <small>DIMENSION (lda, n)</small> . Before entry with <i>uplo</i> = 'U' or 'u', the leading <i>n</i> by <i>n</i> upper triangular part of the array <i>a</i> must contain the upper triangular part of the symmetric matrix and the strictly lower triangular part of <i>a</i> is not referenced. Before entry with <i>uplo</i> = 'L' or 'l', the leading <i>n</i> by <i>n</i> lower triangular part of the array <i>a</i> must contain the lower triangular part of the symmetric matrix and the strictly upper triangular part of <i>a</i> is not referenced.
<i>lda</i>	<small>INTEGER</small> . Specifies the first dimension of <i>a</i> as declared in the calling (sub)program. The value of <i>lda</i> must be at least <small>max(1, n)</small> .
<i>x</i>	<small>REAL for ssymv DOUBLE PRECISION for dsymv</small>
	Array, <small>DIMENSION at least (1 + (n - 1)*abs(incx))</small> . Before entry, the incremented array <i>x</i> must contain the <i>n</i> -element vector <i>x</i> .
<i>incx</i>	<small>INTEGER</small> . Specifies the increment for the elements of <i>x</i> . The value of <i>incx</i> must not be zero.
<i>beta</i>	<small>REAL for ssymv DOUBLE PRECISION for dsymv</small>
	Specifies the scalar <i>beta</i> . When <i>beta</i> is supplied as zero, then <i>y</i> need not be set on input.
<i>y</i>	<small>REAL for ssymv DOUBLE PRECISION for dsymv</small>
	Array, <small>DIMENSION at least (1 + (n - 1)*abs(incy))</small> . Before entry, the incremented array <i>y</i> must contain the <i>n</i> -element vector <i>y</i> .

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

### Output Parameters

*y*                    Overwritten by the updated vector *y*.

---

## ?syr

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

---

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

### Discussion

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

*a* := *alpha*\**x*\**x*' + *a*,

where:

*alpha* is a real scalar

*x* is an *n*-element vector

*a* is an *n* by *n* symmetric matrix.

### Input Parameters

*uplo*                    **CHARACTER\*1.** Specifies whether the upper or lower triangular part of the array *a* is to be referenced, as follows:

<i>uplo</i> value	Part of Array <i>a</i> To Be Referenced
<i>U</i> or <i>u</i>	The upper triangular part of array <i>a</i> is to be referenced.
<i>L</i> or <i>l</i>	The lower triangular part of array <i>a</i> is to be referenced.

---

---

<i>n</i>	<b>INTEGER.</b> Specifies the order of the matrix <i>a</i> . The value of <i>n</i> must be at least zero.
<i>alpha</i>	<b>REAL</b> for <b>ssyr</b> <b>DOUBLE PRECISION</b> for <b>dsyr</b> Specifies the scalar <i>alpha</i> .
<i>x</i>	<b>REAL</b> for <b>ssyr</b> <b>DOUBLE PRECISION</b> for <b>dsyr</b> Array, <b>DIMENSION</b> at least $(1 + (n - 1) * \text{abs}(incx))$ . Before entry, the incremented array <i>x</i> must contain the <i>n</i> -element vector <i>x</i> .
<i>incx</i>	<b>INTEGER.</b> Specifies the increment for the elements of <i>x</i> . The value of <i>incx</i> must not be zero.
<i>a</i>	<b>REAL</b> for <b>ssyr</b> <b>DOUBLE PRECISION</b> for <b>dsyr</b> Array, <b>DIMENSION</b> ( <i>lda</i> , <i>n</i> ). Before entry with <i>uplo</i> = 'U' or 'u', the leading <i>n</i> by <i>n</i> upper triangular part of the array <i>a</i> must contain the upper triangular part of the symmetric matrix and the strictly lower triangular part of <i>a</i> is not referenced. Before entry with <i>uplo</i> = 'L' or 'l', the leading <i>n</i> by <i>n</i> lower triangular part of the array <i>a</i> must contain the lower triangular part of the symmetric matrix and the strictly upper triangular part of <i>a</i> is not referenced.
<i>lda</i>	<b>INTEGER.</b> Specifies the first dimension of <i>a</i> as declared in the calling (sub)program. The value of <i>lda</i> must be at least <b>max(1, n)</b> .

### Output Parameters

<i>a</i>	With <i>uplo</i> = 'U' or 'u', the upper triangular part of the array <i>a</i> is overwritten by the upper triangular part of the updated matrix. With <i>uplo</i> = 'L' or 'l', the lower triangular part of the array <i>a</i> is overwritten by the lower triangular part of the updated matrix.
----------	--

---

## ?syr2

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

---

```
call ssyr2( uplo, n, alpha, x, incx, y, incy, a, lda )
call dsyr2( uplo, n, alpha, x, incx, y, incy, a, lda )
```

### Discussion

The ?syr2 routines perform a matrix-vector operation defined as

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

where:

$\alpha$  is a scalar

$x$  and  $y$  are  $n$ -element vectors

$a$  is an  $n$  by  $n$  symmetric matrix.

### Input Parameters

$uplo$  CHARACTER\*1. Specifies whether the upper or lower triangular part of the array  $a$  is to be referenced, as follows:

$uplo$ value	Part of Array $a$ To Be Referenced
$U$ or $u$	The upper triangular part of array $a$ is to be referenced.
$L$ or $l$	The lower triangular part of array $a$ is to be referenced.

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

$alpha$  REAL for ssyr2  
DOUBLE PRECISION for dsyr2

Specifies the scalar  $\alpha$ .

---

<i>x</i>	<small>REAL for ssyr2 DOUBLE PRECISION for dsyr2</small>
	Array, DIMENSION at least $(1 + (n - 1) * \text{abs}(incx))$ . Before entry, the incremented array <i>x</i> must contain the <i>n</i> -element vector <i>x</i> .
<i>incx</i>	<small>INTEGER. Specifies the increment for the elements of <i>x</i>. The value of <i>incx</i> must not be zero.</small>
<i>y</i>	<small>REAL for ssyr2 DOUBLE PRECISION for dsyr2</small>
	Array, DIMENSION at least $(1 + (n - 1) * \text{abs}(incy))$ . Before entry, the incremented array <i>y</i> must contain the <i>n</i> -element vector <i>y</i> .
<i>incy</i>	<small>INTEGER. Specifies the increment for the elements of <i>y</i>. The value of <i>incy</i> must not be zero.</small>
<i>a</i>	<small>REAL for ssyr2 DOUBLE PRECISION for dsyr2</small>
	Array, DIMENSION ( <i>lda</i> , <i>n</i> ). Before entry with <i>uplo</i> = 'U' or 'u', the leading <i>n</i> by <i>n</i> upper triangular part of the array <i>a</i> must contain the upper triangular part of the symmetric matrix and the strictly lower triangular part of <i>a</i> is not referenced. Before entry with <i>uplo</i> = 'L' or 'l', the leading <i>n</i> by <i>n</i> lower triangular part of the array <i>a</i> must contain the lower triangular part of the symmetric matrix and the strictly upper triangular part of <i>a</i> is not referenced.
<i>lda</i>	<small>INTEGER. Specifies the first dimension of <i>a</i> as declared in the calling (sub)program. The value of <i>lda</i> must be at least <math>\max(1, n)</math>.</small>

## Output Parameters

*a*

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

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

---

## ?tbmv

*Computes a matrix-vector product using  
a triangular band matrix.*

---

```
call stbmv ( uplo, trans, diag, n, k, a, lda, x, incx )
call dtbmv ( uplo, trans, diag, n, k, a, lda, x, incx )
call ctbmv ( uplo, trans, diag, n, k, a, lda, x, incx )
call ztbmv ( uplo, trans, diag, n, k, a, lda, x, incx )
```

## Discussion

The ?tbmv routines perform one of the matrix-vector operations defined as

*x* := *a*\**x*, or *x* := *a'*\**x*, or *x* := *conjg(a')*\**x*,

where:

*x* is an *n*-element vector

*a* is an *n* by *n* unit, or non-unit, upper or lower triangular band matrix, with (*k* + 1) diagonals.

## Input Parameters

*uplo* CHARACTER\*1. Specifies whether the matrix is an upper or lower triangular matrix, as follows:

<i>uplo</i> value	Matrix <i>a</i>
<i>U</i> or <i>u</i>	An upper triangular matrix.
<i>L</i> or <i>l</i>	A lower triangular matrix.

*trans* CHARACTER\*1. Specifies the operation to be performed, as follows:

<i>trans</i> value	Operation to be Performed
<i>N</i> or <i>n</i>	<i>x</i> := <i>a</i> * <i>x</i>
<i>T</i> or <i>t</i>	<i>x</i> := <i>a</i> ' * <i>x</i>
<i>C</i> or <i>c</i>	<i>x</i> := conjg( <i>a</i> ') * <i>x</i>

*diag* CHARACTER\*1. Specifies whether or not *a* is unit triangular, as follows:

<i>diag</i> value	Matrix <i>a</i>
<i>U</i> or <i>u</i>	Matrix <i>a</i> is assumed to be unit triangular.
<i>N</i> or <i>n</i>	Matrix <i>a</i> is not assumed to be unit triangular.

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

*k* INTEGER. On entry with *uplo* = 'U' or 'u', *k* specifies the number of super-diagonals of the matrix *a*. On entry with *uplo* = 'L' or 'l', *k* specifies the number of sub-diagonals of the matrix *a*. The value of *k* must satisfy  $0 \leq k$ .

*a* REAL for *stbmv*  
DOUBLE PRECISION for *dtbmv*  
COMPLEX for *ctbmv*  
DOUBLE COMPLEX for *ztbmv*

Array, DIMENSION (*lda*, *n*). Before entry with *uplo* = 'U' or 'u', the leading  $(k + 1)$  by *n* part of the array *a* must contain the upper triangular band part of the matrix of coefficients, supplied column-by-column, with the leading diagonal of the matrix in row  $(k + 1)$

of the array, the first super-diagonal starting at position 2 in row  $k$ , and so on. The top left  $k$  by  $k$  triangle of the array  $a$  is not referenced. The following program segment transfers an upper triangular band matrix from conventional full matrix storage to band storage:

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

Before entry with  $uplo = 'L'$  or ' $U$ ', the leading  $(k+1)$  by  $n$  part of the array  $a$  must contain the lower triangular band part of the matrix of coefficients, supplied column-by-column, with the leading diagonal of the matrix in row 1 of the array, the first sub-diagonal starting at position 1 in row 2, and so on. The bottom right  $k$  by  $k$  triangle of the array  $a$  is not referenced. The following program segment transfers a lower triangular band matrix from conventional full matrix storage to band storage:

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

Note that when  $diag = 'U'$  or ' $u$ ', the elements of the array  $a$  corresponding to the diagonal elements of the matrix are not referenced, but are assumed to be unity.

#### *lda*

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

---

<b><i>x</i></b>	<small>REAL for <b>stbmv</b> DOUBLE PRECISION for <b>dtbmv</b> COMPLEX for <b>ctbmv</b> DOUBLE COMPLEX for <b>ztbmv</b></small>
	<small>Array, DIMENSION at least <math>(1 + (n - 1) * \text{abs}(incx))</math>. Before entry, the incremented array <b>x</b> must contain the <i>n</i>-element vector <b>x</b>.</small>
<b><i>incx</i></b>	<small>INTEGER. Specifies the increment for the elements of <b>x</b>. The value of <b>incx</b> must not be zero.</small>

### Output Parameters

<b><i>x</i></b>	<small>Overwritten with the transformed vector <b>x</b>.</small>
-----------------	--

---

## ?tbsv

*Solves a system of linear equations whose coefficients are in a triangular band matrix.*

---

```
call stbsv ( UPLO, TRANS, DIAG, N, K, A, LDA, X, INCX )
call dtbsv ( uplo, trans, diag, n, k, a, lda, x, incx )
call ctbsv ( uplo, trans, diag, n, k, a, lda, x, incx )
call ztbsv ( uplo, trans, diag, n, k, a, lda, x, incx )
```

### Discussion

The **?tbsv** routines solve one of the following systems of equations:

**a\*x = b**, or **a'\*x = b**, or **conjg(a')\*x = b**,

where:

**b** and **x** are *n*-element vectors

**a** is an *n* by *n* unit, or non-unit, upper or lower triangular band matrix, with  $(k + 1)$  diagonals.

The routine does not test for singularity or near-singularity. Such tests must be performed before calling this routine.

### Input Parameters

*uplo*            CHARACTER\*1. Specifies whether the matrix is an upper or lower triangular matrix, as follows:

<i>uplo</i> value	Matrix <i>a</i>
U or u	An upper triangular matrix.
L or l	A lower triangular matrix.

*trans*            CHARACTER\*1. Specifies the operation to be performed, as follows:

<i>trans</i> value	Operation to be Performed
N or n	$a * x = b$
T or t	$a' * x = b$
C or c	$\text{conjg}(a') * x = b$

*diag*            CHARACTER\*1. Specifies whether or not *a* is unit triangular, as follows:

<i>diag</i> value	Matrix <i>a</i>
U or u	Matrix <i>a</i> is assumed to be unit triangular.
N or n	Matrix <i>a</i> is not assumed to be unit triangular.

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

*k*            INTEGER. On entry with *uplo* = 'U' or 'u', *k* specifies the number of super-diagonals of the matrix *a*. On entry with *uplo* = 'L' or 'l', *k* specifies the number of sub-diagonals of the matrix *a*. The value of *k* must satisfy  $0 \leq k$ .

*a*

REAL for **stbsv**  
 DOUBLE PRECISION for **dtbsv**  
 COMPLEX for **ctbsv**  
 DOUBLE COMPLEX for **ztbsv**

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

The following program segment transfers an upper triangular band matrix from conventional full matrix storage to band storage:

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

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

The following program segment transfers a lower triangular band matrix from conventional full matrix storage to band storage:

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

When  $\text{diag} = \text{'U'}$  or  $\text{'u'}$ , the elements of the array  $a$  corresponding to the diagonal elements of the matrix are not referenced, but are assumed to be unity.

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

$x$  **REAL** for  $\text{stbsv}$   
**DOUBLE PRECISION** for  $\text{dtbsv}$   
**COMPLEX** for  $\text{ctbsv}$   
**DOUBLE COMPLEX** for  $\text{ztbsv}$

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

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

## Output Parameters

$x$  Overwritten with the solution vector  $x$ .

---

## ?tpmv

*Computes a matrix-vector product using a triangular packed matrix.*

---

```
call stpmv ( uplo, trans, diag, n, ap, x, incx )
call dtpmv ( uplo, trans, diag, n, ap, x, incx )
call ctpmv ( uplo, trans, diag, n, ap, x, incx )
call ztpmv ( uplo, trans, diag, n, ap, x, incx )
```

## Discussion

The **?tpmv** routines perform one of the matrix-vector operations defined as

$x := a * x$ , or  $x := a' * x$ , or  $x := \text{conjg}(a') * x$ ,

where:

*x* is an *n*-element vector

*a* is an *n* by *n* unit, or non-unit, upper or lower triangular matrix, supplied in packed form.

### Input Parameters

*uplo*      CHARACTER\*1. Specifies whether the matrix *a* is an upper or lower triangular matrix, as follows:

<i>uplo</i> value	Matrix <i>a</i>
<i>U</i> or <i>u</i>	An upper triangular matrix.
<i>L</i> or <i>l</i>	A lower triangular matrix.

*trans*      CHARACTER\*1. Specifies the operation to be performed, as follows:

<i>trans</i> value	Operation To Be Performed
<i>N</i> or <i>n</i>	<i>x</i> := <i>a</i> * <i>x</i>
<i>T</i> or <i>t</i>	<i>x</i> := <i>a</i> ' * <i>x</i>
<i>C</i> or <i>c</i>	<i>x</i> := conjg( <i>a</i> ') * <i>x</i>

*diag*      CHARACTER\*1. Specifies whether or not *a* is unit triangular, as follows:

<i>diag</i> value	Matrix <i>a</i>
<i>U</i> or <i>u</i>	Matrix <i>a</i> is assumed to be unit triangular.
<i>N</i> or <i>n</i>	Matrix <i>a</i> is not assumed to be unit triangular.

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

*ap*      REAL for *stpmv*  
 DOUBLE PRECISION for *dtpmv*  
 COMPLEX for *ctpmv*  
 DOUBLE COMPLEX for *ztpmv*

Array, **DIMENSION** at least  $((n * (n + 1)) / 2)$ . Before entry with  $uplo = 'U'$  or ' $u$ ', the array  $ap$  must contain the upper triangular matrix packed sequentially, column-by-column, so that  $ap(1)$  contains  $a(1, 1)$ ,  $ap(2)$  and  $ap(3)$  contain  $a(1, 2)$  and  $a(2, 2)$  respectively, and so on. Before entry with  $uplo = 'L'$  or ' $l$ ', the array  $ap$  must contain the lower triangular matrix packed sequentially, column-by-column, so that  $ap(1)$  contains  $a(1, 1)$ ,  $ap(2)$  and  $ap(3)$  contain  $a(2, 1)$  and  $a(3, 1)$  respectively, and so on. When  $diag = 'U'$  or ' $u$ ', the diagonal elements of  $a$  are not referenced, but are assumed to be unity.

$x$

**REAL** for **stpmv**

**DOUBLE PRECISION** for **dtpmv**

**COMPLEX** for **ctpmv**

**DOUBLE COMPLEX** for **ztpmv**

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

$incx$

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

The value of  $incx$  must not be zero.

## Output Parameters

$x$

Overwritten with the transformed vector  $x$ .

---

## ?tpsv

Solves a system of linear equations  
whose coefficients are in a triangular  
packed matrix.

---

```
call stpsv ( uplo, trans, diag, n, ap, x, incx )
call dtpsv ( uplo, trans, diag, n, ap, x, incx )
call ctpsv ( uplo, trans, diag, n, ap, x, incx )
call ztpsv ( uplo, trans, diag, n, ap, x, incx )
```

### Discussion

The ?tpsv routines solve one of the following systems of equations

$a * x = b$ , or  $a' * x = b$ , or  $\text{conjg}(a') * x = b$ ,

where:

$b$  and  $x$  are  $n$ -element vectors

$a$  is an  $n$  by  $n$  unit, or non-unit, upper or lower triangular matrix, supplied in packed form.

This routine does not test for singularity or near-singularity. Such tests must be performed before calling this routine.

### Input Parameters

*uplo*            CHARACTER\*1. Specifies whether the matrix  $a$  is an upper or lower triangular matrix, as follows:

<i>uplo</i> value	Matrix $a$
$U$ or $u$	An upper triangular matrix.
$L$ or $l$	A lower triangular matrix.

*trans*

**CHARACTER\*1.** Specifies the operation to be performed, as follows:

<i>trans</i> value	Operation To Be Performed
N or n	$a * x = b$
T or t	$a' * x = b$
C or c	$\text{conjg}(a') * x = b$

*diag*

**CHARACTER\*1.** Specifies whether or not *a* is unit triangular, as follows:

<i>diag</i> value	Matrix <i>a</i>
U or u	Matrix <i>a</i> is assumed to be unit triangular.
N or n	Matrix <i>a</i> is not assumed to be unit triangular.

*n*

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

*ap*

**REAL** for **stpsv**  
**DOUBLE PRECISION** for **dtpsv**  
**COMPLEX** for **ctpsv**  
**DOUBLE COMPLEX** for **ztpsv**

Array, **DIMENSION** at least  $((n * (n + 1)) / 2)$ . Before entry with *uplo* = 'U' or 'u', the array *ap* must contain the upper triangular matrix packed sequentially, column-by-column, so that *ap*(1) contains *a*(1, 1), *ap*(2) and *ap*(3) contain *a*(1, 2) and *a*(2, 2) respectively, and so on. Before entry with *uplo* = 'L' or 'l', the array *ap* must contain the lower triangular matrix packed sequentially, column-by-column, so that *ap*(1) contains *a*(1, 1), *ap*(2) and *ap*(3) contain *a*(2, 1) and *a*(3, 1) respectively, and so on. When *diag* = 'U' or 'u', the diagonal elements of *a* are not referenced, but are assumed to be unity.

---

**x**                    REAL for `stpsv`  
DOUBLE PRECISION for `dtpsv`  
COMPLEX for `ctpsv`  
DOUBLE COMPLEX for `ztpsv`  
                         Array, DIMENSION at least  $(1 + (n - 1) * \text{abs}(incx))$ .  
                         Before entry, the incremented array **x** must contain the  
                         *n*-element right-hand side vector **b**.

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

### Output Parameters

**x**                    Overwritten with the solution vector **x**.

---

## ?trmv

*Computes a matrix-vector product using  
a triangular matrix.*

---

```
call strmv ( uplo, trans, diag, n, a, lda, x, incx )
call dtrmv ( uplo, trans, diag, n, a, lda, x, incx )
call ctrmv ( uplo, trans, diag, n, a, lda, x, incx )
call ztrmv ( uplo, trans, diag, n, a, lda, x, incx )
```

### Discussion

The `?trmv` routines perform one of the following matrix-vector operations defined as

**x** := **a**\***x** or **x** := **a'**\***x** or **x** := `conjg(a')`\***x**,

where:

**x** is an *n*-element vector

**a** is an *n* by *n* unit, or non-unit, upper or lower triangular matrix.

## Input Parameters

*uplo*      CHARACTER\*1. Specifies whether the matrix *a* is an upper or lower triangular matrix, as follows:

<i>uplo</i> value	Matrix <i>a</i>
U or u	An upper triangular matrix.
L or l	A lower triangular matrix.

*trans*      CHARACTER\*1. Specifies the operation to be performed, as follows:

<i>trans</i> value	Operation To Be Performed
N or n	<i>x</i> := <i>a</i> * <i>x</i>
T or t	<i>x</i> := <i>a</i> ' * <i>x</i>
C or c	<i>x</i> := conjg( <i>a</i> ')* <i>x</i>

*diag*      CHARACTER\*1. Specifies whether or not *a* is unit triangular, as follows:

<i>diag</i> value	Matrix <i>a</i>
U or u	Matrix <i>a</i> is assumed to be unit triangular.
N or n	Matrix <i>a</i> is not assumed to be unit triangular.

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

*a*      REAL for **strmv**  
 DOUBLE PRECISION for **dtrmv**  
 COMPLEX for **ctrmv**  
 DOUBLE COMPLEX for **ztrmv**  
 Array, DIMENSION (*lda*, *n*). Before entry with *uplo* = 'U' or 'u', the leading *n* by *n* upper triangular part of the array *a* must contain the upper triangular matrix and the strictly lower triangular part of *a* is not referenced. Before entry with *uplo* = 'L' or 'l', the leading *n* by *n* lower triangular part of the array *a* must

contain the lower triangular matrix and the strictly upper triangular part of  $a$  is not referenced. When  $\text{diag} = \text{'U'}$  or  $\text{'u'}$ , the diagonal elements of  $a$  are not referenced either, but are assumed to be unity.

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

$x$  **REAL** for  $\text{strmv}$   
**DOUBLE PRECISION** for  $\text{dtrmv}$   
**COMPLEX** for  $\text{ctrmv}$   
**DOUBLE COMPLEX** for  $\text{ztrmv}$

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

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

### Output Parameters

$x$  Overwritten with the transformed vector  $x$ .

## ?trsv

*Solves a system of linear equations whose coefficients are in a triangular matrix.*

---

```
call strsv ( uplo, trans, diag, n, a, lda, x, incx )
call dtrsv ( uplo, trans, diag, n, a, lda, x, incx )
call ctrsv ( uplo, trans, diag, n, a, lda, x, incx )
call ztrsv ( uplo, trans, diag, n, a, lda, x, incx )
```

## Discussion

The `?trsv` routines solve one of the systems of equations:

$a * x = b$  or  $a' * x = b$ , or  $\text{conjg}(a') * x = b$ ,

where:

$b$  and  $x$  are  $n$ -element vectors

$a$  is an  $n$  by  $n$  unit, or non-unit, upper or lower triangular matrix.

The routine does not test for singularity or near-singularity. Such tests must be performed before calling this routine.

## Input Parameters

`uplo`      CHARACTER\*1. Specifies whether the matrix is an upper or lower triangular matrix, as follows:

<code>uplo</code> value	Matrix $a$
<code>U</code> or <code>u</code>	An upper triangular matrix.
<code>L</code> or <code>l</code>	A lower triangular matrix.

`trans`      CHARACTER\*1. Specifies the operation to be performed, as follows:

<code>trans</code> value	Operation To Be Performed
<code>N</code> or <code>n</code>	$a * x = b$
<code>T</code> or <code>t</code>	$a' * x = b$
<code>C</code> or <code>c</code>	$\text{conjg}(a') * x = b$

`diag`      CHARACTER\*1. Specifies whether or not  $a$  is unit triangular, as follows:

<code>diag</code> value	Matrix $a$
<code>U</code> or <code>u</code>	Matrix $a$ is assumed to be unit triangular.
<code>N</code> or <code>n</code>	Matrix $a$ is not assumed to be unit triangular.

`n`      INTEGER. Specifies the order of the matrix  $a$ . The value of  $n$  must be at least zero.

<i>a</i>	<small>REAL for <code>strsv</code> DOUBLE PRECISION for <code>dtrsv</code> COMPLEX for <code>ctrsv</code> DOUBLE COMPLEX for <code>ztrsv</code></small>
	<small>Array, <code>DIMENSION ( lda, n )</code>. Before entry with <code>uplo = 'U'</code> or <code>'u'</code>, the leading <i>n</i> by <i>n</i> upper triangular part of the array <i>a</i> must contain the upper triangular matrix and the strictly lower triangular part of <i>a</i> is not referenced. Before entry with <code>uplo = 'L'</code> or <code>'l'</code>, the leading <i>n</i> by <i>n</i> lower triangular part of the array <i>a</i> must contain the lower triangular matrix and the strictly upper triangular part of <i>a</i> is not referenced. When <code>diag = 'U'</code> or <code>'u'</code>, the diagonal elements of <i>a</i> are not referenced either, but are assumed to be unity.</small>
<i>lda</i>	<small>INTEGER. Specifies the first dimension of <i>a</i> as declared in the calling (sub)program. The value of <i>lda</i> must be at least <code>max( 1, n )</code>.</small>
<i>x</i>	<small>REAL for <code>strsv</code> DOUBLE PRECISION for <code>dtrsv</code> COMPLEX for <code>ctrsv</code> DOUBLE COMPLEX for <code>ztrsv</code></small>
	<small>Array, <code>DIMENSION</code> at least <code>( 1 + ( n - 1 ) * abs( incx ) )</code>. Before entry, the incremented array <i>x</i> must contain the <i>n</i>-element right-hand side vector <i>b</i>.</small>
<i>incx</i>	<small>INTEGER. Specifies the increment for the elements of <i>x</i>. The value of <i>incx</i> must not be zero.</small>

### Output Parameters

<i>x</i>	Overwritten with the solution vector <i>x</i> .
----------	---

## BLAS Level 3 Routines

BLAS Level 3 routines perform matrix-matrix operations. Table 2-3 lists the BLAS Level 3 routine groups and the data types associated with them.

**Table 2-3**

**BLAS Level 3 Routine Groups and Their Data Types**

Routine Group	Data Types	Description
<a href="#">?gemm</a>	s, d, c, z	Matrix-matrix product of general matrices
<a href="#">?hemm</a>	c, z	Matrix-matrix product of Hermitian matrices
<a href="#">?herk</a>	c, z	Rank-k update of Hermitian matrices
<a href="#">?her2k</a>	c, z	Rank-2k update of Hermitian matrices
<a href="#">?symm</a>	s, d, c, z	Matrix-matrix product of symmetric matrices
<a href="#">?syrk</a>	s, d, c, z	Rank-k update of symmetric matrices
<a href="#">?syr2k</a>	s, d, c, z	Rank-2k update of symmetric matrices
<a href="#">?trmm</a>	s, d, c, z	Matrix-matrix product of triangular matrices
<a href="#">?trsm</a>	s, d, c, z	Linear matrix-matrix solution for triangular matrices

## Symmetric Multiprocessing Version of MKL

Many applications spend considerable time for executing BLAS level 3 routines. This time can be scaled by the number of processors available on the system through using the symmetric multiprocessing (SMP) feature built into the MKL Library. The performance enhancements based on the parallel use of the processors are available without any programming effort on your part.

To enhance performance, the library uses the following methods:

- The operation of BLAS level 3 matrix-matrix functions permits to restructure the code in a way which increases the localization of data reference, enhances cache memory use, and reduces the dependency on the memory bus.

- 
- Once the code has been effectively blocked as described above, one of the matrices is distributed across the processors to be multiplied by the second matrix. Such distribution ensures effective cache management which reduces the dependency on the memory bus performance and brings good scaling results.

---

## ?gemm

*Computes a scalar-matrix-matrix product and adds the result to a scalar-matrix product.*

---

```
call sgemm ( transa, transb, m, n, k, alpha, a, lda,
            b, ldb, beta, c, ldc )
call dgemm ( transa, transb, m, n, k, alpha, a, lda,
            b, ldb, beta, c, ldc )
call cgemm ( transa, transb, m, n, k, alpha, a, lda,
            b, ldb, beta, c, ldc )
call zgemm ( transa, transb, m, n, k, alpha, a, lda,
            b, ldb, beta, c, ldc )
```

### Discussion

The ?gemm routines perform a matrix-matrix operation with general matrices. The operation is defined as

$c := \alpha \cdot \text{op}(a) \cdot \text{op}(b) + \beta \cdot c,$

where:

$\text{op}(x)$  is one of  $\text{op}(x) = x$  or  $\text{op}(x) = x'$  or  $\text{op}(x) = \text{conjg}(x')$ ,

$\alpha$  and  $\beta$  are scalars

$a$ ,  $b$  and  $c$  are matrices:

$\text{op}(a)$  is an  $m$  by  $k$  matrix

$\text{op}(b)$  is a  $k$  by  $n$  matrix

$c$  is an  $m$  by  $n$  matrix.

## Input Parameters

*transa*      CHARACTER\*1. Specifies the form of  $\text{op}(a)$  to be used in the matrix multiplication as follows:

<i>transa</i> value	Form of $\text{op}(a)$
N or n	$\text{op}(a) = a$
T or t	$\text{op}(a) = a'$
C or c	$\text{op}(a) = \text{conjg}(a')$

*transb*      CHARACTER\*1. Specifies the form of  $\text{op}(b)$  to be used in the matrix multiplication as follows:

<i>transb</i> value	Form of $\text{op}(b)$
N or n	$\text{op}(b) = b$
T or t	$\text{op}(b) = b'$
C or c	$\text{op}(b) = \text{conjg}(b')$

*m*      INTEGER. Specifies the number of rows of the matrix  $\text{op}(a)$  and of the matrix *c*. The value of *m* must be at least zero.

*n*      INTEGER. Specifies the number of columns of the matrix  $\text{op}(b)$  and the number of columns of the matrix *c*. The value of *n* must be at least zero.

*k*      INTEGER. Specifies the number of columns of the matrix  $\text{op}(a)$  and the number of rows of the matrix  $\text{op}(b)$ . The value of *k* must be at least zero.

*alpha*      REAL for sgemm  
DOUBLE PRECISION for dgemm  
COMPLEX for cgemm  
DOUBLE COMPLEX for zgemm

Specifies the scalar *alpha*.

---

<i>a</i>	<p>REAL for sgemm DOUBLE PRECISION for dgemm COMPLEX for cgemm DOUBLE COMPLEX for zgemm</p> <p>Array, DIMENSION (<i>lda</i>, <i>ka</i>), where <i>ka</i> is <i>k</i> when <i>transa</i> = 'N' or 'n', and is <i>m</i> otherwise. Before entry with <i>transa</i> = 'N' or 'n', the leading <i>m</i> by <i>k</i> part of the array <i>a</i> must contain the matrix <i>a</i>, otherwise the leading <i>k</i> by <i>m</i> part of the array <i>a</i> must contain the matrix <i>a</i>.</p>
<i>lda</i>	<p>INTEGER. Specifies the first dimension of <i>a</i> as declared in the calling (sub)program. When <i>transa</i> = 'N' or 'n', then <i>lda</i> must be at least <i>max(1, m)</i>, otherwise <i>lda</i> must be at least <i>max(1, k)</i>.</p>
<i>b</i>	<p>REAL for sgemm DOUBLE PRECISION for dgemm COMPLEX for cgemm DOUBLE COMPLEX for zgemm</p> <p>Array, DIMENSION (<i>ldb</i>, <i>kb</i>), where <i>kb</i> is <i>n</i> when <i>transb</i> = 'N' or 'n', and is <i>k</i> otherwise. Before entry with <i>transb</i> = 'N' or 'n', the leading <i>k</i> by <i>n</i> part of the array <i>b</i> must contain the matrix <i>b</i>, otherwise the leading <i>n</i> by <i>k</i> part of the array <i>b</i> must contain the matrix <i>b</i>.</p>
<i>ldb</i>	<p>INTEGER. Specifies the first dimension of <i>b</i> as declared in the calling (sub)program. When <i>transb</i> = 'N' or 'n', then <i>ldb</i> must be at least <i>max(1, k)</i>, otherwise <i>ldb</i> must be at least <i>max(1, n)</i>.</p>
<i>beta</i>	<p>REAL for sgemm DOUBLE PRECISION for dgemm COMPLEX for cgemm DOUBLE COMPLEX for zgemm</p> <p>Specifies the scalar <i>beta</i>. When <i>beta</i> is supplied as zero, then <i>c</i> need not be set on input.</p>

<i>c</i>	REAL for sgemm DOUBLE PRECISION for dgemm COMPLEX for cgemm DOUBLE COMPLEX for zgemm
<i>ldc</i>	Array, DIMENSION ( <i>ldc</i> , <i>n</i> ). Before entry, the leading <i>m</i> by <i>n</i> part of the array <i>c</i> must contain the matrix <i>c</i> , except when <i>beta</i> is zero, in which case <i>c</i> need not be set on entry.

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

### Output Parameters

<i>c</i>	Overwritten by the <i>m</i> by <i>n</i> matrix $(\alpha \cdot \text{op}(a) \cdot \text{op}(b) + \beta \cdot c)$ .
----------	--

---

## ?hemm

*Computes a scalar-matrix-matrix product (either one of the matrices is Hermitian) and adds the result to scalar-matrix product.*

---

```
call chemm ( side, uplo, m, n, alpha, a, lda, b,
            ldb, beta, c, ldc )
call zhemm ( side, uplo, m, n, alpha, a, lda, b,
            ldb, beta, c, ldc )
```

### Discussion

The ?hemm routines perform a matrix-matrix operation using Hermitian matrices. The operation is defined as

$$c := \alpha \cdot a \cdot b + \beta \cdot c$$

or

*c* := *alpha*\**b*\**a* + *beta*\**c*,

where:

*alpha* and *beta* are scalars

*a* is an Hermitian matrix

*b* and *c* are *m* by *n* matrices.

### Input Parameters

*side*      CHARACTER\*1. Specifies whether the Hermitian matrix *a* appears on the left or right in the operation as follows:

<i>side</i> value	Operation To Be Performed
L or l	<i>c</i> := <i>alpha</i> * <i>a</i> * <i>b</i> + <i>beta</i> * <i>c</i>
R or r	<i>c</i> := <i>alpha</i> * <i>b</i> * <i>a</i> + <i>beta</i> * <i>c</i>

*uplo*      CHARACTER\*1. Specifies whether the upper or lower triangular part of the Hermitian matrix *a* is to be referenced as follows:

<i>uplo</i> value	Part of Matrix <i>a</i> To Be Referenced
U or u	Only the upper triangular part of the Hermitian matrix is to be referenced.
L or l	Only the lower triangular part of the Hermitian matrix is to be referenced.

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

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

*alpha*      COMPLEX for *chemm*  
DOUBLE COMPLEX for *zhemm*  
Specifies the scalar *alpha*.

*a*COMPLEX for `chemm`DOUBLE COMPLEX for `zhemm`

Array, DIMENSION (*lda, ka*), where *ka* is *m* when *side* = 'L' or 'l' and is *n* otherwise. Before entry with *side* = 'L' or 'l', the *m* by *m* part of the array *a* must contain the Hermitian matrix, such that when *uplo* = 'U' or 'u', the leading *m* by *m* upper triangular part of the array *a* must contain the upper triangular part of the Hermitian matrix and the strictly lower triangular part of *a* is not referenced, and when *uplo* = 'L' or 'l', the leading *m* by *m* lower triangular part of the array *a* must contain the lower triangular part of the Hermitian matrix, and the strictly upper triangular part of *a* is not referenced. Before entry with *side* = 'R' or 'r', the *n* by *n* part of the array *a* must contain the Hermitian matrix, such that when *uplo* = 'U' or 'u', the leading *n* by *n* upper triangular part of the array *a* must contain the upper triangular part of the Hermitian matrix and the strictly lower triangular part of *a* is not referenced, and when *uplo* = 'L' or 'l', the leading *n* by *n* lower triangular part of the array *a* must contain the lower triangular part of the Hermitian matrix, and the strictly upper triangular part of *a* is not referenced. The imaginary parts of the diagonal elements need not be set, they are assumed to be zero.

*lda*

INTEGER. Specifies the first dimension of *a* as declared in the calling (sub) program. When *side* = 'L' or 'l' then *lda* must be at least `max(1, m)`, otherwise *lda* must be at least `max(1, n)`.

*b*COMPLEX for `chemm`DOUBLE COMPLEX for `zhemm`

Array, DIMENSION (*ldb, n*). Before entry, the leading *m* by *n* part of the array *b* must contain the matrix *b*.

*ldb*

INTEGER. Specifies the first dimension of *b* as declared in the calling (sub) program. The value of *ldb* must be at least `max(1, m)`.

<i>beta</i>	COMPLEX for <code>chemm</code> DOUBLE COMPLEX for <code>zhemm</code>
	Specifies the scalar <i>beta</i> . When <i>beta</i> is supplied as zero, then <i>c</i> need not be set on input.
<i>c</i>	COMPLEX for <code>chemm</code> DOUBLE COMPLEX for <code>zhemm</code>
	Array, DIMENSION ( <i>c</i> , <i>n</i> ). Before entry, the leading <i>m</i> by <i>n</i> part of the array <i>c</i> must contain the matrix <i>c</i> , except when <i>beta</i> is zero, in which case <i>c</i> need not be set on entry.
<i>ldc</i>	INTEGER. Specifies the first dimension of <i>c</i> as declared in the calling (sub)program. The value of <i>ldc</i> must be at least <code>max(1, m)</code> .

### Output Parameters

<i>c</i>	Overwritten by the <i>m</i> by <i>n</i> updated matrix.
----------	---

## ?herk

Performs a rank-*n* update of a Hermitian matrix.

```
call cherk ( uplo, trans, n, k, alpha, a, lda, beta, c,
            ldc )
call zherk ( uplo, trans, n, k, alpha, a, lda, beta, c,
            ldc )
```

### Discussion

The ?herk routines perform a matrix-matrix operation using Hermitian matrices. The operation is defined as

*c* := *alpha*\**a*\**conjg(a')* + *beta*\**c*,

or

*c* := *alpha*\**conjg(a')*\**a* + *beta*\**c*,

where:

*alpha* and *beta* are real scalars

*c* is an *n* by *n* Hermitian matrix

*a* is an *n* by *k* matrix in the first case and a *k* by *n* matrix in the second case.

### Input Parameters

*uplo*      CHARACTER\*1. Specifies whether the upper or lower triangular part of the array *c* is to be referenced as follows:

<i>uplo</i> value	Part of Array <i>c</i> To Be Referenced
U or u	Only the upper triangular part of <i>c</i> is to be referenced.
L or l	Only the lower triangular part of <i>c</i> is to be referenced.

*trans*      CHARACTER\*1. Specifies the operation to be performed as follows:

<i>trans</i> value	Operation to be Performed
N or n	<i>c</i> := <i>alpha</i> * <i>a</i> *CONJG( <i>a</i> ') + <i>beta</i> * <i>c</i>
C or c	<i>c</i> := <i>alpha</i> *CONJG( <i>a</i> ') * <i>a</i> + <i>beta</i> * <i>c</i>

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

*k*      INTEGER. With *trans* = 'N' or 'n', *k* specifies the number of columns of the matrix *a*, and with *trans* = 'C' or 'c', *k* specifies the number of rows of the matrix *a*. The value of *k* must be at least zero.

*alpha*      REAL for *cherk*  
DOUBLE PRECISION for *zherk*

Specifies the scalar *alpha*.

---

<i>a</i>	<b>COMPLEX</b> for <code>cherk</code> <b>DOUBLE COMPLEX</b> for <code>zherk</code> Array, <b>DIMENSION</b> ( <i>lda</i> , <i>ka</i> ), where <i>ka</i> is <i>k</i> when <i>trans</i> = 'N' or 'n', and is <i>n</i> otherwise. Before entry with <i>trans</i> = 'N' or 'n', the leading <i>n</i> by <i>k</i> part of the array <i>a</i> must contain the matrix <i>a</i> , otherwise the leading <i>k</i> by <i>n</i> part of the array <i>a</i> must contain the matrix <i>a</i> .
<i>lda</i>	<b>INTEGER</b> . Specifies the first dimension of <i>a</i> as declared in the calling (sub)program. When <i>trans</i> = 'N' or 'n', then <i>lda</i> must be at least <code>max(1, n)</code> , otherwise <i>lda</i> must be at least <code>max(1, k)</code> .
<i>beta</i>	<b>REAL</b> for <code>cherk</code> <b>DOUBLE PRECISION</b> for <code>zherk</code> Specifies the scalar <i>beta</i> .
<i>c</i>	<b>COMPLEX</b> for <code>cherk</code> <b>DOUBLE COMPLEX</b> for <code>zherk</code> Array, <b>DIMENSION</b> ( <i>ldc</i> , <i>n</i> ). Before entry with <i>uplo</i> = 'U' or 'u', the leading <i>n</i> by <i>n</i> upper triangular part of the array <i>c</i> must contain the upper triangular part of the Hermitian matrix and the strictly lower triangular part of <i>c</i> is not referenced. Before entry with <i>uplo</i> = 'L' or 'l', the leading <i>n</i> by <i>n</i> lower triangular part of the array <i>c</i> must contain the lower triangular part of the Hermitian matrix and the strictly upper triangular part of <i>c</i> is not referenced. The imaginary parts of the diagonal elements need not be set, they are assumed to be zero.
<i>ldc</i>	<b>INTEGER</b> . Specifies the first dimension of <i>c</i> as declared in the calling (sub)program. The value of <i>ldc</i> must be at least <code>max(1, n)</code> .

---

## Output Parameters

*c*

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

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

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

---

## ?her2k

*Performs a rank-2k update of a Hermitian matrix.*

---

```
call cher2k ( uplo, trans, n, k, alpha, a, lda, b, ldb,
              beta, c, ldc )
call zher2k ( uplo, trans, n, k, alpha, a, lda, b, ldb,
              beta, c, ldc )
```

### Discussion

The ?her2k routines perform a rank-2k matrix-matrix operation using Hermitian matrices. The operation is defined as

*c* := *alpha*\**a*\*conjg(*b*') + conjg(*alpha*)\**b*\*conjg(*a*') + *beta*\**c*,  
or

*c* := *alpha*\*conjg(*b*')\**a* + conjg(*alpha*)\*conjg(*a*')\**b* + *beta*\**c*,  
where:

*alpha* is a scalar and *beta* is a real scalar

*c* is an *n* by *n* Hermitian matrix

*a* and *b* are *n* by *k* matrices in the first case and *k* by *n* matrices in the second case.

## Input Parameters

*uplo* CHARACTER\*1. Specifies whether the upper or lower triangular part of the array *c* is to be referenced as follows:

<i>uplo</i> value	Part of Array <i>c</i> To Be Referenced
<i>U</i> or <i>u</i>	Only the upper triangular part of <i>C</i> is to be referenced.
<i>L</i> or <i>l</i>	Only the lower triangular part of <i>C</i> is to be referenced.

*trans* CHARACTER\*1. Specifies the operation to be performed as follows:

<i>trans</i> value	Operation to be Performed
<i>N</i> or <i>n</i>	<i>c</i> := <i>alpha</i> * <i>a</i> *conjg( <i>b</i> ) + <i>alpha</i> * <i>b</i> *conjg( <i>a</i> ) + <i>beta</i> * <i>c</i>
<i>C</i> or <i>c</i>	<i>c</i> := <i>alpha</i> *conjg( <i>a</i> )* <i>b</i> + <i>alpha</i> *conjg( <i>b</i> )* <i>a</i> + <i>beta</i> * <i>c</i>

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

*k* INTEGER. With *trans* = 'N' or 'n', *k* specifies the number of columns of the matrix *a*, and with *trans* = 'C' or 'c', *k* specifies the number of rows of the matrix *a*. The value of *k* must be at least zero.

*alpha* COMPLEX for *cher2k*  
DOUBLE COMPLEX for *zher2k*  
Specifies the scalar *alpha*.

<i>a</i>	COMPLEX for <code>cher2k</code> DOUBLE COMPLEX for <code>zher2k</code>
	Array, DIMENSION ( <i>lda, ka</i> ), where <i>ka</i> is <i>k</i> when <i>trans = 'N'</i> or <i>'n'</i> , and is <i>n</i> otherwise. Before entry with <i>trans = 'N'</i> or <i>'n'</i> , the leading <i>n</i> by <i>k</i> part of the array <i>a</i> must contain the matrix <i>a</i> , otherwise the leading <i>k</i> by <i>n</i> part of the array <i>a</i> must contain the matrix <i>a</i> .
<i>lda</i>	INTEGER. Specifies the first dimension of <i>a</i> as declared in the calling (sub)program. When <i>trans = 'N'</i> or <i>'n'</i> , then <i>lda</i> must be at least <code>max(1, n)</code> , otherwise <i>lda</i> must be at least <code>max(1, k)</code> .
<i>beta</i>	REAL for <code>cher2k</code> DOUBLE PRECISION for <code>zher2k</code>
	Specifies the scalar <i>beta</i> .
<i>b</i>	COMPLEX for <code>cher2k</code> DOUBLE COMPLEX for <code>zher2k</code>
	Array, DIMENSION ( <i>ldb, kb</i> ), where <i>kb</i> is <i>k</i> when <i>trans = 'N'</i> or <i>'n'</i> , and is <i>n</i> otherwise. Before entry with <i>trans = 'N'</i> or <i>'n'</i> , the leading <i>n</i> by <i>k</i> part of the array <i>b</i> must contain the matrix <i>b</i> , otherwise the leading <i>k</i> by <i>n</i> part of the array <i>b</i> must contain the matrix <i>b</i> .
<i>ldb</i>	INTEGER. Specifies the first dimension of <i>b</i> as declared in the calling (sub)program. When <i>trans = 'N'</i> or <i>'n'</i> , then <i>ldb</i> must be at least <code>max(1, n)</code> , otherwise <i>ldb</i> must be at least <code>max(1, k)</code> .
<i>c</i>	COMPLEX for <code>cher2k</code> DOUBLE COMPLEX for <code>zher2k</code>
	Array, DIMENSION ( <i>ldc, n</i> ). Before entry with <i>uplo = 'U'</i> or <i>'u'</i> , the leading <i>n</i> by <i>n</i> upper triangular part of the array <i>c</i> must contain the upper triangular part of the Hermitian matrix and the strictly lower triangular part of <i>c</i> is not referenced.

---

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

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

`ldc`

**INTEGER.** Specifies the first dimension of  $c$  as declared in the calling (sub)program. The value of `ldc` must be at least `max(1, n)`.

## Output Parameters

$c$

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

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

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

---

## ?symm

*Performs a scalar-matrix-matrix product (one matrix operand is symmetric) and adds the result to a scalar-matrix product.*

---

```
call ssymm ( side, uplo, m, n, alpha, a, lda, b, ldb,
            beta, c, ldc )
call dsymm ( side, uplo, m, n, alpha, a, lda, b, ldb,
            beta, c, ldc )
call csymm ( side, uplo, m, n, alpha, a, lda, b, ldb,
            beta, c, ldc )
call zsymm ( side, uplo, m, n, alpha, a, lda, b, ldb,
            beta, c, ldc )
```

### Discussion

The ?symm routines perform a matrix-matrix operation using symmetric matrices. The operation is defined as

$c := \alpha * a * b + \beta * c$ ,

or

$c := \alpha * b * a + \beta * c$ ,

where:

$\alpha$  and  $\beta$  are scalars

$a$  is a symmetric matrix

$b$  and  $c$  are  $m$  by  $n$  matrices.

**Input Parameters**

*side*      CHARACTER\*1. Specifies whether the symmetric matrix *a* appears on the left or right in the operation as follows:

<i>side</i> value	Operation to be Performed
L or l	<i>c</i> := <i>alpha</i> * <i>a</i> * <i>b</i> + <i>beta</i> * <i>c</i>
R or r	<i>c</i> := <i>alpha</i> * <i>b</i> * <i>a</i> + <i>beta</i> * <i>c</i>

*uplo*      CHARACTER\*1. Specifies whether the upper or lower triangular part of the symmetric matrix *a* is to be referenced as follows:

<i>uplo</i> value	Part of Array <i>a</i> To Be Referenced
U or u	Only the upper triangular part of the symmetric matrix is to be referenced.
L or l	Only the lower triangular part of the symmetric matrix is to be referenced.

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

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

*alpha*      REAL for ssymm  
DOUBLE PRECISION for dsymm  
COMPLEX for csymm  
DOUBLE COMPLEX for zsymm

Specifies the scalar *alpha*.

*a*      REAL for ssymm  
DOUBLE PRECISION for dsymm  
COMPLEX for csymm  
DOUBLE COMPLEX for zsymm

Array, DIMENSION (*lda*, *ka*), where *ka* is *m* when *side* = 'L' or 'l' and is *n* otherwise. Before entry with *side* = 'L' or 'l', the *m* by *m* part of the array *a* must contain the symmetric matrix, such that when

*uplo* = 'U' or 'u', the leading *m* by *m* upper triangular part of the array *a* must contain the upper triangular part of the symmetric matrix and the strictly lower triangular part of *a* is not referenced, and when *uplo* = 'L' or 'l', the leading *m* by *m* lower triangular part of the array *a* must contain the lower triangular part of the symmetric matrix and the strictly upper triangular part of *a* is not referenced.

Before entry with *side* = 'R' or 'r', the *n* by *n* part of the array *a* must contain the symmetric matrix, such that when *uplo* = 'U' or 'u', the leading *n* by *n* upper triangular part of the array *a* must contain the upper triangular part of the symmetric matrix and the strictly lower triangular part of *a* is not referenced, and when *uplo* = 'L' or 'l', the leading *n* by *n* lower triangular part of the array *a* must contain the lower triangular part of the symmetric matrix and the strictly upper triangular part of *a* is not referenced.

*lda*

INTEGER. Specifies the first dimension of *a* as declared in the calling (sub)program. When *side* = 'L' or 'l', then *lda* must be at least *max(1, m)*, otherwise *lda* must be at least *max(1, n)*.

*b*

REAL for ssymm  
DOUBLE PRECISION for dsymm  
COMPLEX for csymm  
DOUBLE COMPLEX for zsymm

Array, DIMENSION (*ldb, n*). Before entry, the leading *m* by *n* part of the array *b* must contain the matrix *b*.

*ldb*

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

*beta*                    **REAL** for **ssymm**  
                         **DOUBLE PRECISION** for **dsymm**  
                         **COMPLEX** for **csymm**  
                         **DOUBLE COMPLEX** for **zsymm**

Specifies the scalar *beta*. When *beta* is supplied as zero, then *c* need not be set on input.

*c*                    **REAL** for **ssymm**  
                         **DOUBLE PRECISION** for **dsymm**  
                         **COMPLEX** for **csymm**  
                         **DOUBLE COMPLEX** for **zsymm**

Array, **DIMENSION ( ldc, n )**. Before entry, the leading *m* by *n* part of the array *c* must contain the matrix *c*, except when beta is zero, in which case *c* need not be set on entry.

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

### Output Parameters

*c*                    Overwritten by the *m* by *n* updated matrix.

---

## ?syrk

*Performs a rank-n update of a symmetric matrix.*

---

```
call ssyrk ( uplo, trans, n, k, alpha, a, lda, beta, c,
            ldc )
call dsyrk ( uplo, trans, n, k, alpha, a, lda, beta, c,
            ldc )
call csyrk ( uplo, trans, n, k, alpha, a, lda, beta, c,
            ldc )
call zsyrk ( uplo, trans, n, k, alpha, a, lda, beta, c,
            ldc )
```

### Discussion

The ?syrk routines perform a matrix-matrix operation using symmetric matrices. The operation is defined as

$c := \alpha a^* a + \beta c,$

or

$c := \alpha a^* a^* a + \beta c,$

where:

$\alpha$  and  $\beta$  are scalars

$c$  is an  $n$  by  $n$  symmetric matrix

$a$  is an  $n$  by  $k$  matrix in the first case and a  $k$  by  $n$  matrix in the second case.

## Input Parameters

*uplo* CHARACTER\*1. Specifies whether the upper or lower triangular part of the array *c* is to be referenced as follows:

<i>uplo</i> value	Part of Array <i>c</i> To Be Referenced
<i>U</i> or <i>u</i>	Only the upper triangular part of <i>c</i> is to be referenced.
<i>L</i> or <i>l</i>	Only the lower triangular part of <i>c</i> is to be referenced.

*trans* CHARACTER\*1. Specifies the operation to be performed as follows:

<i>trans</i> value	Operation to be Performed
<i>N</i> or <i>n</i>	<i>c</i> := <i>alpha</i> * <i>a</i> * <i>a</i> ' + <i>beta</i> * <i>c</i>
<i>T</i> or <i>t</i>	<i>c</i> := <i>alpha</i> * <i>a</i> '* <i>a</i> + <i>beta</i> * <i>c</i>
<i>C</i> or <i>c</i>	<i>c</i> := <i>alpha</i> * <i>a</i> '* <i>a</i> + <i>beta</i> * <i>c</i>

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

*k* INTEGER. On entry with *trans* = 'N' or 'n', *k* specifies the number of columns of the matrix *a*, and on entry with *trans* = 'T' or 't' or 'C' or 'c', *k* specifies the number of rows of the matrix *a*. The value of *k* must be at least zero.

*alpha* REAL for ssyrk  
DOUBLE PRECISION for dsyrk  
COMPLEX for csyrk  
DOUBLE COMPLEX for zsydk

Specifies the scalar *alpha*.

<i>a</i>	REAL for <b>ssyrk</b> DOUBLE PRECISION for <b>dsyrk</b> COMPLEX for <b>csyrk</b> DOUBLE COMPLEX for <b>zsyrk</b>  Array, DIMENSION ( <i>lda, ka</i> ), where <i>ka</i> is <i>k</i> when <i>trans</i> = 'N' or 'n', and is <i>n</i> otherwise. Before entry with <i>trans</i> = 'N' or 'n', the leading <i>n</i> by <i>k</i> part of the array <i>a</i> must contain the matrix <i>a</i> , otherwise the leading <i>k</i> by <i>n</i> part of the array <i>a</i> must contain the matrix <i>a</i> .
<i>lda</i>	INTEGER. Specifies the first dimension of <i>a</i> as declared in the calling (sub)program. When <i>trans</i> = 'N' or 'n', then <i>lda</i> must be at least <i>max(1, n)</i> , otherwise <i>lda</i> must be at least <i>max(1, k)</i> .
<i>beta</i>	REAL for <b>ssyrk</b> DOUBLE PRECISION for <b>dsyrk</b> COMPLEX for <b>csyrk</b> DOUBLE COMPLEX for <b>zsyrk</b>  Specifies the scalar <i>beta</i> .
<i>c</i>	REAL for <b>ssyrk</b> DOUBLE PRECISION for <b>dsyrk</b> COMPLEX for <b>csyrk</b> DOUBLE COMPLEX for <b>zsyrk</b>  Array, DIMENSION ( <i>ldc, n</i> ). Before entry with <i>uplo</i> = 'U' or 'u', the leading <i>n</i> by <i>n</i> upper triangular part of the array <i>c</i> must contain the upper triangular part of the symmetric matrix and the strictly lower triangular part of <i>c</i> is not referenced.  Before entry with <i>uplo</i> = 'L' or 'l', the leading <i>n</i> by <i>n</i> lower triangular part of the array <i>c</i> must contain the lower triangular part of the symmetric matrix and the strictly upper triangular part of <i>c</i> is not referenced.
<i>ldc</i>	INTEGER. Specifies the first dimension of <i>c</i> as declared in the calling (sub)program. The value of <i>ldc</i> must be at least <i>max(1, n)</i> .

## Output Parameters

**c**

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

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

---

## ?syr2k

*Performs a rank-2k update of a symmetric matrix.*

---

```
call ssyr2k ( uplo, trans, n, k, alpha, a, lda, b, ldb,
              beta, c, ldc )
call dsyr2k ( uplo, trans, n, k, alpha, a, lda, b, ldb,
              beta, c, ldc )
call csyr2k ( uplo, trans, n, k, alpha, a, lda, b, ldb,
              beta, c, ldc )
call zssyr2k ( uplo, trans, n, k, alpha, a, lda, b, ldb,
              beta, c, ldc )
```

### Discussion

The `?syr2k` routines perform a rank-2k matrix-matrix operation using symmetric matrices. The operation is defined as

`c := alpha*a*b' + alpha*b*a' + beta*c,`

or

`c := alpha*a'*b + alpha*b'*a + beta*c,`

where:

`alpha` and `beta` are scalars

`c` is an `n` by `n` symmetric matrix

$a$  and  $b$  are  $n$  by  $k$  matrices in the first case and  $k$  by  $n$  matrices in the second case.

### Input Parameters

$uplo$  **CHARACTER\*1**. Specifies whether the upper or lower triangular part of the array  $c$  is to be referenced as follows:

$uplo$ value	Part of Array $c$ To Be Referenced
$U$ or $u$	Only the upper triangular part of $c$ is to be referenced.
$L$ or $l$	Only the lower triangular part of $c$ is to be referenced.

$trans$  **CHARACTER\*1**. Specifies the operation to be performed as follows:

$trans$ value	Operation to be Performed
$N$ or $n$	$c := \alpha * a * b' + \alpha * b * a' + \beta * c$
$T$ or $t$	$c := \alpha * a' * a + \alpha * b * a' + \beta * c$

$n$  **INTEGER**. Specifies the order of the matrix  $c$ . The value of  $n$  must be at least zero.

$k$  **INTEGER**. On entry with  $trans = 'N'$  or  $'n'$ ,  $k$  specifies the number of columns of the matrices  $a$  and  $b$ , and on entry with  $trans = 'T'$  or  $'t'$  or  $'C'$  or  $'c'$ ,  $k$  specifies the number of rows of the matrices  $a$  and  $b$ . The value of  $k$  must be at least zero.

$\alpha$  **REAL** for  $ssyr2k$   
**DOUBLE PRECISION** for  $dsyr2k$   
**COMPLEX** for  $csyr2k$   
**DOUBLE COMPLEX** for  $zsyr2k$

Specifies the scalar  $\alpha$ .

---

<i>a</i>	REAL for <b>ssyr2k</b> DOUBLE PRECISION for <b>dsyr2k</b> COMPLEX for <b>csyr2k</b> DOUBLE COMPLEX for <b>zsyr2k</b>
	Array, DIMENSION ( <i>lda, ka</i> ), where <i>ka</i> is <i>k</i> when <i>trans</i> = 'N' or 'n', and is <i>n</i> otherwise. Before entry with <i>trans</i> = 'N' or 'n', the leading <i>n</i> by <i>k</i> part of the array <i>a</i> must contain the matrix <i>a</i> , otherwise the leading <i>k</i> by <i>n</i> part of the array <i>a</i> must contain the matrix <i>a</i> .
<i>lda</i>	INTEGER. Specifies the first dimension of <i>a</i> as declared in the calling (sub)program. When <i>trans</i> = 'N' or 'n', then <i>lda</i> must be at least <b>max(1, n)</b> , otherwise <i>lda</i> must be at least <b>max(1, k)</b> .
<i>b</i>	REAL for <b>ssyr2k</b> DOUBLE PRECISION for <b>dsyr2k</b> COMPLEX for <b>csyr2k</b> DOUBLE COMPLEX for <b>zsyr2k</b>
	Array, DIMENSION ( <i>ldb, kb</i> ) where <i>kb</i> is <i>k</i> when <i>trans</i> = 'N' or 'n' and is 'n' otherwise. Before entry with <i>trans</i> = 'N' or 'n', the leading <i>n</i> by <i>k</i> part of the array <i>b</i> must contain the matrix <i>b</i> , otherwise the leading <i>k</i> by <i>n</i> part of the array <i>b</i> must contain the matrix <i>b</i> .
<i>ldb</i>	INTEGER. Specifies the first dimension of <i>a</i> as declared in the calling (sub)program. When <i>trans</i> = 'N' or 'n', then <i>ldb</i> must be at least <b>max(1, n)</b> , otherwise <i>ldb</i> must be at least <b>max(1, k)</b> .
<i>beta</i>	REAL for <b>ssyr2k</b> DOUBLE PRECISION for <b>dsyr2k</b> COMPLEX for <b>csyr2k</b> DOUBLE COMPLEX for <b>zsyr2k</b>
	Specifies the scalar <i>beta</i> .

<i>c</i>	REAL for <b>ssyr2k</b> DOUBLE PRECISION for <b>dsyr2k</b> COMPLEX for <b>csyr2k</b> DOUBLE COMPLEX for <b>zsyr2k</b>
	Array, DIMENSION ( <i>ldc</i> , <i>n</i> ). Before entry with <i>uplo</i> = 'U' or 'u', the leading <i>n</i> by <i>n</i> upper triangular part of the array <i>c</i> must contain the upper triangular part of the symmetric matrix and the strictly lower triangular part of <i>c</i> is not referenced.  Before entry with <i>uplo</i> = 'L' or 'l', the leading <i>n</i> by <i>n</i> lower triangular part of the array <i>c</i> must contain the lower triangular part of the symmetric matrix and the strictly upper triangular part of <i>c</i> is not referenced.
<i>ldc</i>	INTEGER. Specifies the first dimension of <i>c</i> as declared in the calling (sub)program. The value of <i>ldc</i> must be at least <i>max(1, n)</i> .

## Output Parameters

<i>c</i>	With <i>uplo</i> = 'U' or 'u', the upper triangular part of the array <i>c</i> is overwritten by the upper triangular part of the updated matrix.  With <i>uplo</i> = 'L' or 'l', the lower triangular part of the array <i>c</i> is overwritten by the lower triangular part of the updated matrix.
----------	--

---

## ?trmm

*Computes a scalar-matrix-matrix product (one matrix operand is triangular).*

---

```
call strmm ( side, uplo, transa, diag, m, n, alpha, a,
            lda, b, ldb )
call dtrmm ( side, uplo, transa, diag, m, n, alpha, a,
            lda, b, ldb )
call ctrmm ( side, uplo, transa, diag, m, n, alpha, a,
            lda, b, ldb )
call ztrmm ( side, uplo, transa, diag, m, n, alpha, a,
            lda, b, ldb )
```

### Discussion

The ?trmm routines perform a matrix-matrix operation using triangular matrices. The operation is defined as

$b := \alpha \cdot op(a) \cdot b$

or

$b := \alpha \cdot b \cdot op(a)$

where:

$\alpha$  is a scalar

$b$  is an  $m$  by  $n$  matrix

$a$  is a unit, or non-unit, upper or lower triangular matrix

$op(a)$  is one of  $op(a) = a$  or  $op(a) = a'$  or  $op(a) = conjg(a')$ .

## Input Parameters

*side* CHARACTER\*1. Specifies whether  $\text{op}(a)$  multiplies  $b$  from the left or right in the operation as follows:

<i>side</i> value	Operation To Be Performed
L or l	$b := \alpha * \text{op}(a) * b$
R or r	$b := \alpha * b * \text{op}(a)$

*uplo* CHARACTER\*1. Specifies whether the matrix  $a$  is an upper or lower triangular matrix as follows:

<i>uplo</i> value	Matrix $a$
U or u	Matrix $a$ is an upper triangular matrix.
L or l	Matrix $a$ is a lower triangular matrix.

*transa* CHARACTER\*1. Specifies the form of  $\text{op}(a)$  to be used in the matrix multiplication as follows:

<i>transa</i> value	Form of $\text{op}(a)$
N or n	$\text{op}(a) = a$
T or t	$\text{op}(a) = a'$
C or c	$\text{op}(a) = \text{conjg}(a')$

*diag* CHARACTER\*1. Specifies whether or not  $a$  is unit triangular as follows:

<i>diag</i> value	Matrix $a$
U or u	Matrix $a$ is assumed to be unit triangular.
N or n	Matrix $a$ is not assumed to be unit triangular.

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

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

<i>alpha</i>	<small>REAL for <b>strmm</b> DOUBLE PRECISION for <b>dtrmm</b> COMPLEX for <b>ctrmm</b> DOUBLE COMPLEX for <b>ztrmm</b></small>
	Specifies the scalar <i>alpha</i> . When <i>alpha</i> is zero, then <i>a</i> is not referenced and <i>b</i> need not be set before entry.
<i>a</i>	<small>REAL for <b>strmm</b> DOUBLE PRECISION for <b>dtrmm</b> COMPLEX for <b>ctrmm</b> DOUBLE COMPLEX for <b>ztrmm</b></small>
	Array, <small>DIMENSION (<i>lda</i>,<i>k</i>)</small> , where <i>k</i> is <i>m</i> when <i>side</i> = 'L' or 'l' and is <i>n</i> when <i>side</i> = 'R' or 'r'. Before entry with <i>uplo</i> = 'U' or 'u', the leading <i>k</i> by <i>k</i> upper triangular part of the array <i>a</i> must contain the upper triangular matrix and the strictly lower triangular part of <i>a</i> is not referenced. Before entry with <i>uplo</i> = 'L' or 'l', the leading <i>k</i> by <i>k</i> lower triangular part of the array <i>a</i> must contain the lower triangular matrix and the strictly upper triangular part of <i>a</i> is not referenced. When <i>diag</i> = 'U' or 'u', the diagonal elements of <i>a</i> are not referenced either, but are assumed to be unity.
<i>lda</i>	<small>INTEGER. Specifies the first dimension of <i>a</i> as declared in the calling (sub)program. When <i>side</i> = 'L' or 'l', then <i>lda</i> must be at least <small>max(1, m)</small>, when <i>side</i> = 'R' or 'r', then <i>lda</i> must be at least <small>max(1, n)</small>.</small>
<i>b</i>	<small>REAL for <b>strmm</b> DOUBLE PRECISION for <b>dtrmm</b> COMPLEX for <b>ctrmm</b> DOUBLE COMPLEX for <b>ztrmm</b></small>
	Array, <small>DIMENSION (<i>ldb</i>,<i>n</i>)</small> . Before entry, the leading <i>m</i> by <i>n</i> part of the array <i>b</i> must contain the matrix <i>b</i> .
<i>ldb</i>	<small>INTEGER. Specifies the first dimension of <i>b</i> as declared in the calling (sub)program. The value of <i>ldb</i> must be at least <small>max(1, m)</small>.</small>

## Output Parameters

**b** Overwritten by the transformed matrix.

---

## ?trsm

*Solves a matrix equation (one matrix operand is triangular).*

---

```
call strsm ( side, uplo, transa, diag, m, n, alpha, a,
            lda, b, ldb )
call dtrsm ( side, uplo, transa, diag, m, n, alpha, a,
            lda, b, ldb )
call ctrsm ( side, uplo, transa, diag, m, n, alpha, a,
            lda, b, ldb )
call ztrsm ( side, uplo, transa, diag, m, n, alpha, a,
            lda, b, ldb )
```

### Discussion

The ?trsm routines solve one of the following matrix equations:

$\text{op}(a)*x = \alpha*b$ ,

or

$x*\text{op}(a) = \alpha*b$ ,

where:

$\alpha$  is a scalar

$x$  and  $b$  are  $m$  by  $n$  matrices

$a$  is a unit, or non-unit, upper or lower triangular matrix

$\text{op}(a)$  is one of  $\text{op}(a) = a$  or  $\text{op}(a) = a'$  or  
 $\text{op}(a) = \text{conjg}(a')$ .

The matrix  $x$  is overwritten on  $b$ .

## Input Parameters

*side* CHARACTER\*1. Specifies whether  $\text{op}(a)$  appears on the left or right of  $x$  for the operation to be performed as follows:

<i>side</i> value	Operation To Be Performed
L or l	$\text{op}(a)*x = \alpha*b$
R or r	$x*\text{op}(a) = \alpha*b$

*uplo* CHARACTER\*1. Specifies whether the matrix  $a$  is an upper or lower triangular matrix as follows:

<i>uplo</i> value	Matrix $a$
U or u	Matrix $a$ is an upper triangular matrix.
L or l	Matrix $a$ is a lower triangular matrix.

*transa* CHARACTER\*1. Specifies the form of  $\text{op}(a)$  to be used in the matrix multiplication as follows:

<i>transa</i> value	Form of $\text{op}(a)$
N or n	$\text{op}(a) = a$
T or t	$\text{op}(a) = a'$
C or c	$\text{op}(a) = \text{conj}(a')$

*diag* CHARACTER\*1. Specifies whether or not  $a$  is unit triangular as follows:

<i>diag</i> value	Matrix $a$
U or u	Matrix $a$ is assumed to be unit triangular.
N or n	Matrix $a$ is not assumed to be unit triangular.

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

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

<i>alpha</i>	REAL for <b>strsm</b> DOUBLE PRECISION for <b>dtrsm</b> COMPLEX for <b>ctrsm</b> DOUBLE COMPLEX for <b>ztrsm</b>
	Specifies the scalar <i>alpha</i> . When <i>alpha</i> is zero, then <i>a</i> is not referenced and <i>b</i> need not be set before entry.
<i>a</i>	REAL for <b>strsm</b> DOUBLE PRECISION for <b>dtrsm</b> COMPLEX for <b>ctrsm</b> DOUBLE COMPLEX for <b>ztrsm</b>
	Array, DIMENSION ( <i>lda</i> , <i>k</i> ), where <i>k</i> is <i>m</i> when <i>side</i> = 'L' or 'l' and is <i>n</i> when <i>side</i> = 'R' or 'r'. Before entry with <i>uplo</i> = 'U' or 'u', the leading <i>k</i> by <i>k</i> upper triangular part of the array <i>a</i> must contain the upper triangular matrix and the strictly lower triangular part of <i>a</i> is not referenced.
	Before entry with <i>uplo</i> = 'L' or 'l', the leading <i>k</i> by <i>k</i> lower triangular part of the array <i>a</i> must contain the lower triangular matrix and the strictly upper triangular part of <i>a</i> is not referenced. When <i>diag</i> = 'U' or 'u', the diagonal elements of <i>a</i> are not referenced either, but are assumed to be unity.
<i>lda</i>	INTEGER. Specifies the first dimension of <i>a</i> as declared in the calling (sub)program. When <i>side</i> = 'L' or 'l', then <i>lda</i> must be at least $\max(1, m)$ , when <i>side</i> = 'R' or 'r', then <i>lda</i> must be at least $\max(1, n)$ .
<i>b</i>	REAL for <b>strsm</b> DOUBLE PRECISION for <b>dtrsm</b> COMPLEX for <b>ctrsm</b> DOUBLE COMPLEX for <b>ztrsm</b>
	Array, DIMENSION ( <i>ldb</i> , <i>n</i> ). Before entry, the leading <i>m</i> by <i>n</i> part of the array <i>b</i> must contain the right-hand side matrix <i>b</i> .

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

### Output Parameters

*b*      Overwritten by the solution matrix *x*.

## Sparse BLAS Routines and Functions

This section describes Sparse BLAS, an extension of BLAS Level 1 included in Intel® Math Kernel Library beginning with MKL release 2.1. Sparse BLAS is a group of routines and functions that perform a number of common vector operations on sparse vectors stored in compressed form.

*Sparse vectors* are those in which the majority of elements are zeros. Sparse BLAS routines and functions are specially implemented to take advantage of vector sparsity. This allows you to achieve large savings in computer time and memory. If  $nz$  is the number of non-zero vector elements, the computer time taken by Sparse BLAS operations will be  $O(nz)$ .

### Vector Arguments in Sparse BLAS

**Compressed sparse vectors.** Let  $a$  be a vector stored in an array, and assume that the only non-zero elements of  $a$  are the following:

$$a(k_1), a(k_2), a(k_3) \dots a(k_{nz}),$$

where  $nz$  is the total number of non-zero elements in  $a$ .

In Sparse BLAS, this vector can be represented in compressed form by two FORTRAN arrays,  $x$  (values) and  $indx$  (indices). Each array has  $nz$  elements:

$$x(1)=a(k_1), x(2)=a(k_2), \dots x(nz)=a(k_{nz}),$$

$$indx(1)=k_1, indx(2)=k_2, \dots indx(nz)=k_{nz}.$$

Thus, a sparse vector is fully determined by the triple ( $nz$ ,  $x$ ,  $indx$ ). If you pass a negative or zero value of  $nz$  to Sparse BLAS, the subroutines do not modify any arrays or variables.

**Full-storage vectors.** Sparse BLAS routines can also use a vector argument fully stored in a single FORTRAN array (a full-storage vector). If  $y$  is a full-storage vector, its elements must be stored contiguously: the first element in  $y(1)$ , the second in  $y(2)$ , and so on. This corresponds to an increment  $incy = 1$  in BLAS Level 1. No increment value for full-storage vectors is passed as an argument to Sparse BLAS routines or functions.

## Naming Conventions in Sparse BLAS

Similar to BLAS, the names of Sparse BLAS subprograms have prefixes that determine the data type involved: `s` and `d` for single- and double-precision real; `c` and `z` for single- and double-precision complex.

If a Sparse BLAS routine is an extension of a “dense” one, the subprogram name is formed by appending the suffix `i` (standing for *indexed*) to the name of the corresponding “dense” subprogram. For example, the Sparse BLAS routine `saxpyi` corresponds to the BLAS routine `saxpy`, and the Sparse BLAS function `cdotci` corresponds to the BLAS function `cdotc`.

## Routines and Data Types in Sparse BLAS

Routines and data types supported in the MKL implementation of Sparse BLAS are listed in Table 2-4.

**Table 2-4 Sparse BLAS Routines and Their Data Types**

Routine/ Function	Data Types	Description
<code>?axpyi</code>	s, d, c, z	Scalar-vector product plus vector (routines)
<code>?doti</code>	s, d	Dot product (functions)
<code>?dotci</code>	c, z	Complex dot product conjugated (functions)
<code>?dotui</code>	c, z	Complex dot product unconjugated (functions)
<code>?gthr</code>	s, d, c, z	Gathering a full-storage sparse vector into compressed form: <code>nz</code> , <code>x</code> , <code>indx</code> (routines)
<code>?gthrz</code>	s, d, c, z	Gathering a full-storage sparse vector into compressed form and assigning zeros to gathered elements in the full-storage vector (routines)
<code>?roti</code>	s, d	Givens rotation (routines)
<code>?sctr</code>	s, d, c, z	Scattering a vector from compressed form to full-storage form (routines)

## BLAS Routines That Can Work With Sparse Vectors

The following BLAS Level 1 routines will give correct results when you pass to them a compressed-form array  $x$  (with the increment  $incx = 1$ ):

<code>?asum</code>	sum of absolute values of vector elements
<code>?copy</code>	copying a vector
<code>?nrm2</code>	Euclidean norm of a vector
<code>?scal</code>	scaling a vector
<code>i?amax</code>	index of the element with the largest absolute value or, for complex flavors, the largest sum $ Re_x(i)  +  Im_x(i) $ .
<code>i?amin</code>	index of the element with the smallest absolute value or, for complex flavors, the smallest sum $ Re_x(i)  +  Im_x(i) $ .

The result  $i$  returned by `i?amax` and `i?amin` should be interpreted as index in the compressed-form array, so that the largest (smallest) value is  $x(i)$ ; the corresponding index in full-storage array is  $indx(i)$ .

You can also call `?rotg` to compute the parameters of Givens rotation and then pass these parameters to the Sparse BLAS routines `?roti`.

---

## ?axpyi

*Adds a scalar multiple of compressed sparse vector to a full-storage vector.*

---

```
call saxpyi ( nz, a, x, indx, y )
call daxpyi ( nz, a, x, indx, y )
call caxpyi ( nz, a, x, indx, y )
call zaxpyi ( nz, a, x, indx, y )
```

### Discussion

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

$$y := a*x + y$$

where:

$a$  is a scalar

(*nz*, *x*, *indx*) is a sparse vector stored in compressed form

*y* is a vector in full storage form.

The ?axpyi routines reference or modify only the elements of *y* whose indices are listed in the array *indx*. The values in *indx* must be distinct.

### Input Parameters

*nz* INTEGER. The number of elements in *x* and *indx*.

*a* REAL for *saxpyi*  
DOUBLE PRECISION for *daxpyi*  
COMPLEX for *caxpyi*  
DOUBLE COMPLEX for *zaxpyi*

Specifies the scalar *a*.

*x* REAL for *saxpyi*  
DOUBLE PRECISION for *daxpyi*  
COMPLEX for *caxpyi*  
DOUBLE COMPLEX for *zaxpyi*  
Array, DIMENSION at least *nz*.

*indx* INTEGER. Specifies the indices for the elements of *x*.  
Array, DIMENSION at least *nz*.

*y* REAL for *saxpyi*  
DOUBLE PRECISION for *daxpyi*  
COMPLEX for *caxpyi*  
DOUBLE COMPLEX for *zaxpyi*  
Array, DIMENSION at least  $\max_i(\text{indx}(i))$ .

### Output Parameters

*y* Contains the updated vector *y*.

## ?doti

*Computes the dot product of a compressed sparse real vector by a full-storage real vector.*

---

```
res = sdoti ( nz, x, indx, y )
res = ddoti ( nz, x, indx, y )
```

### Discussion

The ?doti functions return the dot product of *x* and *y* defined as

$$x(1)*y(indx(1)) + x(2)*y(indx(2)) + \dots + x(nz)*y(indx(nz))$$

where the triple (*nz*, *x*, *indx*) defines a sparse real vector stored in compressed form, and *y* is a real vector in full storage form. The functions reference only the elements of *y* whose indices are listed in the array *indx*. The values in *indx* must be distinct.

### Input Parameters

<i>nz</i>	INTEGER. The number of elements in <i>x</i> and <i>indx</i> .
<i>x</i>	REAL for sdoti DOUBLE PRECISION for ddoti Array, DIMENSION at least <i>nz</i> .
<i>indx</i>	INTEGER. Specifies the indices for the elements of <i>x</i> . Array, DIMENSION at least <i>nz</i> .
<i>y</i>	REAL for sdoti DOUBLE PRECISION for ddoti Array, DIMENSION at least $\max_i(indx(i))$ .

### Output Parameters

<i>res</i>	REAL for sdoti DOUBLE PRECISION for ddoti Contains the dot product of <i>x</i> and <i>y</i> , if <i>nz</i> is positive. Otherwise, <i>res</i> contains 0.
------------	--

## ?dotci

Computes the conjugated dot product of  
a compressed sparse complex vector  
with a full-storage complex vector.

```
res = cdotci ( nz, x, indx, y )
res = zdotci ( nz, x, indx, y )
```

### Discussion

The ?dotci functions return the dot product of *x* and *y* defined as

$$\text{conjg}(x(1)) * y(indx(1)) + \dots + \text{conjg}(x(nz)) * y(indx(nz))$$

where the triple (*nz*, *x*, *indx*) defines a sparse complex vector stored in compressed form, and *y* is a real vector in full storage form. The functions reference only the elements of *y* whose indices are listed in the array *indx*. The values in *indx* must be distinct.

### Input Parameters

<i>nz</i>	INTEGER. The number of elements in <i>x</i> and <i>indx</i> .
<i>x</i>	COMPLEX for cdotci DOUBLE COMPLEX for zdotci Array, DIMENSION at least <i>nz</i> .
<i>indx</i>	INTEGER. Specifies the indices for the elements of <i>x</i> . Array, DIMENSION at least <i>nz</i> .
<i>y</i>	COMPLEX for cdotci DOUBLE COMPLEX for zdotci Array, DIMENSION at least max <sub>i</sub> ( <i>indx</i> ( <i>i</i> )).

### Output Parameters

<i>res</i>	COMPLEX for cdotci DOUBLE COMPLEX for zdotci
	Contains the conjugated dot product of <i>x</i> and <i>y</i> , if <i>nz</i> is positive. Otherwise, <i>res</i> contains 0.

## ?dotui

*Computes the dot product of a compressed sparse complex vector by a full-storage complex vector.*

---

```
res = cdotui ( nz, x, indx, y )
res = zdotui ( nz, x, indx, y )
```

### Discussion

The ?dotui functions return the dot product of *x* and *y* defined as

*x*(1)\**y*(*indx*(1)) + *x*(2)\**y*(*indx*(2)) + ... + *x*(*nz*)\**y*(*indx*(*nz*))

where the triple (*nz*, *x*, *indx*) defines a sparse complex vector stored in compressed form, and *y* is a real vector in full storage form. The functions reference only the elements of *y* whose indices are listed in the array *indx*. The values in *indx* must be distinct.

### Input Parameters

<i>nz</i>	INTEGER. The number of elements in <i>x</i> and <i>indx</i> .
<i>x</i>	COMPLEX for cdotui DOUBLE COMPLEX for zdotui Array, DIMENSION at least <i>nz</i> .
<i>indx</i>	INTEGER. Specifies the indices for the elements of <i>x</i> . Array, DIMENSION at least <i>nz</i> .
<i>y</i>	COMPLEX for cdotui DOUBLE COMPLEX for zdotui Array, DIMENSION at least max <sub><i>i</i></sub> ( <i>indx</i> ( <i>i</i> )).

### Output Parameters

<i>res</i>	COMPLEX for cdotui DOUBLE COMPLEX for zdotui Contains the dot product of <i>x</i> and <i>y</i> , if <i>nz</i> is positive. Otherwise, <i>res</i> contains 0.
------------	---

---

## ?gthr

Gathers a full-storage sparse vector's elements into compressed form.

---

```
call sgthr ( nz, y, x, indx )
call dgthr ( nz, y, x, indx )
call cgthr ( nz, y, x, indx )
call zgthr ( nz, y, x, indx )
```

### Discussion

The ?gthr routines gather the specified elements of a full-storage sparse vector *y* into compressed form (*nz*, *x*, *indx*). The routines reference only the elements of *y* whose indices are listed in the array *indx*:

$$x(i) = y(indx(i)), \text{ for } i=1, 2, \dots, nz.$$

### Input Parameters

*nz*                    INTEGER. The number of elements of *y* to be gathered.

*indx*                INTEGER. Specifies indices of elements to be gathered.  
Array, DIMENSION at least *nz*.

*y*                    REAL for sgthr  
                        DOUBLE PRECISION for dgthr  
                        COMPLEX for cgthr  
                        DOUBLE COMPLEX for zgthr  
                        Array, DIMENSION at least max<sub>i</sub>(indx(i)).

### Output Parameters

*x*                    REAL for sgthr  
                        DOUBLE PRECISION for dgthr  
                        COMPLEX for cgthr  
                        DOUBLE COMPLEX for zgthr  
                        Array, DIMENSION at least *nz*.

Contains the vector converted to the compressed form.

---

## ?gthrz

*Gathers a sparse vector's elements into compressed form, replacing them by zeros.*

---

```
call sgthrz ( nz, y, x, indx )
call dgthrz ( nz, y, x, indx )
call cgthrz ( nz, y, x, indx )
call zgthrz ( nz, y, x, indx )
```

### Discussion

The ?gthrz routines gather the elements with indices specified by the array *indx* from a full-storage vector *y* into compressed form (*nz*, *x*, *indx*) and overwrite the gathered elements of *y* by zeros. Other elements of *y* are not referenced or modified (see also ?gthr).

### Input Parameters

<i>nz</i>	INTEGER. The number of elements of <i>y</i> to be gathered.
<i>indx</i>	INTEGER. Specifies indices of elements to be gathered. Array, DIMENSION at least <i>nz</i> .
<i>y</i>	REAL for sgthrz DOUBLE PRECISION for dgthrz COMPLEX for cgthrz DOUBLE COMPLEX for zgthrz Array, DIMENSION at least max <sub><i>i</i></sub> ( <i>indx</i> ( <i>i</i> )).

### Output Parameters

<i>x</i>	REAL for sgthrz DOUBLE PRECISION for dgthrz COMPLEX for cgthrz DOUBLE COMPLEX for zgthrz Array, DIMENSION at least <i>nz</i> . Contains the vector converted to the compressed form.
<i>y</i>	The updated vector <i>y</i> .

---

## ?roti

Applies Givens rotation to sparse vectors  
one of which is in compressed form.

---

```
call sroti ( nz, x, indx, y, c, s )
call droti ( nz, x, indx, y, c, s )
```

### Discussion

The ?roti routines apply the Givens rotation to elements of two real vectors, *x* (in compressed form *nz*, *x*, *indx*) and *y* (in full storage form):

$$\begin{aligned}x(i) &= c*x(i) + s*y(indx(i)) \\y(indx(i)) &= c*y(indx(i)) - s*x(i)\end{aligned}$$

The routines reference only the elements of *y* whose indices are listed in the array *indx*. The values in *indx* must be distinct.

### Input Parameters

<i>nz</i>	INTEGER. The number of elements in <i>x</i> and <i>indx</i> .
<i>x</i>	REAL for sroti DOUBLE PRECISION for droti Array, DIMENSION at least <i>nz</i> .
<i>indx</i>	INTEGER. Specifies the indices for the elements of <i>x</i> . Array, DIMENSION at least <i>nz</i> .
<i>y</i>	REAL for sroti DOUBLE PRECISION for droti Array, DIMENSION at least max <sub>i</sub> ( <i>indx</i> ( <i>i</i> )).
<i>c</i>	A scalar: REAL for sroti DOUBLE PRECISION for droti.
<i>s</i>	A scalar: REAL for sroti DOUBLE PRECISION for droti.

### Output Parameters

<i>x</i> and <i>y</i>	The updated arrays.
-----------------------	---------------------

---

## ?sctr

*Converts compressed sparse vectors into full storage form.*

---

```
call ssctr ( nz, x, indx, y )
call dsctr ( nz, x, indx, y )
call csctr ( nz, x, indx, y )
call zsctr ( nz, x, indx, y )
```

### Discussion

The `?sctr` routines scatter the elements of the compressed sparse vector (`nz, x, indx`) to a full-storage vector `y`. The routines modify only the elements of `y` whose indices are listed in the array `indx`:

$y(indx(i)) = x(i)$ , for  $i=1, 2, \dots, nz$ .

### Input Parameters

<code>nz</code>	<code>INTEGER</code> . The number of elements of <code>x</code> to be scattered.
<code>indx</code>	<code>INTEGER</code> . Specifies indices of elements to be scattered. Array, <code>DIMENSION</code> at least <code>nz</code> .
<code>x</code>	<code>REAL</code> for <code>ssctr</code> <code>DOUBLE PRECISION</code> for <code>dsctr</code> <code>COMPLEX</code> for <code>csctr</code> <code>DOUBLE COMPLEX</code> for <code>zsctr</code> Array, <code>DIMENSION</code> at least <code>nz</code> . Contains the vector to be converted to full-storage form.

### Output Parameters

<code>y</code>	<code>REAL</code> for <code>ssctr</code> <code>DOUBLE PRECISION</code> for <code>dsctr</code> <code>COMPLEX</code> for <code>csctr</code> <code>DOUBLE COMPLEX</code> for <code>zsctr</code> Array, <code>DIMENSION</code> at least $\max_i(indx(i))$ . Contains the vector <code>y</code> with updated elements.
----------------	--

# *Fast Fourier Transforms*

3

This chapter describes the fast Fourier transform (FFT) routines. The FFT routines included consist of two classes: one-dimensional and two-dimensional. Both one-dimensional and two-dimensional routines have been optimized to effectively use cache memory.

The chapter contains these major sections:

- One-dimensional FFTs
- Two-dimensional FFTs

Each of the major sections contains the description of three groups of the FFTs.

## One-dimensional FFTs

The one-dimensional FFTs include the following groups:

- Complex-to-Complex Transforms
- Real-to-Complex Transforms
- Complex-to-Real Transforms.

All one-dimensional FFTs are in-place. The transform length must be a power of 2. The complex-to-complex transform routines perform both forward and inverse transforms of a complex vector. The real-to-complex transform routines perform forward transforms of a real vector. The complex-to-real transform routines perform inverse transforms of a complex conjugate-symmetric vector, which is packed in a real array.

## Data Storage Types

Each FFT group contains two sets of FFTs having the similar functionality: one set is used for the Fortran-interface and the other for the C-interface. The former set stores the complex data as a Fortran complex data type, while the latter stores the complex data as float arrays of real and imaginary parts separately. These sets are distinguished by naming the FFTs within each set. The names of the FFTs used for the C-interface have the letter “c” added to the end of the FFTs’ Fortran names. For example, the names of the `cfft1d/zfft1d` FFTs for the corresponding C-interface routines are `cfft1dc/zfft1dc`. All names of the C-type data items are lower case.

[Table 3-1](#) lists the one-dimensional FFT routine groups and the data types associated with them.

**Table 3-1 One-dimensional FFTs: Names and Data Types**

Group	Stored as Fortran Complex Data	Stored as C Real Data	Data Types	Description
Complex-to-Complex	<code>cfft1d/zfft1d</code>	<code>cfft1dc/zfft1dc</code>	c, z	Transform complex data to complex data.
Real-to-Complex	<code>scfft1d/dzfft1d</code>	<code>scfft1dc/dzfft1dc</code>	sc, dz	Transform forward real-to-complex data. Complement <code>csfft1d/zdfft1d</code> and <code>csfft1dc/zdfft1dc</code> FFTs.
Complex-to-Real	<code>csfft1d/dzfft1d</code>	<code>csfft1dc/dzfft1dc</code>	cs, zd	Transform inverse complex-to-real data. Complement <code>scfft1d/dzfft1d</code> and <code>scfft1dc/dzfft1dc</code> FFTs.

## Data Structure Requirements

For C-interface, storage of the complex-to-complex transform routines data requires separate float arrays for the real and imaginary parts. The real-to-complex and complex-to-real pairs require a single float input/output array.

The C-interface requires scalar values to be passed by value.

All transforms require additional memory to store the transform coefficients. When performing multiple FFTs of the same dimension, the

table of coefficients should be created only once and then used on all the FFTs afterwards. Using the same table rather than creating it repeatedly for each FFT produces an obvious performance gain.

## Complex-to-Complex One-dimensional FFTs

Each of the complex-to-complex routines computes a forward or inverse FFT of a complex vector.

The forward FFT is computed according to the mathematical equation

$$z_j = \sum_{k=0}^{n-1} r_k * w^{-j*k}, \quad 0 \leq j \leq n-1$$

The inverse FFT is computed according to the mathematical equation

$$r_j = \frac{1}{n} \sum_{k=0}^{n-1} z_k * w^{j*k}, \quad 0 \leq j \leq n-1$$

where  $w = \exp\left[\frac{2\pi i}{n}\right]$ ,  $i$  being the imaginary unit.

The operation performed by the complex-to-complex routines is determined by the value of the *isign* parameter used by each of these routines.

If *isign* = -1, perform the forward FFT where input and output are in normal order.

If *isign* = +1, perform the inverse FFT where input and output are in normal order.

If *isign* = -2, perform the forward FFT where input is in normal order and output is in bit-reversed order.

If *isign* = +2, perform the inverse FFT where input is in bit-reversed order and output is in normal order.

If *isign* = 0, initialize FFT coefficients for both the forward and inverse FFTs.

The above equations apply to all FFTs with all data types indicated in [Table 3-1](#).

To compute a forward or inverse FFT of a given length, first initialize the coefficients by calling the function with *isign* = 0. Thereafter, any number of transforms of the same length can be computed by calling the function with *isign* = +1, -1, +2, -2.

---

## cfft1d/zfft1d

*Fortran-interface routines. Compute the forward or inverse FFT of a complex vector (in-place)*

---

```
call cfft1d ( r, n, isign, wsave )
call zfft1d ( r, n, isign, wsave )
```

### Discussion

The operation performed by the `cfft1d/zfft1d` routines is determined by the value of `isign`. See the equations of the operations for the [“Complex-to-Complex One-dimensional FFTs”](#) above.

### Input Parameters

<code>r</code>	<code>COMPLEX</code> for <code>cfft1d</code> <code>DOUBLE COMPLEX</code> for <code>zfft1d</code>
<code>n</code>	Array, <code>DIMENSION</code> at least ( <code>n</code> ). Contains the complex vector on which the transform is to be performed. Not referenced if <code>isign = 0</code> .
<code>isign</code>	<code>INTEGER</code> . Transform length; <code>n</code> must be a power of 2. <code>INTEGER</code> . Flag indicating the type of operation to be performed: if <code>isign = 0</code> , initialize the coefficients <code>wave</code> ; if <code>isign = -1</code> , perform the forward FFT where input and output are in normal order; if <code>isign = +1</code> , perform the inverse FFT where input and output are in normal order; if <code>isign = -2</code> , perform the forward FFT where input is in normal order and output is in bit-reversed order; if <code>isign = +2</code> , perform the inverse FFT where input is in bit-reversed order and output is in normal order.
<code>wave</code>	<code>COMPLEX</code> for <code>cfft1d</code> <code>DOUBLE COMPLEX</code> for <code>zfft1d</code>

Array, `DIMENSION` at least  $((3*n)/2)$ . If `isign = 0`,

then `wsave` is an output parameter. Otherwise, `wsave` contains the FFT coefficients initialized on a previous call with `isign = 0`.

### Output Parameters

<code>r</code>	Contains the complex result of the transform depending on <code>isign</code> . Does not change if <code>isign = 0</code> .
<code>wsave</code>	If <code>isign = 0</code> , <code>wsave</code> contains the initialized FFT coefficients. Otherwise, <code>wsave</code> does not change.

---

## cfft1dc/zfft1dc

*C-interface routines. Compute the forward or inverse FFT of a complex vector (in-place).*

---

```
void cfft1dc (float* r, float* i, int n, int isign, float* wsave)
void zfft1dc (double* r, double* i, int n, int isign, double* wsave)
```

### Discussion

The operation performed by the `cfft1dc/zfft1dc` routines is determined by the value of `isign`. See the equations of the operations for the [“Complex-to-Complex One-dimensional FFTs”](#).

### Input Parameters

<code>r</code>	<code>float*</code> for <code>cfft1dc</code> <code>double*</code> for <code>zfft1dc</code> Pointer to an array of size at least ( <code>n</code> ). Contains the real parts of complex vector to be transformed. Not referenced if <code>isign = 0</code> .
<code>i</code>	<code>float*</code> for <code>cfft1dc</code> <code>double*</code> for <code>zfft1dc</code> Pointer to an array of size at least ( <code>n</code> ). Contains the imaginary parts of complex vector to be transformed.

	Not referenced if <i>isign</i> = 0.
<i>n</i>	<i>int</i> . Transform length; <i>n</i> must be a power of 2.
<i>isign</i>	<i>int</i> . Flag indicating the type of operation to be performed: if <i>isign</i> = 0, initialize the coefficients <i>wsave</i> ; if <i>isign</i> = -1, perform the forward FFT where input and output are in normal order; if <i>isign</i> = +1, perform the inverse FFT where input and output are in normal order; if <i>isign</i> = -2, perform the forward FFT where input is in normal order and output is in bit-reversed order; if <i>isign</i> = +2, perform the inverse FFT where input is in bit-reversed order and output is in normal order.
<i>wsave</i>	<i>float*</i> for <b>cfft1dc</b> <i>double*</i> for <b>zfft1dc</b> Pointer to an array of size at least $(3*n)$ . If <i>isign</i> = 0, then <i>wsave</i> is an output parameter. Otherwise, <i>wsave</i> contains the FFT coefficients initialized on a previous call with <i>isign</i> = 0.

### Output Parameters

<i>r</i>	Contains the real part of the transform depending on <i>isign</i> . Does not change if <i>isign</i> = 0.
<i>i</i>	Contains the imaginary part of the transform depending on <i>isign</i> . Does not change if <i>isign</i> = 0.
<i>wsave</i>	If <i>isign</i> = 0, <i>wsave</i> contains the initialized FFT coefficients. Otherwise, <i>wsave</i> does not change.

### Real-to-Complex One-dimensional FFTs

Each of the real-to-complex routines computes forward FFT of a real input vector according to the mathematical equation

$$z_j = \sum_{k=0}^{n-1} t_k * w^{-j*k}, \quad 0 \leq j \leq n-1$$

for  $t_k = \text{cmplx}(r_k, 0)$ , where  $r_k$  is the real input vector,  $0 \leq k \leq n - 1$ .

The mathematical result  $z_j$ ,  $0 \leq j \leq n - 1$ , is the complex conjugate-symmetric vector, where  $z(n/2+i) = \text{conjg}(z(n/2-i))$ ,  $1 \leq i \leq n/2 - 1$ , and moreover  $z(0)$  and  $z(n/2)$  are real values.

This complex conjugate-symmetric (CCS) vector can be stored in the complex array of size  $(n/2+1)$  or in the real array of size  $(n+2)$ . The data storage of the CCS format is defined later for Fortran-interface and C-interface routines separately.

[Table 3-2](#) shows a comparison of the effects of performing the `cfft1d/zfft1d` complex-to-complex FFT on a vector of length  $n=8$  in which all the imaginary elements are zeros, with the real-to-complex `scfft1d/zdffft1d` FFT applied to the same vector. The advantage of the latter approach is that only half of the input data storage is required and there is no need to zero the imaginary part. The last two columns are stored in the real array of size  $(n+2)$  containing the complex conjugate-symmetric vector in CCS format.

To compute a forward FFT of a given length, first initialize the coefficients by calling the routine you are going to use with `isign = 0`. Thereafter, any number of real-to-complex and complex-to-real transforms of the same length can be computed by calling that routine with the `isign` value other than `0`.

**Table 3-2 Comparison of the Storage Effects of Complex-to-Complex and Real-to-Complex FFTs**

Input Vectors		Output Vectors				
<code>cfft1d</code>		<code>scfft1d</code>	<code>cfft1d</code>		<code>scfft1d</code>	
Complex Data		Real Data	Complex Data		Real Data	
Real	Imaginary		Real	Imaginary	(Real)	(Imaginary)
0.841471	0.000000	0.841471	1.543091	0.000000	1.543091	0.000000
0.909297	0.000000	0.909297	3.875664	0.910042	3.875664	0.910042
0.141120	0.000000	0.141120	-0.915560	-0.397326	-0.915560	-0.397326
-0.756802	0.000000	-0.756802	-0.274874	-0.121691	-0.274874	-0.121691
-0.958924	0.000000	-0.958924	-0.181784	0.000000	-0.181784	0.000000
-0.279415	0.000000	-0.279415	-0.274874	0.121691		
0.656987	0.000000	0.656987	-0.915560	0.397326		
0.989358	0.000000	0.989358	3.875664	-0.910042		

---

## scfft1d/dzfft1d

*Fortran-interface routines. Compute forward FFT of a real vector and represent the complex conjugate-symmetric result in CCS format (in-place).*

---

```
call scfft1d ( r, n, isign, wsave )
call dzfft1d ( r, n, isign, wsave )
```

### Discussion

The operation performed by the `scfft1d/dzfft1d` routines is determined by the value of `isign`. See the equations of the operations for “[Real-to-Complex One-dimensional FFTs](#)” above. These routines are complementary to the complex-to-real transform routines [`csfft1d/zdfft1d`](#).

### Input Parameters

<code>r</code>	<code>REAL</code> for <code>scfft1d</code> <code>DOUBLE PRECISION</code> for <code>dzfft1d</code>
<code>n</code>	Array, <code>DIMENSION</code> at least <code>(n+2)</code> . First <code>n</code> elements contain the input vector to be transformed. The elements <code>r(n+1)</code> and <code>r(n+2)</code> are used on output. The array <code>r</code> is not referenced if <code>isign = 0</code> .
<code>isign</code>	<code>INTEGER</code> . Transform length; <code>n</code> must be a power of 2. <code>INTEGER</code> . Flag indicating the type of operation to be performed: if <code>isign</code> is 0, initialize the coefficients <code>wsave</code> ; if <code>isign</code> is not 0, perform the forward FFT.
<code>wsave</code>	<code>REAL</code> for <code>scfft1d</code> <code>DOUBLE PRECISION</code> for <code>dzfft1d</code>

Array, `DIMENSION` at least `(2*n+4)`. If `isign = 0`, then `wave` contains output data. Otherwise, `wave` contains coefficients required to perform the FFT that has been initialized on a previous call to this routine or the complementary complex-to-real FFT routine.

### Output Parameters

`r` If `isign = 0`, `r` does not change. If `isign` is not `0`, the output real-valued array `r(1:n+2)` contains the complex conjugate-symmetric vector `z(1:n)` packed in CCS format for Fortran interface.

The table below shows the relationship between them.

<code>r(1)</code>	<code>r(2)</code>	<code>r(3)</code>	<code>r(4)</code>	...	<code>r(n-1)</code>	<code>r(n)</code>	<code>r(n+1)</code>	<code>r(n+2)</code>
<code>z(1)</code>	0	<code>REz(2)</code>	<code>IMz(2)</code>	...	<code>REz(n/2)</code>	<code>IMz(n/2)</code>	<code>z(n/2+1)</code>	0

The full complex vector `z(1:n)` is defined by

`z(i) = cmplx(r(2*i-1), r(2*i)),`  
 $1 \leq i \leq n/2+1,$

`z(n/2+i) = conjg(z(n/2+2-i)),`  
 $2 \leq i \leq n/2.$

Then, `z(1:n)` is the forward FFT of a real input vector `r(1:n)`.

`wave` If `isign = 0`, `wave` contains the coefficients required by the called routine. Otherwise `wave` does not change.

---

## scfft1dc/dzfft1dc

*C-interface routines. Compute forward FFT of a real vector and represent the complex conjugate-symmetric result in CCS format (in-place).*

---

```
void scfft1dc ( float* r, int n, int isign, float* wsave );
void dzfft1dc ( double* r, int n, int isign, double* wsave );
```

### Discussion

The operation performed by the **scfft1dc/dzfft1dc** routines is determined by the value of *isign*. See the equations of the operations for the “[Real-to-Complex One-dimensional FFTs](#)” above.

These routines are complementary to the complex-to-real transform routines [csfft1dc/zdfft1dc](#).

### Input Parameters

<i>r</i>	<i>float*</i> for <b>scfft1dc</b> <i>double*</i> for <b>dzfft1dc</b>
	Pointer to an array of size at least <i>(n+2)</i> . First <i>n</i> elements contain the input vector to be transformed. The array <i>r</i> is not referenced if <i>isign = 0</i> .
<i>n</i>	<i>int</i> . Transform length; <i>n</i> must be a power of 2.
<i>isign</i>	<i>int</i> . Flag indicating the type of operation to be performed: if <i>isign</i> is 0, initialize the coefficients <i>wsave</i> ; if <i>isign</i> is not 0, perform the forward FFT.
<i>wsave</i>	<i>float*</i> for <b>scfft1dc</b> <i>double*</i> for <b>dzfft1dc</b>

Pointer to an array of size at least  $(2*n+4)$ .  
 If  $isign = 0$ , then  $wave$  contains output data.  
 Otherwise,  $wave$  contains coefficients required to perform the FFT that has been initialized on a previous call to this routine or the complementary complex-to-real FFT routine.

### Output Parameters

$r$  If  $isign = 0$ ,  $r$  does not change. If  $isign$  is not  $0$ , the output real-valued array  $r(0:n+1)$  contains the complex conjugate-symmetric vector  $z(0:n-1)$  packed in CCS format for C-interface.  
 The table below shows the relationship between them.

$r(0)$	$r(1)$	$r(2)$	...	$r(n/2)$	$r(n/2+1)$	$r(n/2+2)$	...	$r(n)$	$r(n+1)$
$z(0)$	$\text{RE}z(1)$	$\text{RE}z(2)$	...	$z(n/2)$	0	$\text{IM}z(1)$	...	$\text{IM}z(n/2-1)$	0

The full complex vector  $z(0:n-1)$  is defined by

$$z(i) = \text{cmplx}(r(i), r(n/2+1+i)), \quad 0 \leq i \leq n/2,$$

$$z(n/2+i) = \text{conjg}(z(n/2-i)), \quad 1 \leq i \leq n/2-1.$$

Then,  $z(0:n-1)$  is the forward FFT of the real input vector of length  $n$ .

$wave$  If  $isign = 0$ ,  $wave$  contains the coefficients required by the called routine. Otherwise  $wave$  does not change.

### Complex-to-Real One-dimensional FFTs

Each of the complex-to-real routines computes a one-dimensional inverse FFT according to the mathematical equation

$$t_j = \frac{1}{n} \sum_{k=0}^{n-1} z_k * w^{j*k}, \quad 0 \leq j \leq n-1$$

The mathematical input is the complex conjugate-symmetric vector  $z_j$ ,  $0 \leq j \leq n-1$ , where  $z(n/2+i) = \text{conjg}(z(n/2-i))$ ,  $1 \leq i \leq n/2-1$ , and moreover  $z(0)$  and  $z(n/2)$  are real values.

The mathematical result is  $t_j = \text{cplx}(r_j, 0)$ , where  $r_j$  is a real vector,  $0 \leq j \leq n - 1$ .

Input to the complex-to-real transform routines is a real array of size  $(n+2)$ , which contains the complex conjugate-symmetric vector  $z(0:n-1)$  in CCS format (see [“Real-to-Complex One-dimensional FFTs”](#) above).

Output of the complex-to-real routines is a real vector of size  $n$ .

[Table 3-3](#) is identical to [Table 3-2](#), except for reversing the input and output vectors. In the complex-to-real routines the last two columns are stored in the input real array of size  $(n+2)$  containing the complex conjugate-symmetric vector in CCS format.

To compute an inverse FFT of a given length, first initialize the coefficients by calling the routine you are going to use with  $\text{isign} = 0$ . Thereafter, any number of real-to-complex and complex-to-real transforms of the same length can be computed by calling the appropriate routine with the  $\text{isign}$  value other than  $0$ .

**Table 3-3 Comparison of the Storage Effects of Complex-to-Real and Complex-to-Complex FFTs**

Output Vectors				Input Vectors			
cfft1d		csfft1d	cfft1d		csfft1d		
Complex Data		Real Data	Complex Data		Real Data		
Real	Imaginary		Real	Imaginary	(Real)	(Imaginary)	
0.841471	0.000000	0.841471	1.543091	0.000000	1.543091	0.000000	
0.909297	0.000000	0.909297	3.875664	0.910042	3.875664	0.910042	
0.141120	0.000000	0.141120	-0.915560	-0.397326	-0.915560	-0.397326	
-0.756802	0.000000	-0.756802	-0.274874	-0.121691	-0.274874	-0.121691	
-0.958924	0.000000	-0.958924	-0.181784	0.000000	-0.181784	0.000000	
-0.279415	0.000000	-0.279415	-0.274874	0.121691			
0.656987	0.000000	0.656987	-0.915560	0.397326			
0.989358	0.000000	0.989358	3.875664	-0.910042			

## csfft1d/zdfft1d

*Fortran-interface routines.*

*Compute inverse FFT of a complex conjugate-symmetric vector packed in CCS format (in-place).*

---

```
call csfft1d ( r, n, isign, wsave )
call zdfft1d ( r, n, isign, wsave )
```

### Discussion

The operation performed by the `csfft1d/zdfft1d` routines is determined by the value of `isign`. See the equations of the operations for the “[Complex-to-Real One-dimensional FFTs](#)” above.

These routines are complementary to the real-to-complex transform routines [`scfft1d/dzfft1d`](#).

### Input Parameters

`r`                    REAL for `csfft1d`  
DOUBLE PRECISION for `zdfft1d`  
Array, DIMENSION at least (`n+2`).  
Not referenced if `isign = 0`.  
If `isign` is not 0, then `r(1:n+2)` contains the complex conjugate-symmetric vector packed in CCS format for Fortran-interface.  
The table below shows the relationship between them

---

<code>r(1)</code>	<code>r(2)</code>	<code>r(3)</code>	<code>r(4)</code>	...	<code>r(n-1)</code>	<code>r(n)</code>	<code>r(n+1)</code>	<code>r(n+2)</code>
<code>z(1)</code>	0	<code>REz(2)</code>	<code>IMz(2)</code>	...	<code>REz(n/2)</code>	<code>IMz(n/2)</code>	<code>z(n/2+1)</code>	0

The full complex vector  $z(1:n)$  is defined by

$$\begin{aligned} z(i) &= \text{cmplx}(r(2*i-1), r(2*i)), \\ 1 \leq i &\leq n/2+1, \\ z(n/2+i) &= \text{conjg}(z(n/2+2-i)), \\ 2 \leq i &\leq n/2. \end{aligned}$$

After the transform,  $r(1:n)$  contains the inverse FFT of the complex conjugate-symmetric vector  $z(1:n)$ .

***n*** **INTEGER.** Transform length; *n* must be a power of 2.

***isign*** **INTEGER.** Flag indicating the type of operation to be performed:

if *isign* is 0, initialize the coefficients *wsave*;  
if *isign* is not 0, perform the inverse FFT.

***wsave*** **REAL** for **csfft1d**

**DOUBLE PRECISION** for **zdffft1d**

Array, **DIMENSION** at least  $(2*n+4)$ . If *isign* = 0, then *wsave* contains output data. Otherwise, *wsave* contains coefficients required to perform the FFT that has been initialized on a previous call to this routine or the complementary real-to-complex FFT routine.

## Output Parameters

***r*** If *isign* is not 0, then *r(1:n)* is the real result of the inverse FFT of the complex conjugate-symmetric vector  $z(1:n)$ . Does not change if *isign* = 0.

***wsave*** If *isign* = 0, *wsave* contains the coefficients required by the called routine. Otherwise *wsave* does not change.

## csfft1dc/zdfft1dc

*C-interface routines. Compute inverse FFT of a complex conjugate-symmetric vector packed in CCS format (in-place).*

---

```
void csfft1dc ( float* r, int n, int isign, float* wsave )
void zdfft1dc ( double* r, int n, int isign, double* wsave )
```

### Discussion

The operation performed by the `csfft1dc/zdfft1dc` routines is determined by the value of `isign`. See the equations of the operations for the [“Complex-to-Real One-dimensional FFTs”](#) above.

These routines are complementary to the real-to-complex transform routines [scfft1dc/dzfft1dc](#).

### Input Parameters

`r`                    `float*` for `csfft1dc`  
                       `double*` for `zdfft1dc`

Pointer to an array of size at least `(n+2)`. Not referenced if `isign = 0`.

If `isign` is not 0, then `r(0:n+1)` contains the complex conjugate-symmetric vector packed in CCS format for C-interface.

The table below shows the relationship between them

.

<code>r(0)</code>	<code>r(1)</code>	<code>r(2)</code>	<code>...</code>	<code>r(n/2)</code>	<code>r(n/2+1)</code>	<code>r(n/2+2)</code>	<code>...</code>	<code>r(n)</code>	<code>r(n+1)</code>
<code>z(0)</code>	<code>REz(1)</code>	<code>REz(2)</code>	<code>...</code>	<code>z(n/2)</code>	0	<code>IMz(1)</code>	<code>...</code>	<code>IMz(n/2-1)</code>	0

The full complex vector  $z(0:n-1)$  is defined by  
 $z(i) = \text{cmplx}(r(i), r(n/2+1+i)), 0 \leq i \leq n/2,$

$z(n/2+i) = \text{conjg}(z(n/2-i)), 1 \leq i \leq n/2-1.$

After the transform,  $r(0:n-1)$  is the inverse FFT of the complex conjugate-symmetric vector  $z(0:n-1)$ .

*n* *int*. Transform length; *n* must be a power of 2.

*isign* *int*. Flag indicating the type of operation to be performed:

if *isign* = 0, initialize the coefficients *wave*;  
 if *isign* is not 0, perform the inverse FFT.

*wave* *float\** for *csfft1dc*  
*double\** for *zdfft1dc*  
 Pointer to an array of size at least  $(2*n+4)$ .  
 If *isign* = 0, then *wave* contains output data.  
 Otherwise, *wave* contains coefficients required to perform the FFT that has been initialized on a previous call to this routine or the complementary real-to-complex FFT routine.

## Output Parameters

*r* If *isign* is not 0, then *r(0:n-1)* is the real result of the inverse FFT of the complex conjugate-symmetric vector  $z(0:n-1)$ . Does not change if *isign* = 0.

*wave* If *isign* = 0, *wave* contains the coefficients required by the called routine. Otherwise *wave* does not change.

## Two-dimensional FFTs

The two-dimensional FFTs are functionally the same as one-dimensional FFTs. They contain the following groups:

- Complex-to-Complex Transforms
- Real-to-Complex Transforms
- Complex-to-Real Transforms.

All two-dimensional FFTs are in-place. Transform lengths must be a power of 2. The complex-to-complex transform routines perform both forward and inverse transforms of a complex matrix. The real-to-complex transform routines perform forward transforms of a real matrix. The complex-to-real transform routines perform inverse transforms of a complex conjugate-symmetric matrix, which is packed in a real array.

The naming conventions are also the same as those for one-dimensional FFTs, with “2d” replacing “1d” in all cases. [Table 3-4](#) lists the two-dimensional FFT routine groups and the data types associated with them.

**Table 3-4    Two-dimensional FFTs: Names and Data Types**

Group	Stored as FORTRAN		Data Types	Description
	Complex Data	Stored as C Real Data		
Complex- to- Complex	<a href="#"><u>cfft2d/</u></a> <a href="#"><u>zfft2d</u></a>	<a href="#"><u>cfft2dc/</u></a> <a href="#"><u>zfft2dc</u></a>	c, z	Transform complex data to complex data.
Real-to- Complex	<a href="#"><u>scfft2d/</u></a> <a href="#"><u>dzfft2d</u></a>	<a href="#"><u>scfft2dc/</u></a> <a href="#"><u>dzfft2dc</u></a>	sc, dz	Transform forward real-to-complex data. Complement <a href="#"><u>csfft2d/zdffft2d</u></a> and <a href="#"><u>csfft2dc/zdffft2dc</u></a> FFTs.
Complex- to-Real	<a href="#"><u>csfft2d/</u></a> <a href="#"><u>zdffft2d</u></a>	<a href="#"><u>csfft2dc/</u></a> <a href="#"><u>zdffft2dc</u></a>	cs, zd	Transform inverse complex-to-real data. Complement <a href="#"><u>scfft2d/dzfft2d</u></a> and <a href="#"><u>scfft2dc/dzfft2dc</u></a> FFTs.

The C-interface requires scalar values to be passed by value. The major difference between the one-dimensional and two-dimensional FFTs is that your application does not need to provide storage for transform coefficients.

The data storage types and data structure requirements are the same as for one-dimensional FFTs. For more information, see the [“Data Storage Types”](#) and [“Data Structure Requirements”](#) sections at the beginning of this chapter.

## Complex-to-Complex Two-dimensional FFTs

Each of the complex-to-complex routines computes a forward or inverse FFT of a complex matrix in-place.

The forward FFT is computed according to the mathematical equation

$$z_{i,j} = \sum_{k=0}^{m-1} \sum_{l=0}^{n-1} r_{k,l} * w_m^{-i*k} * w_n^{-j*l}, \quad 0 \leq i \leq m-1, \quad 0 \leq j \leq n-1$$

The inverse FFT is computed according to the mathematical equation

$$r_{i,j} = \frac{1}{m*n} \sum_{k=0}^{m-1} \sum_{l=0}^{n-1} z_{k,l} * w_m^{i*k} * w_n^{j*l}, \quad 0 \leq i \leq m-1, \quad 0 \leq j \leq n-1$$

where  $w_m = \exp\left[\frac{2\pi i}{m}\right]$ ,  $w_n = \exp\left[\frac{2\pi i}{n}\right]$ ,  $i$  being the imaginary unit.

The operation performed by the complex-to-complex routines is determined by the value of the *isign* parameter.

If *isign* = -1, perform the forward FFT where input and output are in normal order.

If *isign* = +1, perform the inverse FFT where input and output are in normal order.

If *isign* = -2, perform the forward FFT where input is in normal order and output is in bit-reversed order.

If *isign* = +2, perform the inverse FFT where input is in bit-reversed order and output is in normal order.

The above equations apply to all FFTs with all data types indicated in [Table 3-4](#).

## cfft2d/zfft2d

*Fortran-interface routines. Compute the forward or inverse FFT of a complex matrix (in-place).*

---

```
call cfft2d ( r, m, n, isign )
call zfft2d ( r, m, n, isign )
```

### Discussion

The operation performed by the **cfft2d/zfft2d** routines is determined by the value of *isign*. See the equations of the operations for [“Complex-to-Complex Two-dimensional FFTs”](#).

### Input Parameters

<i>r</i>	<small>COMPLEX for <b>cfft2d</b> DOUBLE COMPLEX for <b>zfft2d</b></small>
	Array, <small>DIMENSION at least (m, n), with its leading dimension equal to <i>m</i>. This array contains the complex matrix to be transformed.</small>
<i>m</i>	<small>INTEGER. Column transform length (number of rows); <i>m</i> must be a power of 2.</small>
<i>n</i>	<small>INTEGER. Row transform length (number of columns); <i>n</i> must be a power of 2.</small>
<i>isign</i>	<small>INTEGER. Flag indicating the type of operation to be performed: if <i>isign</i> = -1, perform the forward FFT where input and output are in normal order; if <i>isign</i> = +1, perform the inverse FFT where input and output are in normal order; if <i>isign</i> = -2, perform the forward FFT where input is in normal order and output is in bit-reversed order; if <i>isign</i> = +2, perform the inverse FFT where input is in bit-reversed order and output is in normal order.</small>

## Output Parameters

*r* Contains the complex result of the transform depending on *isign*.

---

## cfft2dc/zfft2dc

*C-interface routines. Compute the forward or inverse FFT of a complex matrix (in-place).*

---

```
void cfft2dc ( float* r, float* i, int m, int n, int isign )
void zfft2dc ( double* r, double* i, int m, int n, int isign )
```

### Discussion

The operation performed by the `cfft2dc/zfft2dc` routines is determined by the value of *isign*. See the equations of the operations for the “[Complex-to-Complex Two-dimensional FFTs](#)” above.

## Input Parameters

*r* `float*` for `cfft2dc`  
`double*` for `zfft2dc`

Pointer to a two-dimensional array of size at least  $(m, n)$ , with its leading dimension equal to *n*. The array contains the real parts of a complex matrix to be transformed.

*i* `float*` for `cfft2dc`  
`double*` for `zfft2dc`

Pointer to a two-dimensional array of size at least  $(m, n)$ , with its leading dimension equal to *n*. The array contains the imaginary parts of a complex matrix to be transformed.

*m* `int`. Column transform length (number of rows); *m* must be a power of 2.

*n* **int**. Row transform length (number of columns); *n* must be a power of 2.

*isign* **int**. Flag indicating the type of operation to be performed:

- if *isign* = -1, perform the forward FFT where input and output are in normal order;
- if *isign* = +1, perform the inverse FFT where input and output are in normal order;
- if *isign* = -2, perform the forward FFT where input is in normal order and output is in bit-reversed order;
- if *isign* = +2, perform the inverse FFT where input is in bit-reversed order and output is in normal order.

### Output Parameters

*r* Contains the real parts of the complex result depending on *isign*.

*i* Contains the imaginary parts of the complex depending on *isign*.

## Real-to-Complex Two-dimensional FFTs

Each of the real-to-complex routines computes the forward FFT of a real matrix according to the mathematical equation

$$z_{i,j} = \sum_{k=0}^{m-1} \sum_{l=0}^{n-1} t_{k,l} * w_m^{-i*k} * w_n^{-j*l}, \quad 0 \leq i \leq m-1, \quad 0 \leq j \leq n-1$$

$t_{k,l} = \text{cmplx}(r_{k,l}, 0)$ , where  $r_{k,l}$  is a real input matrix,  $0 \leq k \leq m-1$ ,  $0 \leq l \leq n-1$ .

The mathematical result  $z_{i,j}$ ,  $0 \leq i \leq m-1$ ,  $0 \leq j \leq n-1$ , is the complex matrix of size  $(m, n)$ .

Each column is the complex conjugate-symmetric vector as follows:

for  $0 \leq j \leq n-1$ ,

$z(m/2+i, j) = \text{conjg}(z(m/2-i, j))$ ,  $1 \leq i \leq m/2-1$ .

Moreover,  $z(0, j)$  and  $z(m/2, j)$  are real values for  $j=0$  and  $j=n/2$ .

This mathematical result can be stored in the complex two-dimensional array of size  $(m/2+1, n/2+1)$  or in the real two-dimensional array of size  $(m+2, n+2)$ . The data storage of CCS format is defined later for Fortran-interface and C-interface routines separately.

---

## scfft2d/dzfft2d

*Fortran-interface routines. Compute forward FFT of a real matrix and represent the complex conjugate-symmetric result in CCS format (in-place).*

---

```
call scfft2d ( r, m, n )
call dzfft2d ( r, m, n )
```

### Discussion

See the equations of the operations for the [“Real-to-Complex Two-dimensional FFTs”](#) above.

These routines are complementary to the complex-to-real transform routines [csfft2d/zdffft2d](#).

### Input Parameters

<i>r</i>	<small>REAL for scfft2d DOUBLE PRECISION for dzfft2d</small>
	Array, <small>DIMENSION at least (m+2, n+2)</small> , with its leading dimension equal to (m+2). The first <i>m</i> rows and <i>n</i> columns of this array contain the real matrix to be transformed. <a href="#">Table 3-5</a> presents the input data layout.
<i>m</i>	<small>INTEGER. Column transform length (number of rows); <i>m</i> must be a power of 2.</small>
<i>n</i>	<small>INTEGER. Row transform length (number of columns); <i>n</i> must be a power of 2.</small>

**Table 3-5 Fortran-interface Real Data Storage for the Real-to-Complex and Complex-to-Real Two-dimensional FFTs**

$r(1, 1)$	$r(1, 2)$	...	$r(1, n-1)$	$r(1, n)$	n/u	n/u
$r(2, 1)$	$r(2, 2)$	...	$r(2, n-1)$	$r(2, n)$	n/u	n/u
$r(3, 1)$	$r(3, 2)$	...	$r(3, n-1)$	$r(3, n)$	n/u	n/u
$r(4, 1)$	$r(4, 2)$	...	$r(4, n-1)$	$r(4, n)$	n/u	n/u
...	...	...	...	...	...	...
$r(m-1, 1)$	$r(m-1, 2)$	...	$r(m-1, n-1)$	$r(m-1, n)$	n/u	n/u
$r(m, 1)$	$r(m, 2)$	...	$r(m, n-1)$	$r(m, n)$	n/u	n/u
n/u	n/u	...	n/u	n/u	n/u	n/u
n/u	n/u	...	n/u	n/u	n/u	n/u

\* n/u - not used

### Output Parameters

$r$  The output real array  $r(1:m+2, 1:n+2)$  contains the complex conjugate-symmetric matrix  $z(1:m, 1:n)$  packed in CCS format for Fortran-interface as follows:

- Rows 1 and  $m+1$  contain in  $n+2$  locations the complex conjugate-symmetric vectors  $z(1, j)$  and  $z(m/2+1, j)$  packed in CCS format (see [“Real-to-Complex One-dimensional FFTs”](#) above).
 

The full complex vector  $z(1, j)$  is defined by:

$$z(1, j) = \text{cmplx}(r(1, 2*j-1), r(1, 2*j)), \quad 1 \leq j \leq n/2+1,$$

$$z(1, n/2+1+j) = \text{conjg}(z(1, n/2+1-j)), \quad 1 \leq j \leq n/2-1.$$

The full complex vector  $z(m/2+1, j)$  is defined by:

$$z(m/2+1, j) = \text{cmplx}(r(m+1, 2*j-1), r(m+1, 2*j)),$$

$$1 \leq j \leq n/2+1,$$

$$z(m/2+1, n/2+1+j) = \text{conjg}(z(m/2+1, n/2+1-j)),$$

$$1 \leq j \leq n/2-1;$$
- Rows from 3 to  $m$  contain in  $n$  locations complex vectors represented as
 
$$z(i+1, j) = \text{cmplx}(r(2*i+1, j), r(2*i+2, j)),$$

$$1 \leq i \leq m/2-1, \quad 1 \leq j \leq n.$$

- The rest matrix elements can be obtained from

$$z(m/2+1+i, j) = \text{conjg}(z(m/2+1-i, j)),$$

$$1 \leq i \leq m/2-1, 1 \leq j \leq n.$$

The storage of the complex conjugate-symmetric matrix  $z$  for Fortran-interface is shown in [Table 3-6](#).

**Table 3-6      Fortran-interface Data Storage of CCS Format for the Real-to-Complex and Complex-to-Real Two-Dimensional FFTs**

$z(1,1)$	0	$\text{RE}z(1,2)$	$\text{IM}z(1,2)$	...	$\text{RE}z(1,n/2)$	$\text{IM}z(1,n/2)$	$z(1, n/2+1)$	0
0	0	0	0	...	0	0	0	0
$\text{RE}z(2,1)$	$\text{RE}z(2,2)$	$\text{RE}z(2,3)$	$\text{RE}z(2,4)$	...	$\text{RE}z(2,n-1)$	$\text{RE}z(2,n)$	n/u	n/u
$\text{IM}z(2,1)$	$\text{IM}z(2,2)$	$\text{IM}z(2,3)$	$\text{IM}z(2,4)$	...	$\text{IM}z(2,n-1)$	$\text{IM}z(2,n)$	n/u	n/u
...	...	...	...	...	...	...	n/u	n/u
$\text{RE}z(m/2,1)$	$\text{RE}z(m/2,2)$	$\text{RE}z(m/2,3)$	$\text{RE}z(m/2,4)$	...	$\text{RE}z(m/2, n-1)$	$\text{RE}z(m/2, n)$	n/u	n/u
$\text{IM}z(m/2,1)$	$\text{IM}z(m/2,2)$	$\text{IM}z(m/2,3)$	$\text{IM}z(m/2,4)$	...	$\text{IM}z(m/2, n-1)$	$\text{IM}z(m/2, n)$	n/u	n/u
$z(m/2+1,1)$	0	$\text{RE}z(m/2+1,2)$	$\text{IM}z(m/2+1,2)$	...	$\text{RE}z(m/2+1, n/2)$	$\text{IM}z(m/2+1, n/2)$	$z(m/2+1, n/2+1)$	0
0	0	0	0	...	0	0	n/u	n/u

\* n/u - not used

## scfft2dc/dzfft2dc

*C-interface routine. Compute forward FFT of a real matrix and represent the complex conjugate-symmetric result in CCS format (in-place).*

```
void scfft2dc ( float* r, int m, int n )
void dzfft2dc ( double* r, int m, int n )
```

## Discussion

See the equations of the operations for the [“Real-to-Complex Two-dimensional FFTs”](#) above.

These routines are complementary to the complex-to-real transform routines [`csfft2dc/zdfft2dc`](#).

## Input Parameters

`r`                    `float*` for `scfft2dc`  
                       `double*` for `dzfft2dc`

Pointer to an array of size at least  $(m+2, n+2)$ , with its leading dimension equal to  $(n+2)$ . The first  $m$  rows and  $n$  columns of this array contain the real matrix to be transformed.

[Table 3-7](#) presents the input data layout.

`m`                    `int`. Column transform length;  
                       `m` must be a power of 2.

`n`                    `int`. Row transform length;  
                       `n` must be a power of 2.

**Table 3-7 C-interface Real Data Storage for a Real-to-Complex and Complex-to-Real Two-dimensional FFTs**

<code>r(0, 0)</code>	<code>r(0, 1)</code>	...	<code>r(0, n-2)</code>	<code>r(0, n-1)</code>	n/u	n/u
<code>r(1, 0)</code>	<code>r(1, 1)</code>	...	<code>r(1, n-2)</code>	<code>r(1, n-1)</code>	n/u	n/u
<code>r(2, 0)</code>	<code>r(2, 1)</code>	...	<code>r(2, n-2)</code>	<code>r(2, n-1)</code>	n/u	n/u
<code>r(3, 0)</code>	<code>r(3, 1)</code>	...	<code>r(3, n-2)</code>	<code>r(3, n-1)</code>	n/u	n/u
...	...	...	...	...	...	...
<code>r(m-2, 0)</code>	<code>r(m-2, 1)</code>	...	<code>r(m-2, n-2)</code>	<code>r(m-2, n-1)</code>	n/u	n/u
<code>r(m-1, 0)</code>	<code>r(m-1, 1)</code>	...	<code>r(m-1, n-2)</code>	<code>r(m-1, n-1)</code>	n/u	n/u
n/u	n/u	...	n/u	n/u	n/u	n/u
n/u	n/u	...	n/u	n/u	n/u	n/u

## Output Parameters

*r* The output real array  $r(0:m+1, 0:n+1)$  contains the complex conjugate-symmetric matrix  $z(0:m-1, 0:n-1)$  packed in CCS format for C-interface as follows:

- Columns 0 and  $n/2$  contain in  $m+2$  locations the complex conjugate-symmetric vectors  $z(i, 0)$  and  $z(i, n/2)$  in CCS format (see “[Real-to-Complex One-dimensional FFTs](#)” above).

The full complex vector  $z(i, 0)$  is defined by:

$$z(i, 0) = \text{cmplx}(r(i, 0), r(m/2+i+1, 0)), \quad 0 \leq i \leq m/2,$$

$$z(m/2+i, 0) = \text{conjg}(z(m/2-i, 0)), \quad 1 \leq i \leq m/2-1.$$

The full complex vector  $z(i, n/2)$  is defined by:

$$z(i, n/2) = \text{cmplx}(r(i, n/2), r(m/2+i+1, n/2)), \quad 0 \leq i \leq m/2,$$

$$z(m/2+i, n/2) = \text{conjg}(z(m/2-i, n/2)), \quad 1 \leq i \leq m/2-1.$$

- Columns from 1 to  $n/2-1$  contain real parts, and columns from  $n/2+2$  to  $n$  contain imaginary parts of complex vectors. These values for each vector are stored in  $m$  locations represented as follows

$$z(i, j) = \text{cmplx}(r(i, j), r(i, n/2+1+j)),$$

$$0 \leq i \leq m-1, \quad 1 \leq j \leq n/2-1.$$

- The rest matrix elements can be obtained from

$$z(i, n/2+j) = \text{conjg}(z(i, n/2-j)),$$

$$0 \leq i \leq m-1, \quad 1 \leq j \leq n/2-1.$$

The storage of the complex conjugate-symmetric matrix  $z$  for C-interface is shown in [Table 3-8](#).

**Table 3-8 C-interface Data Storage of CCS Format for the Real-to-Complex and Complex-to-Real Two-dimensional FFT**

z(0,0)	REz(0,1)	...	REz(0, n/2-1)	z(0,n/2)	0	IMz(0,1)	...	IMz(0, n/2-1)	0
REz(1,0)	REz(1,1)	...	REz(1, n/2-1)	REz(1,n/2)	0	IMz(1,1)	...	IMz(1, n/2-1)	0
...	...	...	...	...	0	...	...	...	0
REz(m/2-1, 0)	REz(m/2-1, 1)	...	REz(m/2-1, n/2-1)	REz(m/2-1, n/2)	0	IMz(m/2-1, 1)	...	IMz(m/2-1, n/2-1)	0
z(m/2,0)	REz(m/2,1)	...	REz(m/2, n/2-1)	z(m/2,n/2)	0	IMz(m/2,1)	...	IMz(m/2, n/2-1)	0
0	REz(m/2+1, 1)	...	REz(m/2+1, n/2-1)	0	0	IMz(m/2+1, 1)	...	IMz(m/2+1, n/2-1)	0
IMz(1,0)	REz(m/2+2, 1)	...	REz(m/2+2, n/2-1)	IMz(1,n/2)	0	IMz(m/2+2, 1)	...	IMz(m/2+2, n/2-1)	0
...	...	...	...	...	0	...	...	...	0
IMz(m/2-2, 0)	REz(m-1,1)	...	REz(m-1, n/2-1)	IMz(m/2-2, n/2)	0	IMz(m-1,1)	...	IMz(m-1, n/2-1)	0
IMz(m/2-1, 0)	n/u	...	n/u	IMz(m/2-1, n/2)	n/u	n/u	...	n/u	n/u
0	n/u	...	n/u	0	n/u	n/u	...	n/u	n/u

### Complex-to-Real Two-dimensional FFTs

Each of the complex-to-real routines computes a two-dimensional inverse FFT according to the mathematical equation:

$$t_{i,j} = \frac{1}{m*n} \sum_{k=0}^{m-1} \sum_{l=0}^{n-1} z_k, l^* w_m^{i*k} * w_n^{j*l}, \quad 0 \leq i \leq m-1, \quad 0 \leq j \leq n-1$$

The mathematical input  $z_{i,j}$ ,  $0 \leq i \leq m-1$ ,  $0 \leq j \leq n-1$ , is a complex matrix of size  $(m, n)$ . Each column is the complex conjugate-symmetric vector as follows:

for  $0 \leq j \leq n-1$ ,  
 $z(m/2+i, j) = \text{conjg}(z(m/2-i, j))$ ,  $1 \leq i \leq m/2-1$ .  
 Moreover,  $z(0, j)$  and  $z(m/2, j)$  are real values for  $j=0$  and  $j=n/2$ .

This mathematical input can be stored in the complex two-dimensional array of size  $(m/2+1, n/2+1)$  or in the real two-dimensional array of size  $(m+2, n+2)$ . For the details of data storage of CCS format see [“Real-to-Complex One-dimensional FFTs”](#) above.

The mathematical result of the transform is  $t_{k, l} = \text{cmplx}(r_{k, l}, 0)$ , where  $r_{k, l}$  is the real matrix,  $0 \leq k \leq m-1$ ,  $0 \leq l \leq n-1$ .

---

## csfft2d/zdfft2d

*Fortran-interface routine.*

*Compute inverse FFT of a complex conjugate-symmetric matrix packed in CCS format (in-place).*

---

```
call csfft2d ( r, m, n )
call zdfft2d ( r, m, n )
```

### Discussion

See the equations of the operations for the [“Complex-to-Real Two-dimensional FFTs”](#) above. These routines are complementary to the real-to-complex transform routines [scfft2d/dzfft2d](#).

### Input Parameters

*r*                   SINGLE PRECISION REAL\*4 for **csfft2d**  
                   DOUBLE PRECISION REAL\*8 for **zdfft2d**

Array, **DIMENSION** at least  $(m+2, n+2)$ , with its leading dimension equal to  $(m+2)$ . This array contains the complex conjugate-symmetric matrix in CCS format to be transformed. The input data layout is given in [Table 3-6](#).

*m*                   **INTEGER.** Column transform length (number of rows); *m* must be a power of 2.

*n*                   **INTEGER.** Row transform length (number of columns); *n* must be a power of 2.

### Output Parameters

*r*                   Contains the real result returned by the transform. For the output data layout, see [Table 3-5](#).

---

## csfft2dc/zdfft2dc

*C-interface routines.*

*Compute inverse FFT of a complex conjugate-symmetric matrix packed in CCS format (in-place).*

---

```
void csfft2dc ( float* r, int m, int n );
void zdfft2dc ( double* r, int m, int n );
```

### Discussion

See the equations of the operations for the “[Complex-to-Real Two-dimensional FFTs](#)” above. These routines are complementary to the real-to-complex transform routines [scfft2dc/dzfft2dc](#).

### Input Parameters

*r*                   **float\*** for `csfft2dc`  
**double\*** for `zdfft2dc`

Pointer to an array of size at least  $(m+2, n+2)$ , with its leading dimension equal to  $(n+2)$ . This array contains the complex conjugate-symmetric matrix in CCS format to be transformed. The input data layout is given in [Table 3-8](#).

*m*                   **int.** Column transform length; *m* must be a power of 2.

*n* *int*. Row transform length; *n* must be a power of 2.

### Output Parameters

*r* Contains the real result returned by the transform. The output data layout is the same as that for the input data of *scfft2dc/dzfft2dc*. See [Table 3-7](#) for the details.

# LAPACK Routines: Linear Equations

4

This chapter describes the Math Kernel Library implementation of routines from the LAPACK package that are used for solving systems of linear equations and performing a number of related computational tasks. The library includes LAPACK routines for both real and complex data.

Routines are supported for systems of equations with the following types of matrices:

- general
- banded
- symmetric or Hermitian positive-definite (both full and packed storage)
- symmetric or Hermitian positive-definite banded
- symmetric or Hermitian indefinite (both full and packed storage)
- symmetric or Hermitian indefinite banded
- triangular (both full and packed storage)
- triangular banded
- tridiagonal.

For each of the above matrix types, the library includes routines for performing the following computations: *factoring* the matrix (except for triangular matrices); *equilibrating* the matrix; *solving* a system of linear equations; *estimating the condition number* of a matrix; *refining* the solution of linear equations and computing its error bounds; *inverting* the matrix.

To solve a particular problem, you can either call two or more [computational routines](#) or call a corresponding [driver routine](#) that combines several tasks in one call, such as `?gesv` for factoring and solving. Thus, to solve a system of linear equations with a general matrix, you can first call `?getrf` (*LU factorization*) and then `?getrs` (computing the solution).

Then, you might wish to call `?gerfs` to refine the solution and get the error bounds. Alternatively, you can just use the driver routine `?gesvx` which performs all these tasks in one call.

## Routine Naming Conventions

For each routine introduced in this chapter, you can use the LAPACK name.

**LAPACK names** are listed in Tables 4-1 and 4-2, and have the structure **xyyzzz** or **xyyzz**, which is described below.

The initial letter **x** indicates the data type:

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

The second and third letters **yy** indicate the matrix type and storage scheme:

<b>ge</b>	general
<b>gb</b>	general band
<b>gt</b>	general tridiagonal
<b>po</b>	symmetric or Hermitian positive-definite
<b>pp</b>	symmetric or Hermitian positive-definite (packed storage)
<b>pb</b>	symmetric or Hermitian positive-definite band
<b>pt</b>	symmetric or Hermitian positive-definite tridiagonal
<b>sy</b>	symmetric indefinite
<b>sp</b>	symmetric indefinite (packed storage)
<b>he</b>	Hermitian indefinite
<b>hp</b>	Hermitian indefinite (packed storage)
<b>tr</b>	triangular
<b>tp</b>	triangular (packed storage)
<b>tb</b>	triangular band

For computational routines, the last three letters **zzz** indicate the computation performed:

<b>trf</b>	form a triangular matrix factorization
<b>trs</b>	solve the linear system with a factored matrix
<b>con</b>	estimate the matrix condition number
<b>rfs</b>	refine the solution and compute error bounds
<b>tri</b>	compute the inverse matrix using the factorization
<b>equ</b>	equilibrate a matrix.

For example, the routine **sgetrf** performs the triangular factorization of general real matrices in single precision; the corresponding routine for complex matrices is **cgetrf**.

For driver routines, the names can end either with **-sv** (meaning a *simple* driver), or with **-svx** (meaning an *expert* driver).

## Matrix Storage Schemes

LAPACK routines use the following matrix storage schemes:

- *Full storage*: a matrix  $A$  is stored in a two-dimensional array  $a$ , with the matrix element  $a_{ij}$  stored in the array element  $a(i, j)$ .
- *Packed storage* scheme allows you to store symmetric, Hermitian, or triangular matrices more compactly: the upper or lower triangle of the matrix is packed by columns in a one-dimensional array.
- *Band storage*: an  $m$  by  $n$  band matrix with  $k_1$  sub-diagonals and  $k_u$  super-diagonals is stored compactly in a two-dimensional array  $ab$  with  $k_1+k_u+1$  rows and  $n$  columns. Columns of the matrix are stored in the corresponding columns of the array, and *diagonals* of the matrix are stored in rows of the array.

In Chapters 4 and 5, arrays that hold matrices in packed storage have names ending in  $p$ ; arrays with matrices in band storage have names ending in  $b$ .

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

## Mathematical Notation

Descriptions of LAPACK routines use the following notation:

$Ax = b$	A system of linear equations with an $n$ by $n$ matrix $A = \{a_{ij}\}$ , a right-hand side vector $b = \{b_i\}$ , and an unknown vector $x = \{x_i\}$ .
$AX = B$	A set of systems with a common matrix $A$ and multiple right-hand sides. The columns of $B$ are individual right-hand sides, and the columns of $X$ are the corresponding solutions.
$ x $	the vector with elements $ x_i $ (absolute values of $x_i$ ).
$ A $	the matrix with elements $ a_{ij} $ (absolute values of $a_{ij}$ ).
$\ x\ _\infty = \max_i  x_i $	The <i>infinity-norm</i> of the vector $x$ .
$\ A\ _\infty = \max_i \sum_j  a_{ij} $	The <i>infinity-norm</i> of the matrix $A$ .
$\ A\ _1 = \max_j \sum_i  a_{ij} $	The <i>one-norm</i> of the matrix $A$ . $\ A\ _1 = \ A^T\ _\infty = \ A^H\ _\infty$
$\kappa(A) = \ A\  \ A^{-1}\ $	The <i>condition number</i> of the matrix $A$ .

## Error Analysis

In practice, most computations are performed with rounding errors. Besides, you often need to solve a system  $Ax = b$  where the data (the elements of  $A$  and  $b$ ) are not known exactly. Therefore, it's important to understand how the data errors and rounding errors can affect the solution  $x$ .

**Data perturbations.** If  $x$  is the exact solution of  $Ax = b$ , and  $x + \delta x$  is the exact solution of a perturbed problem  $(A + \delta A)x = (b + \delta b)$ , then

$$\frac{\|\delta x\|}{\|x\|} \leq \kappa(A) \left( \frac{\|\delta A\|}{\|A\|} + \frac{\|\delta b\|}{\|b\|} \right), \text{ where } \kappa(A) = \|A\| \|A^{-1}\|.$$

In other words, relative errors in  $A$  or  $b$  may be amplified in the solution vector  $x$  by a factor  $\kappa(A) = \|A\| \|A^{-1}\|$  called the *condition number* of  $A$ .

**Rounding errors** have the same effect as relative perturbations  $c(n)\varepsilon$  in the original data. Here  $\varepsilon$  is the *machine precision*, and  $c(n)$  is a modest function of the matrix order  $n$ . The corresponding solution error is  $\|\delta x\|/\|x\| \leq c(n)\kappa(A)\varepsilon$ . (The value of  $c(n)$  is seldom greater than  $10n$ .)

Thus, if your matrix  $A$  is *ill-conditioned* (that is, its condition number  $\kappa(A)$  is very large), then the error in the solution  $x$  is also large; you may even encounter a complete loss of precision. LAPACK provides routines that allow you to estimate  $\kappa(A)$  (see [Routines for Estimating the Condition Number](#)) and also give you a more precise estimate for the actual solution error (see [Refining the Solution and Estimating Its Error](#)).

## Computational Routines

[Table 4-1](#) lists the LAPACK computational routines for factorizing, equilibrating, and inverting *real* matrices, estimating their condition numbers, solving systems of equations with real matrices, refining the solution, and estimating its error.

[Table 4-2](#) lists similar routines for *complex* matrices.

**Table 4-1 Computational Routines for Systems of Equations with Real Matrices**

Matrix type, storage scheme	Factorize matrix	Equilibrate matrix	Solve system	Condition number	Estimate error	Invert matrix
general	<a href="#">?getrf</a>	<a href="#">?geequ</a>	<a href="#">?getrs</a>	<a href="#">?gecon</a>	<a href="#">?gerfs</a>	<a href="#">?getri</a>
general band	<a href="#">?gbtrf</a>	<a href="#">?gbequ</a>	<a href="#">?gbtrs</a>	<a href="#">?gbcon</a>	<a href="#">?gbrfs</a>	
general tridiagonal	<a href="#">?gttrf</a>		<a href="#">?gttrs</a>	<a href="#">?gtcon</a>	<a href="#">?gtrfs</a>	
symmetric positive-definite	<a href="#">?potrf</a>	<a href="#">?poequ</a>	<a href="#">?potrs</a>	<a href="#">?pocon</a>	<a href="#">?porfs</a>	<a href="#">?potri</a>
symmetric positive-definite, packed storage	<a href="#">?pptrf</a>	<a href="#">?ppequ</a>	<a href="#">?pptrs</a>	<a href="#">?ppcon</a>	<a href="#">?pprfs</a>	<a href="#">?pptri</a>
symmetric positive-definite, band	<a href="#">?pbtrf</a>	<a href="#">?pbequ</a>	<a href="#">?pbtrs</a>	<a href="#">?pbcon</a>	<a href="#">?pbrfs</a>	
symmetric positive-definite, tridiagonal	<a href="#">?pttrf</a>		<a href="#">?pttrs</a>	<a href="#">?ptcon</a>	<a href="#">?ptrfs</a>	
symmetric indefinite	<a href="#">?sytrf</a>		<a href="#">?sytrs</a>	<a href="#">?sycon</a>	<a href="#">?syrfs</a>	<a href="#">?sytri</a>
symmetric indefinite, packed storage	<a href="#">?sptrf</a>		<a href="#">?sptrs</a>	<a href="#">?spcon</a>	<a href="#">?sprfs</a>	<a href="#">?sptri</a>
triangular			<a href="#">?trtrs</a>	<a href="#">?trcon</a>	<a href="#">?trrfs</a>	<a href="#">?trtri</a>
triangular, packed storage			<a href="#">?tptrs</a>	<a href="#">?tpcon</a>	<a href="#">?tprfs</a>	<a href="#">?tptri</a>
triangular band			<a href="#">?tbtrs</a>	<a href="#">?tbcon</a>	<a href="#">?tbrfs</a>	

In this table ? denotes **s** (single precision) or **d** (double precision).

**Table 4-2 Computational Routines for Systems of Equations with Complex Matrices**

Matrix type, storage scheme	Factorize matrix	Equilibrate matrix	Solve system	Condition number	Estimate error	Invert matrix
general	<a href="#">?getrf</a>	<a href="#">?geequ</a>	<a href="#">?getrs</a>	<a href="#">?gecon</a>	<a href="#">?gerfs</a>	<a href="#">?getri</a>
general band	<a href="#">?gbtrf</a>	<a href="#">?gbequ</a>	<a href="#">?gbtrs</a>	<a href="#">?gbcon</a>	<a href="#">?gbrfs</a>	
general tridiagonal	<a href="#">?gttrf</a>		<a href="#">?gttrs</a>	<a href="#">?gtcon</a>	<a href="#">?gtrfs</a>	
Hermitian positive-definite	<a href="#">?potrf</a>	<a href="#">?poequ</a>	<a href="#">?potrs</a>	<a href="#">?pocon</a>	<a href="#">?porfs</a>	<a href="#">?potri</a>
Hermitian positive-definite, packed storage	<a href="#">?pptrf</a>	<a href="#">?ppequ</a>	<a href="#">?pptrs</a>	<a href="#">?ppcon</a>	<a href="#">?pprfs</a>	<a href="#">?pptri</a>
Hermitian positive-definite, band	<a href="#">?pbtrf</a>	<a href="#">?pbequ</a>	<a href="#">?pbtrs</a>	<a href="#">?pbcon</a>	<a href="#">?pbrfs</a>	
Hermitian positive-definite, tridiagonal	<a href="#">?pttrf</a>		<a href="#">?pttrs</a>	<a href="#">?ptcon</a>	<a href="#">?ptrfs</a>	
Hermitian indefinite	<a href="#">?hetrf</a>		<a href="#">?hetrs</a>	<a href="#">?hecon</a>	<a href="#">?herfs</a>	<a href="#">?hetri</a>
symmetric indefinite	<a href="#">?sytrf</a>		<a href="#">?sytrs</a>	<a href="#">?sycon</a>	<a href="#">?syrfs</a>	<a href="#">?sytri</a>
Hermitian indefinite, packed storage	<a href="#">?hptrf</a>		<a href="#">?hptrs</a>	<a href="#">?hpcon</a>	<a href="#">?hprfs</a>	<a href="#">?hptri</a>
symmetric indefinite, packed storage	<a href="#">?sptrf</a>		<a href="#">?sptrs</a>	<a href="#">?spcon</a>	<a href="#">?sprfs</a>	<a href="#">?sptri</a>
triangular			<a href="#">?trtrs</a>	<a href="#">?trcon</a>	<a href="#">?trrfs</a>	<a href="#">?trtri</a>
triangular, packed storage			<a href="#">?tptrs</a>	<a href="#">?tpcon</a>	<a href="#">?tprfs</a>	<a href="#">?tptri</a>
triangular band			<a href="#">?tbtrs</a>	<a href="#">?tbcon</a>	<a href="#">?tbrfs</a>	

In this table ? stands for **c** (single precision complex) or **z** (double precision complex).

## Routines for Matrix Factorization

This section describes the LAPACK routines for matrix factorization. The following factorizations are supported:

- *LU* factorization
- Cholesky factorization of real symmetric positive-definite matrices
- Cholesky factorization of Hermitian positive-definite matrices
- Bunch-Kaufman factorization of real and complex symmetric matrices
- Bunch-Kaufman factorization of Hermitian matrices.

You can compute the *LU* factorization using full and band storage of matrices; the Cholesky factorization using full, packed, and band storage; and the Bunch-Kaufman factorization using full and packed storage.

### ?getrf

*Computes the LU factorization of a general m by n matrix.*

```
call sgetrf ( m, n, a, lda, ipiv, info )
call dgetrf ( m, n, a, lda, ipiv, info )
call cgetrf ( m, n, a, lda, ipiv, info )
call zgetrf ( m, n, a, lda, ipiv, info )
```

#### Discussion

The routine forms the *LU* factorization of a general  $m$  by  $n$  matrix  $A$  as

$$A = PLU$$

where  $P$  is a permutation matrix,  $L$  is lower triangular with unit diagonal elements (lower trapezoidal if  $m > n$ ) and  $U$  is upper triangular (upper trapezoidal if  $m < n$ ). Usually  $A$  is square ( $m = n$ ), and both  $L$  and  $U$  are triangular. The routine uses partial pivoting, with row interchanges.

## Input Parameters

<i>m</i>	<b>INTEGER.</b> The number of rows in the matrix <i>A</i> ( <i>m</i> ≥ 0).
<i>n</i>	<b>INTEGER.</b> The number of columns in <i>A</i> ( <i>n</i> ≥ 0).
<i>a</i>	<b>REAL</b> for <b>sgetrf</b> <b>DOUBLE PRECISION</b> for <b>dgetrf</b> <b>COMPLEX</b> for <b>cgetrf</b> <b>DOUBLE COMPLEX</b> for <b>zgetrf</b> . Array, <b>DIMENSION</b> ( <i>lda</i> , *). Contains the matrix <i>A</i> . The second dimension of <i>a</i> must be at least <b>max</b> (1, <i>n</i> ).
<i>lda</i>	<b>INTEGER.</b> The first dimension of <i>a</i> .

## Output Parameters

<i>a</i>	Overwritten by <i>L</i> and <i>U</i> . The unit diagonal elements of <i>L</i> are not stored.
<i>ipiv</i>	<b>INTEGER.</b> Array, <b>DIMENSION</b> at least <b>max</b> (1, <b>min</b> ( <i>m</i> , <i>n</i> )). The pivot indices: row <i>i</i> was interchanged with row <i>ipiv(i)</i> .
<i>info</i>	<b>INTEGER.</b> If <i>info</i> =0, the execution is successful. If <i>info</i> = <i>-i</i> , the <i>i</i> th parameter had an illegal value. If <i>info</i> = <i>i</i> , <i>u<sub>ii</sub></i> is 0. The factorization has been completed, but <i>U</i> is exactly singular. Division by 0 will occur if you use the factor <i>U</i> for solving a system of linear equations.

## Application Notes

The computed *L* and *U* are the exact factors of a perturbed matrix *A* + *E*, where

$$|E| \leq c(\min(m, n))\varepsilon P|L||U|$$

*c(n)* is a modest linear function of *n*, and  $\varepsilon$  is the machine precision.

The approximate number of floating-point operations for real flavors is

$$\begin{aligned} (2/3)n^3 & \quad \text{if } m = n, \\ (1/3)n^2(3m-n) & \quad \text{if } m > n, \end{aligned}$$

$$(1/3)\textcolor{red}{m}^2(3\textcolor{red}{n}-\textcolor{red}{m}) \quad \text{if } \textcolor{red}{m} < \textcolor{red}{n}.$$

The number of operations for complex flavors is 4 times greater.

After calling this routine with  $\textcolor{red}{m} = \textcolor{red}{n}$ , you can call the following:

[?getrs](#) to solve  $AX = B$  or  $A^T X = B$  or  $A^H X = B$ ;

[?gecon](#) to estimate the condition number of  $A$ ;

[?getri](#) to compute the inverse of  $A$ .

## ?gbtrf

*Computes the LU factorization of a general m by n band matrix.*

---

```
call sgbtrf ( m, n, kl, ku, ab, ldab, ipiv, info )
call dgbtrf ( m, n, kl, ku, ab, ldab, ipiv, info )
call cgbtrf ( m, n, kl, ku, ab, ldab, ipiv, info )
call zgbtrf ( m, n, kl, ku, ab, ldab, ipiv, info )
```

### Discussion

The routine forms the LU factorization of a general  $m$  by  $n$  band matrix  $A$  with  $kl$  non-zero sub-diagonals and  $ku$  non-zero super-diagonals. Usually  $A$  is square ( $m = n$ ), and then

$$A = PLU$$

where  $P$  is a permutation matrix;  $L$  is lower triangular with unit diagonal elements and at most  $kl$  non-zero elements in each column;  $U$  is an upper triangular band matrix with  $kl + ku$  super-diagonals. The routine uses partial pivoting, with row interchanges (which creates the additional  $kl$  super-diagonals in  $U$ ).

### Input Parameters

$m$	INTEGER. The number of rows in the matrix $A$ ( $m \geq 0$ ).
$n$	INTEGER. The number of columns in $A$ ( $n \geq 0$ ).
$kl$	INTEGER. The number of sub-diagonals within the band of $A$ ( $kl \geq 0$ ).
$ku$	INTEGER. The number of super-diagonals within the band of $A$ ( $ku \geq 0$ ).
$ab$	REAL for <b>sgbtrf</b> DOUBLE PRECISION for <b>dgbtrf</b> COMPLEX for <b>cgbtrf</b> DOUBLE COMPLEX for <b>zgbtrf</b> . Array, DIMENSION ( $ldab, *$ ).

The array  $\text{ab}$  contains the matrix  $A$  in band storage  
(see [Matrix Storage Schemes](#)).

The second dimension of  $\text{ab}$  must be at least  $\max(1, n)$ .

$\text{ldab}$       **INTEGER.** The first dimension of the array  $\text{ab}$ .  
( $\text{ldab} \geq 2\text{k}_1 + \text{ku} + 1$ )

### Output Parameters

$\text{ab}$       Overwritten by  $L$  and  $U$ . The diagonal and  $\text{k}_1 + \text{ku}$  super-diagonals of  $U$  are stored in the first  $1 + \text{k}_1 + \text{ku}$  rows of  $\text{ab}$ . The multipliers used to form  $L$  are stored in the next  $\text{k}_1$  rows.

$\text{ipiv}$       **INTEGER.**  
Array, **DIMENSION** at least  $\max(1, \min(m, n))$ .  
The pivot indices: row  $i$  was interchanged with row  $\text{ipiv}(i)$ .

$\text{info}$       **INTEGER.** If  $\text{info} = 0$ , the execution is successful.  
If  $\text{info} = -i$ , the  $i$ th parameter had an illegal value.  
If  $\text{info} = i$ ,  $u_{ii}$  is 0. The factorization has been completed, but  $U$  is exactly singular. Division by 0 will occur if you use the factor  $U$  for solving a system of linear equations.

### Application Notes

The computed  $L$  and  $U$  are the exact factors of a perturbed matrix  $A + E$ , where

$$|E| \leq c(\text{k}_1 + \text{ku} + 1) \varepsilon P |L| |U|$$

$c(k)$  is a modest linear function of  $k$ , and  $\varepsilon$  is the machine precision.

The total number of floating-point operations for real flavors varies between approximately  $2n(\text{ku}+1)\text{k}_1$  and  $2n(\text{k}_1+\text{ku}+1)\text{k}_1$ . The number of operations for complex flavors is 4 times greater. All these estimates assume that  $\text{k}_1$  and  $\text{ku}$  are much less than  $\min(m, n)$ .

After calling this routine with  $m = n$ , you can call the following:

[?gbtrs](#)      to solve  $AX = B$  or  $A^T X = B$  or  $A^H X = B$ ;

[?gbcon](#)      to estimate the condition number of  $A$ .

---

## ?gttrf

*Computes the LU factorization of a tridiagonal matrix.*

---

```
call sgttrf ( n, dl, d, du, du2, ipiv, info )
call dgttrf ( n, dl, d, du, du2, ipiv, info )
call cgtrf ( n, dl, d, du, du2, ipiv, info )
call zgttrf ( n, dl, d, du, du2, ipiv, info )
```

### Discussion

The routine computes the *LU* factorization of a real or complex tridiagonal matrix  $A$  in the form

$$A = PLU$$

where  $P$  is a permutation matrix;  $L$  is lower bidiagonal with unit diagonal elements; and  $U$  is an upper triangular matrix with nonzeros in only the main diagonal and first two superdiagonals. The routine uses elimination with partial pivoting and row interchanges .

### Input Parameters

<i>n</i>	INTEGER. The order of the matrix $A$ ( $n \geq 0$ ).
<i>dl, d, du</i>	REAL for <b>sgttrf</b> DOUBLE PRECISION for <b>dgttrf</b> COMPLEX for <b>cgttrf</b> DOUBLE COMPLEX for <b>zgttrf</b> . Arrays containing elements of $A$ . The array <i>dl</i> of dimension ( <i>n</i> - 1) contains the sub-diagonal elements of $A$ . The array <i>d</i> of dimension <i>n</i> contains the diagonal elements of $A$ . The array <i>du</i> of dimension ( <i>n</i> - 1) contains the super-diagonal elements of $A$ .

## Output Parameters

<i>dl</i>	Overwritten by the ( <i>n</i> -1) multipliers that define the matrix <i>L</i> from the <i>LU</i> factorization of <i>A</i> .
<i>d</i>	Overwritten by the <i>n</i> diagonal elements of the upper triangular matrix <i>U</i> from the <i>LU</i> factorization of <i>A</i> .
<i>du</i>	Overwritten by the ( <i>n</i> -1) elements of the first super-diagonal of <i>U</i> .
<i>du2</i>	<small>REAL for <code>sgttrf</code> DOUBLE PRECISION for <code>dgttrf</code> COMPLEX for <code>cgttrf</code> DOUBLE COMPLEX for <code>zgttrf</code>.</small> Array, dimension ( <i>n</i> -2). On exit, <i>du2</i> contains ( <i>n</i> -2) elements of the second super-diagonal of <i>U</i> .
<i>ipiv</i>	<small>INTEGER.</small> Array, dimension ( <i>n</i> ). The pivot indices: row <i>i</i> was interchanged with row <i>ipiv(i)</i> .
<i>info</i>	<small>INTEGER.</small> If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> th parameter had an illegal value. If <i>info</i> = <i>i</i> , <i>u<sub>ii</sub></i> is 0. The factorization has been completed, but <i>U</i> is exactly singular. Division by zero will occur if you use the factor <i>U</i> for solving a system of linear equations.

## Application Notes

- [?gbtrs](#) to solve  $AX = B$  or  $A^T X = B$  or  $A^H X = B$ ;  
[?gbcon](#) to estimate the condition number of *A*.

---

## ?potrf

*Computes the Cholesky factorization of a symmetric (Hermitian) positive-definite matrix.*

---

```
call spotrf ( uplo, n, a, lda, info )
call dpotrf ( uplo, n, a, lda, info )
call cpotrf ( uplo, n, a, lda, info )
call zpotrf ( uplo, n, a, lda, info )
```

### Discussion

This routine forms the Cholesky factorization of a symmetric positive-definite or, for complex data, Hermitian positive-definite matrix  $A$ :

$$\begin{aligned} A &= U^H U && \text{if } \text{uplo} = \text{'U'} \\ A &= LL^H && \text{if } \text{uplo} = \text{'L'} \end{aligned}$$

where  $L$  is a lower triangular matrix and  $U$  is upper triangular.

### Input Parameters

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. Indicates whether the upper or lower triangular part of $A$ is stored and how $A$ is factored: If <i>uplo</i> = 'U', the array <i>a</i> stores the upper triangular part of the matrix $A$ , and $A$ is factored as $U^H U$ . If <i>uplo</i> = 'L', the array <i>a</i> stores the lower triangular part of the matrix $A$ ; $A$ is factored as $LL^H$ .
<i>n</i>	INTEGER. The order of matrix $A$ ( $n \geq 0$ ).
<i>a</i>	REAL for <b>spotrf</b> DOUBLE PRECISION for <b>dpotrf</b> COMPLEX for <b>cpotrf</b> DOUBLE COMPLEX for <b>zpotrf</b> . Array, DIMENSION ( <i>lda</i> , *).

The array  $a$  contains either the upper or the lower triangular part of the matrix  $A$  (see  $uplo$ ).

The second dimension of  $a$  must be at least  $\max(1, n)$ .

$lda$

**INTEGER.** The first dimension of  $a$ .

## Output Parameters

$a$

The upper or lower triangular part of  $a$  is overwritten by the Cholesky factor  $U$  or  $L$ , as specified by  $uplo$ .

$info$

**INTEGER.** If  $info=0$ , the execution is successful.

If  $info = -i$ , the  $i$ th parameter had an illegal value.

If  $info = i$ , the leading minor of order  $i$  (and hence the matrix  $A$  itself) is not positive-definite, and the factorization could not be completed. This may indicate an error in forming the matrix  $A$ .

## Application Notes

If  $uplo = 'U'$ , the computed factor  $U$  is the exact factor of a perturbed matrix  $A + E$ , where

$$|E| \leq c(n)\epsilon |U^H||U|, \quad |e_{ij}| \leq c(n)\epsilon \sqrt{a_{ii}a_{jj}}$$

$c(n)$  is a modest linear function of  $n$ , and  $\epsilon$  is the machine precision.

A similar estimate holds for  $uplo = 'L'$ .

The total number of floating-point operations is approximately  $(1/3)n^3$  for real flavors or  $(4/3)n^3$  for complex flavors.

After calling this routine, you can call the following:

?potrs

to solve  $AX = B$ ;

?pocon

to estimate the condition number of  $A$ ;

?potri

to compute the inverse of  $A$ .

---

## ?pptrf

*Computes the Cholesky factorization of a symmetric (Hermitian) positive-definite matrix using packed storage.*

---

```
call spptrf ( uplo, n, ap, info )
call dpptrf ( uplo, n, ap, info )
call cpptrf ( uplo, n, ap, info )
call zpptrf ( uplo, n, ap, info )
```

### Discussion

This routine forms the Cholesky factorization of a symmetric positive-definite or, for complex data, Hermitian positive-definite packed matrix  $A$ :

$$\begin{aligned} A &= U^H U && \text{if } \text{uplo} = 'U' \\ A &= LL^H && \text{if } \text{uplo} = 'L' \end{aligned}$$

where  $L$  is a lower triangular matrix and  $U$  is upper triangular.

### Input Parameters

<i>uplo</i>	CHARACTER*1. Must be ' <i>U</i> ' or ' <i>L</i> '.
	Indicates whether the upper or lower triangular part of $A$ is packed in the array <i>ap</i> , and how $A$ is factored:
	If <i>uplo</i> = ' <i>U</i> ', the array <i>ap</i> stores the upper triangular part of the matrix $A$ , and $A$ is factored as $U^H U$ .
	If <i>uplo</i> = ' <i>L</i> ', the array <i>ap</i> stores the lower triangular part of the matrix $A$ ; $A$ is factored as $LL^H$ .
<i>n</i>	INTEGER. The order of matrix $A$ ( <i>n</i> $\geq 0$ ).
<i>ap</i>	REAL for <i>spptrf</i> DOUBLE PRECISION for <i>dpptrf</i> COMPLEX for <i>cpptrf</i> DOUBLE COMPLEX for <i>zpptrf</i> . Array, DIMENSION at least $\max(1, n(n+1)/2)$ .

The array `ap` contains either the upper or the lower triangular part of the matrix  $A$  (as specified by `uplo`) in *packed storage* (see [Matrix Storage Schemes](#)).

## Output Parameters

<code>ap</code>	The upper or lower triangular part of $A$ in packed storage is overwritten by the Cholesky factor $U$ or $L$ , as specified by <code>uplo</code> .
<code>info</code>	<p><code>INTEGER</code>. If <code>info</code>=0, the execution is successful.</p> <p>If <code>info</code> = <math>-i</math>, the <math>i</math>th parameter had an illegal value.</p> <p>If <code>info</code> = <math>i</math>, the leading minor of order <math>i</math> (and hence the matrix <math>A</math> itself) is not positive-definite, and the factorization could not be completed. This may indicate an error in forming the matrix <math>A</math>.</p>

## Application Notes

If `uplo` = '`U`', the computed factor  $U$  is the exact factor of a perturbed matrix  $A + E$ , where

$$|E| \leq c(\mathbf{n})\varepsilon|U^H||U|, \quad |e_{ij}| \leq c(\mathbf{n})\varepsilon\sqrt{a_{ii}a_{jj}}$$

$c(\mathbf{n})$  is a modest linear function of  $\mathbf{n}$ , and  $\varepsilon$  is the machine precision.

A similar estimate holds for `uplo` = '`L`'.

The total number of floating-point operations is approximately  $(1/3)\mathbf{n}^3$  for real flavors and  $(4/3)\mathbf{n}^3$  for complex flavors.

After calling this routine, you can call the following:

<a href="#"><code>?ptrs</code></a>	to solve $AX = B$ ;
<a href="#"><code>?ppcon</code></a>	to estimate the condition number of $A$ ;
<a href="#"><code>?pptri</code></a>	to compute the inverse of $A$ .

---

## ?pbtrf

*Computes the Cholesky factorization of a symmetric (Hermitian) positive-definite band matrix.*

---

```
call spbtrf ( uplo, n, kd, ab, ldab, info )
call dpbtrf ( uplo, n, kd, ab, ldab, info )
call cpbtrf ( uplo, n, kd, ab, ldab, info )
call zpbtrf ( uplo, n, kd, ab, ldab, info )
```

### Discussion

This routine forms the Cholesky factorization of a symmetric positive-definite or, for complex data, Hermitian positive-definite band matrix  $A$ :

$$\begin{aligned} A &= U^H U && \text{if } \text{uplo} = 'U' \\ A &= LL^H && \text{if } \text{uplo} = 'L' \end{aligned}$$

where  $L$  is a lower triangular matrix and  $U$  is upper triangular.

### Input Parameters

<i>uplo</i>	CHARACTER*1. Must be ' <i>U</i> ' or ' <i>L</i> '. Indicates whether the upper or lower triangular part of $A$ is stored in the array <i>ab</i> , and how $A$ is factored: If <i>uplo</i> = ' <i>U</i> ', the array <i>ab</i> stores the upper triangular part of the matrix $A$ , and $A$ is factored as $U^H U$ . If <i>uplo</i> = ' <i>L</i> ', the array <i>ab</i> stores the lower triangular part of the matrix $A$ ; $A$ is factored as $LL^H$ .
<i>n</i>	INTEGER. The order of matrix $A$ ( <i>n</i> $\geq$ 0).
<i>kd</i>	INTEGER. The number of super-diagonals or sub-diagonals in the matrix $A$ ( <i>kd</i> $\geq$ 0).
<i>ab</i>	REAL for <i>spbtrf</i> DOUBLE PRECISION for <i>dpbtrf</i> COMPLEX for <i>cpbtrf</i> DOUBLE COMPLEX for <i>zpbtrf</i> . Array, DIMENSION ( <i>ldab</i> ,*).

The array  $\text{ap}$  contains either the upper or the lower triangular part of the matrix  $A$  (as specified by  $\text{uplo}$ ) in *band storage* (see [Matrix Storage Schemes](#)).

The second dimension of  $\text{ab}$  must be at least  $\max(1, n)$ .

$\text{ldab}$       **INTEGER.** The first dimension of the array  $\text{ab}$ .  
 $(\text{ldab} \geq \text{kd} + 1)$

### Output Parameters

$\text{ap}$       The upper or lower triangular part of  $A$  (in band storage) is overwritten by the Cholesky factor  $U$  or  $L$ , as specified by  $\text{uplo}$ .

$\text{info}$       **INTEGER.** If  $\text{info}=0$ , the execution is successful.  
If  $\text{info} = -i$ , the  $i$ th parameter had an illegal value.  
If  $\text{info} = i$ , the leading minor of order  $i$  (and hence the matrix  $A$  itself) is not positive-definite, and the factorization could not be completed. This may indicate an error in forming the matrix  $A$ .

### Application Notes

If  $\text{uplo} = \text{'U'}$ , the computed factor  $U$  is the exact factor of a perturbed matrix  $A + E$ , where

$$|E| \leq c(\text{kd} + 1)\varepsilon|U^H||U|, \quad |e_{ij}| \leq c(\text{kd} + 1)\varepsilon\sqrt{a_{ii}a_{jj}}$$

$c(n)$  is a modest linear function of  $n$ , and  $\varepsilon$  is the machine precision.

A similar estimate holds for  $\text{uplo} = \text{'L'}$ .

The total number of floating-point operations for real flavors is approximately  $n(\text{kd}+1)^2$ . The number of operations for complex flavors is 4 times greater. All these estimates assume that  $\text{kd}$  is much less than  $n$ .

After calling this routine, you can call the following:

<a href="#"><u>?pbtrs</u></a>	to solve $AX = B$ ;
<a href="#"><u>?pbcon</u></a>	to estimate the condition number of $A$ ;

---

## ?pttrf

*Computes the factorization of  
a symmetric (Hermitian) positive-definite  
tridiagonal matrix.*

---

```
call spttrf ( n, d, e, info )
call dpttrf ( n, d, e, info )
call cpttrf ( n, d, e, info )
call zpttrf ( n, d, e, info )
```

### Discussion

This routine forms the factorization of a symmetric positive-definite or, for complex data, Hermitian positive-definite tridiagonal matrix  $A$ :

$A = LDL^H$ , where  $D$  is diagonal and  $L$  is unit lower bidiagonal. The factorization may also be regarded as having the form  $A = U^H D U$ , where  $D$  is unit upper bidiagonal.

### Input Parameters

<i>n</i>	INTEGER. The order of the matrix $A$ ( $n \geq 0$ ).
<i>d</i>	REAL for <b>spttrf</b> , <b>cpttrf</b> DOUBLE PRECISION for <b>dpttrf</b> , <b>zpttrf</b> . Array, dimension ( <i>n</i> ). Contains the diagonal elements of $A$ .
<i>e</i>	REAL for <b>spttrf</b> DOUBLE PRECISION for <b>dpttrf</b> COMPLEX for <b>cpttrf</b> DOUBLE COMPLEX for <b>zpttrf</b> . Array, dimension ( <i>n</i> - 1). Contains the sub-diagonal elements of $A$ .

### Output Parameters

<i>d</i>	Overwritten by the <i>n</i> diagonal elements of the diagonal matrix $D$ from the $LDL^H$ factorization of $A$ .
----------	--

*e* Overwritten by the (*n* - 1) off-diagonal elements of the unit bidiagonal factor *L* or *U* from the factorization of *A*.

*info* INTEGER. If *info*=0, the execution is successful.

If *info* = *-i*, the *i*th parameter had an illegal value.

If *info* = *i*, the leading minor of order *i* (and hence the matrix *A* itself) is not positive-definite; if *i* < *n*, the factorization could not be completed, while if *i* = *n*, the factorization was completed, but *d* (*n*) = 0 .

---

## ?sytrf

*Computes the Bunch-Kaufman factorization of a symmetric matrix.*

---

```
call ssytrf ( uplo, n, a, lda, ipiv, work, lwork, info )
call dsytrf ( uplo, n, a, lda, ipiv, work, lwork, info )
call csytrf ( uplo, n, a, lda, ipiv, work, lwork, info )
call zsytrf ( uplo, n, a, lda, ipiv, work, lwork, info )
```

### Discussion

This routine forms the Bunch-Kaufman factorization of a symmetric matrix:

$$\begin{aligned} \text{if } \text{uplo} = 'U', \quad A &= PUDU^TP^T \\ \text{if } \text{uplo} = 'L', \quad A &= PLDL^TP^T \end{aligned}$$

where  $A$  is the input matrix,  $P$  is a permutation matrix,  $U$  and  $L$  are upper and lower triangular matrices with unit diagonal, and  $D$  is a symmetric block-diagonal matrix with 1-by-1 and 2-by-2 diagonal blocks.  $U$  and  $L$  have 2-by-2 unit diagonal blocks corresponding to the 2-by-2 blocks of  $D$ .

### Input Parameters

<i>uplo</i>	CHARACTER*1. Must be ' <i>U</i> ' or ' <i>L</i> '.
	Indicates whether the upper or lower triangular part of $A$ is stored and how $A$ is factored:
If <i>uplo</i> = ' <i>U</i> ',	the array <i>a</i> stores the upper triangular part of the matrix $A$ , and $A$ is factored as $PUDU^TP^T$ .
If <i>uplo</i> = ' <i>L</i> ',	the array <i>a</i> stores the lower triangular part of the matrix $A$ ; $A$ is factored as $PLDL^TP^T$ .
<i>n</i>	INTEGER. The order of matrix $A$ ( <i>n</i> $\geq 0$ ).
<i>a</i>	REAL for <b>ssytrf</b> DOUBLE PRECISION for <b>dsytrf</b> COMPLEX for <b>csytrf</b> DOUBLE COMPLEX for <b>zsytrf</b> . Array, DIMENSION ( <i>lda</i> , *).

The array  $a$  contains either the upper or the lower triangular part of the matrix  $A$  (see  $\text{uplo}$ ).

The second dimension of  $a$  must be at least  $\max(1, n)$ .

$\text{lda}$

$\text{work}$

$\text{lwork}$

**INTEGER.** The first dimension of  $a$ ; at least  $\max(1, n)$ .

Same type as  $a$ . Workspace array of dimension  $\text{lwork}$

**INTEGER.** The size of the  $\text{work}$  array ( $\text{lwork} \geq n$ )

See [Application notes](#) for the suggested value of  $\text{lwork}$ .

## Output Parameters

$a$

The upper or lower triangular part of  $a$  is overwritten by details of the block-diagonal matrix  $D$  and the multipliers used to obtain the factor  $U$  (or  $L$ ).

$\text{work}(1)$

If  $\text{info}=0$ , on exit  $\text{work}(1)$  contains the minimum value of  $\text{lwork}$  required for optimum performance. Use this  $\text{lwork}$  for subsequent runs.

$\text{ipiv}$

**INTEGER.**

Array, **DIMENSION** at least  $\max(1, n)$ .

Contains details of the interchanges and the block structure of  $D$ .

If  $\text{ipiv}(i) = k > 0$ , then  $d_{ii}$  is a 1-by-1 block, and the  $i$ th row and column of  $A$  was interchanged with the  $k$ th row and column.

If  $\text{uplo} = 'U'$  and  $\text{ipiv}(i) = \text{ipiv}(i-1) = -m < 0$ , then  $D$  has a 2-by-2 block in rows/columns  $i$  and  $i-1$ , and  $(i-1)$ th row and column of  $A$  was interchanged with the  $m$ th row and column.

If  $\text{uplo} = 'L'$  and  $\text{ipiv}(i) = \text{ipiv}(i+1) = -m < 0$ , then  $D$  has a 2-by-2 block in rows/columns  $i$  and  $i+1$ , and  $(i+1)$ th row and column of  $A$  was interchanged with the  $m$ th row and column.

$\text{info}$

**INTEGER.** If  $\text{info}=0$ , the execution is successful.

If  $\text{info} = -i$ , the  $i$ th parameter had an illegal value.

If  $\text{info} = i$ ,  $d_{ii}$  is 0. The factorization has been completed, but  $D$  is exactly singular. Division by 0 will occur if you use  $D$  for solving a system of linear equations.

## Application Notes

For better performance, try using `lwork = n * blocksize`, where `blocksize` is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use a generous value of `lwork` for the first run. On exit, examine `work(1)` and use this value for subsequent runs.

The 2-by-2 unit diagonal blocks and the unit diagonal elements of  $U$  and  $L$  are not stored. The remaining elements of  $U$  and  $L$  are stored in the corresponding columns of the array `a`, but additional row interchanges are required to recover  $U$  or  $L$  explicitly (which is seldom necessary).

If `ipiv(i) = i` for all  $i = 1 \dots n$ , then all off-diagonal elements of  $U$  ( $L$ ) are stored explicitly in the corresponding elements of the array `a`.

If `uplo = 'U'`, the computed factors  $U$  and  $D$  are the exact factors of a perturbed matrix  $A + E$ , where

$$|E| \leq c(n) \epsilon P |U| |D| |U^T| P^T$$

$c(n)$  is a modest linear function of  $n$ , and  $\epsilon$  is the machine precision.

A similar estimate holds for the computed  $L$  and  $D$  when `uplo = 'L'`.

The total number of floating-point operations is approximately  $(1/3)n^3$  for real flavors or  $(4/3)n^3$  for complex flavors.

After calling this routine, you can call the following:

<a href="#">?sytrs</a>	to solve $AX = B$ ;
<a href="#">?sycon</a>	to estimate the condition number of $A$ ;
<a href="#">?sytri</a>	to compute the inverse of $A$ .

## ?hetrf

*Computes the Bunch-Kaufman factorization of a complex Hermitian matrix.*

```
call chetrf ( uplo, n, a, lda, ipiv, work, lwork, info )
call zhetrf ( uplo, n, a, lda, ipiv, work, lwork, info )
```

### Discussion

This routine forms the Bunch-Kaufman factorization of a Hermitian matrix:

$$\begin{aligned} \text{if } \text{uplo} = \text{'U'}, \quad A &= PUDU^H P^T \\ \text{if } \text{uplo} = \text{'L'}, \quad A &= PLDL^H P^T \end{aligned}$$

where  $A$  is the input matrix,  $P$  is a permutation matrix,  $U$  and  $L$  are upper and lower triangular matrices with unit diagonal, and  $D$  is a Hermitian block-diagonal matrix with 1-by-1 and 2-by-2 diagonal blocks.  $U$  and  $L$  have 2-by-2 unit diagonal blocks corresponding to the 2-by-2 blocks of  $D$ .

### Input Parameters

<i>uplo</i>	CHARACTER*1. Must be ' <b>U</b> ' or ' <b>L</b> '.
	Indicates whether the upper or lower triangular part of $A$ is stored and how $A$ is factored:
If <i>uplo</i> = ' <b>U</b> ',	the array <i>a</i> stores the upper triangular part of the matrix $A$ , and $A$ is factored as $PUDU^H P^T$ .
If <i>uplo</i> = ' <b>L</b> ',	the array <i>a</i> stores the lower triangular part of the matrix $A$ ; $A$ is factored as $PLDL^H P^T$ .
<i>n</i>	INTEGER. The order of matrix $A$ ( $n \geq 0$ ).
<i>a</i>	COMPLEX for <b>chetrf</b> DOUBLE COMPLEX for <b>zhetrf</b> . Array, DIMENSION ( <i>lda</i> , *). The array <i>a</i> contains either the upper or the lower triangular part of the matrix $A$ (see <i>uplo</i> ). The second dimension of <i>a</i> must be at least $\max(1, n)$ .

<i>lda</i>	<b>INTEGER.</b> The first dimension of <i>a</i> ; at least $\max(1, n)$ .
<i>work</i>	Same type as <i>a</i> . Workspace array of dimension <i>lwork</i>
<i>lwork</i>	<b>INTEGER.</b> The size of the <i>work</i> array ( <i>lwork</i> $\geq n$ ) See <a href="#">Application notes</a> for the suggested value of <i>lwork</i> .

## Output Parameters

<i>a</i>	The upper or lower triangular part of <i>a</i> is overwritten by details of the block-diagonal matrix <i>D</i> and the multipliers used to obtain the factor <i>U</i> (or <i>L</i> ).
<i>work(1)</i>	If <i>info</i> =0, on exit <i>work(1)</i> contains the minimum value of <i>lwork</i> required for optimum performance. Use this <i>lwork</i> for subsequent runs.
<i>ipiv</i>	<b>INTEGER.</b> Array, <b>DIMENSION</b> at least $\max(1, n)$ . Contains details of the interchanges and the block structure of <i>D</i> . If <i>ipiv(i) = k &gt; 0</i> , then $d_{ii}$ is a 1-by-1 block, and the <i>i</i> th row and column of <i>A</i> was interchanged with the <i>k</i> th row and column. If <i>uplo = 'U'</i> and <i>ipiv(i) = ipiv(i-1) = -m &lt; 0</i> , then <i>D</i> has a 2-by-2 block in rows/columns <i>i</i> and <i>i-1</i> , and ( <i>i-1</i> )th row and column of <i>A</i> was interchanged with the <i>m</i> th row and column. If <i>uplo = 'L'</i> and <i>ipiv(i) = ipiv(i+1) = -m &lt; 0</i> , then <i>D</i> has a 2-by-2 block in rows/columns <i>i</i> and <i>i+1</i> , and ( <i>i+1</i> )th row and column of <i>A</i> was interchanged with the <i>m</i> th row and column.
<i>info</i>	<b>INTEGER.</b> If <i>info</i> =0, the execution is successful. If <i>info = -i</i> , the <i>i</i> th parameter had an illegal value. If <i>info = i</i> , $d_{ii}$ is 0. The factorization has been completed, but <i>D</i> is exactly singular. Division by 0 will occur if you use <i>D</i> for solving a system of linear equations.

## Application Notes

This routine is suitable for Hermitian matrices that are not known to be positive-definite. If  $A$  is in fact positive-definite, the routine does not perform interchanges, and no 2-by-2 diagonal blocks occur in  $D$ .

For better performance, try using  $\text{lwork} = n * \text{blocksize}$ , where  $\text{blocksize}$  is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use a generous value of  $\text{lwork}$  for the first run. On exit, examine  $\text{work}(1)$  and use this value for subsequent runs.

The 2-by-2 unit diagonal blocks and the unit diagonal elements of  $U$  and  $L$  are not stored. The remaining elements of  $U$  and  $L$  are stored in the corresponding columns of the array  $a$ , but additional row interchanges are required to recover  $U$  or  $L$  explicitly (which is seldom necessary).

If  $\text{ipiv}(i) = i$  for all  $i = 1 \dots n$ , then all off-diagonal elements of  $U$  ( $L$ ) are stored explicitly in the corresponding elements of the array  $a$ .

If  $\text{uplo} = 'U'$ , the computed factors  $U$  and  $D$  are the exact factors of a perturbed matrix  $A + E$ , where

$$|E| \leq c(n) \epsilon P |U| |D| |U^T| P^T$$

$c(n)$  is a modest linear function of  $n$ , and  $\epsilon$  is the machine precision.

A similar estimate holds for the computed  $L$  and  $D$  when  $\text{uplo} = 'L'$ .

The total number of floating-point operations is approximately  $(4/3)n^3$ .

After calling this routine, you can call the following:

- |                               |   |
|-------------------------------|---|
| <a href="#"><u>?hetrs</u></a> | to solve $AX = B$ ;                       |
| <a href="#"><u>?hecon</u></a> | to estimate the condition number of $A$ ; |
| <a href="#"><u>?hetri</u></a> | to compute the inverse of $A$ .           |

---

## ?sptrf

*Computes the Bunch-Kaufman factorization of a symmetric matrix using packed storage.*

---

```
call ssptrf ( uplo, n, ap, ipiv, info )
call dsptrf ( uplo, n, ap, ipiv, info )
call csptrf ( uplo, n, ap, ipiv, info )
call zsptrf ( uplo, n, ap, ipiv, info )
```

### Discussion

This routine forms the Bunch-Kaufman factorization of a symmetric matrix  $A$  using packed storage:

$$\begin{aligned} \text{if } \text{uplo} = 'U', \quad A &= PUDU^TP^T \\ \text{if } \text{uplo} = 'L', \quad A &= PLDL^TP^T \end{aligned}$$

where  $P$  is a permutation matrix,  $U$  and  $L$  are upper and lower triangular matrices with unit diagonal, and  $D$  is a symmetric block-diagonal matrix with 1-by-1 and 2-by-2 diagonal blocks.  $U$  and  $L$  have 2-by-2 unit diagonal blocks corresponding to the 2-by-2 blocks of  $D$ .

### Input Parameters

**uplo**                    CHARACTER\*1. Must be '**U**' or '**L**'.

Indicates whether the upper or lower triangular part of  $A$  is packed in the array **ap** and how  $A$  is factored:

If **uplo** = '**U**', the array **ap** stores the upper triangular part of the matrix  $A$ , and  $A$  is factored as  $PUDU^TP^T$ .

If **uplo** = '**L**', the array **ap** stores the lower triangular part of the matrix  $A$ ;  $A$  is factored as  $PLDL^TP^T$ .

**n**                    INTEGER. The order of matrix  $A$  ( $n \geq 0$ ).

*ap*      REAL for `ssptrf`  
DOUBLE PRECISION for `dsptrf`  
COMPLEX for `csptrf`  
DOUBLE COMPLEX for `zsptrf`.  
 Array, DIMENSION at least  $\max(1, n(n+1)/2)$ .  
 The array *ap* contains either the upper or the lower triangular part of the matrix  $A$  (as specified by *uplo*) in *packed storage* (see [Matrix Storage Schemes](#)).

### Output Parameters

*ap*      The upper or lower triangle of  $A$  (as specified by *uplo*) is overwritten by details of the block-diagonal matrix  $D$  and the multipliers used to obtain the factor  $U$  (or  $L$ ).

*ipiv*     INTEGER.  
 Array, DIMENSION at least  $\max(1, n)$ .  
 Contains details of the interchanges and the block structure of  $D$ .  
 If  $ipiv(i) = k > 0$ , then  $d_{ii}$  is a 1-by-1 block, and the *i*th row and column of  $A$  was interchanged with the *k*th row and column.  
 If *uplo* = 'U' and  $ipiv(i) = ipiv(i-1) = -m < 0$ , then  $D$  has a 2-by-2 block in rows/columns *i* and *i*-1, and (*i*-1)th row and column of  $A$  was interchanged with the *m*th row and column.  
 If *uplo* = 'L' and  $ipiv(i) = ipiv(i+1) = -m < 0$ , then  $D$  has a 2-by-2 block in rows/columns *i* and *i*+1, and (*i*+1)th row and column of  $A$  was interchanged with the *m*th row and column.

*info*     INTEGER. If *info*=0, the execution is successful.  
 If *info* = *-i*, the *i*th parameter had an illegal value.  
 If *info* = *i*,  $d_{ii}$  is 0. The factorization has been completed, but  $D$  is exactly singular. Division by 0 will occur if you use  $D$  for solving a system of linear equations.

## Application Notes

The 2-by-2 unit diagonal blocks and the unit diagonal elements of  $U$  and  $L$  are not stored. The remaining elements of  $U$  and  $L$  overwrite elements of the corresponding columns of the matrix  $A$ , but additional row interchanges are required to recover  $U$  or  $L$  explicitly (which is seldom necessary).

If  $\text{ipiv}(i) = i$  for all  $i = 1 \dots n$ , then all off-diagonal elements of  $U$  ( $L$ ) are stored explicitly in packed form.

If  $\text{uplo} = \text{'U'}$ , the computed factors  $U$  and  $D$  are the exact factors of a perturbed matrix  $A + E$ , where

$$|E| \leq c(n) \varepsilon P |U| |D| |U^T| P^T$$

$c(n)$  is a modest linear function of  $n$ , and  $\varepsilon$  is the machine precision.

A similar estimate holds for the computed  $L$  and  $D$  when  $\text{uplo} = \text{'L'}$ .

The total number of floating-point operations is approximately  $(1/3)n^3$  for real flavors or  $(4/3)n^3$  for complex flavors.

After calling this routine, you can call the following:

<a href="#">?sptrs</a>	to solve $AX = B$ ;
<a href="#">?spcon</a>	to estimate the condition number of $A$ ;
<a href="#">?sptri</a>	to compute the inverse of $A$ .

## ?hptrf

*Computes the Bunch-Kaufman factorization of a complex Hermitian matrix using packed storage.*

```
call chptrf ( uplo, n, ap, ipiv, info )
call zhptrf ( uplo, n, ap, ipiv, info )
```

### Discussion

This routine forms the Bunch-Kaufman factorization of a Hermitian matrix using packed storage:

```
if uplo='U',      A = PUDUHPT
if uplo='L',      A = PLDLHPT
```

where  $A$  is the input matrix,  $P$  is a permutation matrix,  $U$  and  $L$  are upper and lower triangular matrices with unit diagonal, and  $D$  is a Hermitian block-diagonal matrix with 1-by-1 and 2-by-2 diagonal blocks.  $U$  and  $L$  have 2-by-2 unit diagonal blocks corresponding to the 2-by-2 blocks of  $D$ .

### Input Parameters

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'.
	Indicates whether the upper or lower triangular part of $A$ is packed and how $A$ is factored:
	If <i>uplo</i> = 'U', the array <i>ap</i> stores the upper triangular part of the matrix $A$ , and $A$ is factored as $PUDU^H P^T$ .
	If <i>uplo</i> = 'L', the array <i>ap</i> stores the lower triangular part of the matrix $A$ ; $A$ is factored as $PLDL^H P^T$ .
<i>n</i>	INTEGER. The order of matrix $A$ ( $n \geq 0$ ).
<i>ap</i>	COMPLEX for <i>chptrf</i> DOUBLE COMPLEX for <i>zhptrf</i> . Array, DIMENSION at least max(1, $n(n+1)/2$ ).

The array  $\text{ap}$  contains either the upper or the lower triangular part of the matrix  $A$  (as specified by  $\text{uplo}$ ) in *packed storage* (see [Matrix Storage Schemes](#)).

## Output Parameters

$\text{ap}$	The upper or lower triangle of $A$ (as specified by $\text{uplo}$ ) is overwritten by details of the block-diagonal matrix $D$ and the multipliers used to obtain the factor $U$ (or $L$ ).
$\text{ipiv}$	<p><b>INTEGER.</b>          Array, <b>DIMENSION</b> at least <math>\max(1, n)</math>.          Contains details of the interchanges and the block structure of <math>D</math>.</p> <p>If <math>\text{ipiv}(i) = k &gt; 0</math>, then <math>d_{ii}</math> is a 1-by-1 block, and the <math>i</math>th row and column of <math>A</math> was interchanged with the <math>k</math>th row and column.</p> <p>If <math>\text{uplo} = 'U'</math> and <math>\text{ipiv}(i) = \text{ipiv}(i-1) = -m &lt; 0</math>, then <math>D</math> has a 2-by-2 block in rows/columns <math>i</math> and <math>i-1</math>, and (<math>i-1</math>)th row and column of <math>A</math> was interchanged with the <math>m</math>th row and column.</p> <p>If <math>\text{uplo} = 'L'</math> and <math>\text{ipiv}(i) = \text{ipiv}(i+1) = -m &lt; 0</math>, then <math>D</math> has a 2-by-2 block in rows/columns <math>i</math> and <math>i+1</math>, and (<math>i+1</math>)th row and column of <math>A</math> was interchanged with the <math>m</math>th row and column.</p>
$\text{info}$	<p><b>INTEGER.</b> If <math>\text{info}=0</math>, the execution is successful.          If <math>\text{info} = -i</math>, the <math>i</math>th parameter had an illegal value.          If <math>\text{info} = i</math>, <math>d_{ii}</math> is 0. The factorization has been completed, but <math>D</math> is exactly singular. Division by 0 will occur if you use <math>D</math> for solving a system of linear equations.</p>

### Application Notes

The 2-by-2 unit diagonal blocks and the unit diagonal elements of  $U$  and  $L$  are not stored. The remaining elements of  $U$  and  $L$  are stored in the corresponding columns of the array  $\text{a}$ , but additional row interchanges are required to recover  $U$  or  $L$  explicitly (which is seldom necessary).

If  $\text{ipiv}(i) = i$  for all  $i = 1 \dots n$ , then all off-diagonal elements of  $U$  ( $L$ ) are stored explicitly in the corresponding elements of the array  $\text{a}$ .

If  $\text{uplo} = \text{'U'}$ , the computed factors  $U$  and  $D$  are the exact factors of a perturbed matrix  $A + E$ , where

$$|E| \leq c(n) \varepsilon P |U| |D| |U^T| P^T$$

$c(n)$  is a modest linear function of  $n$ , and  $\varepsilon$  is the machine precision.

A similar estimate holds for the computed  $L$  and  $D$  when  $\text{uplo} = \text{'L'}$ .

The total number of floating-point operations is approximately  $(4/3)n^3$ .

After calling this routine, you can call the following:

- |                               |   |
|-------------------------------|---|
| <a href="#"><u>?hptrs</u></a> | to solve $AX = B$ ;                       |
| <a href="#"><u>?hpcon</u></a> | to estimate the condition number of $A$ ; |
| <a href="#"><u>?hptri</u></a> | to compute the inverse of $A$ .           |

## Routines for Solving Systems of Linear Equations

This section describes the LAPACK routines for solving systems of linear equations. Before calling most of these routines, you need to factorize the matrix of your system of equations (see [“Routines for Matrix Factorization”](#) in this chapter). However, the factorization is not necessary if your system of equations has a triangular matrix.

---

## ?getrs

*Solves a system of linear equations with  
an LU-factored square matrix, with  
multiple right-hand sides.*

---

```
call sgetrs (trans, n, nrhs, a, lda, ipiv, b, ldb, info)
call dgetrs (trans, n, nrhs, a, lda, ipiv, b, ldb, info)
call cgetrs (trans, n, nrhs, a, lda, ipiv, b, ldb, info)
call zgetrs (trans, n, nrhs, a, lda, ipiv, b, ldb, info)
```

### Discussion

This routine solves for  $X$  the following systems of linear equations:

$$\begin{aligned} AX = B & \quad \text{if } \text{trans} = 'N', \\ A^T X = B & \quad \text{if } \text{trans} = 'T', \\ A^H X = B & \quad \text{if } \text{trans} = 'C' \text{ (for complex matrices only).} \end{aligned}$$

Before calling this routine, you must call [?getrf](#) to compute the *LU* factorization of  $A$ .

### Input Parameters

<i>trans</i>	CHARACTER*1. Must be ' <b>N</b> ' or ' <b>T</b> ' or ' <b>C</b> '. Indicates the form of the equations: If <i>trans</i> = ' <b>N</b> ', then $AX = B$ is solved for $X$ . If <i>trans</i> = ' <b>T</b> ', then $A^T X = B$ is solved for $X$ . If <i>trans</i> = ' <b>C</b> ', then $A^H X = B$ is solved for $X$ .
<i>n</i>	INTEGER. The order of $A$ ; the number of rows in $B$ ( <i>n</i> $\geq$ 0).
<i>nrhs</i>	INTEGER. The number of right-hand sides ( <i>nrhs</i> $\geq$ 0).
<i>a</i> , <i>b</i>	REAL for <b>sgetrs</b> DOUBLE PRECISION for <b>dgetrs</b> COMPLEX for <b>cgetrs</b> DOUBLE COMPLEX for <b>zgetrs</b> . Arrays: <i>a</i> ( <i>lda</i> ,*), <i>b</i> ( <i>ldb</i> ,*).

The array  $a$  contains the matrix  $A$ .

The array  $b$  contains the matrix  $B$  whose columns are the right-hand sides for the systems of equations.

The second dimension of  $a$  must be at least  $\max(1, n)$ , the second dimension of  $b$  at least  $\max(1, nrhs)$ .

$lda$

`INTEGER`. The first dimension of  $a$ ;  $lda \geq \max(1, n)$ .

$ldb$

`INTEGER`. The first dimension of  $b$ ;  $ldb \geq \max(1, n)$ .

$ipiv$

`INTEGER`.

Array, `DIMENSION` at least  $\max(1, n)$ .

The  $ipiv$  array, as returned by [?getrf](#).

## Output Parameters

$b$

Overwritten by the solution matrix  $X$ .

$info$

`INTEGER`. If  $info=0$ , the execution is successful.

If  $info = -i$ , the  $i$ th parameter had an illegal value.

## Application Notes

For each right-hand side  $b$ , the computed solution is the exact solution of a perturbed system of equations  $(A + E)x = b$  where

$$|E| \leq c(n)\varepsilon P|L||U|$$

$c(n)$  is a modest linear function of  $n$ , and  $\varepsilon$  is the machine precision.

If  $x_0$  is the true solution, the computed solution  $x$  satisfies this error bound:

$$\frac{\|x - x_0\|_\infty}{\|x\|_\infty} \leq c(n) \operatorname{cond}(A, x) \varepsilon$$

where  $\operatorname{cond}(A, x) = \| |A^{-1}| |A| \|_\infty / \|x\|_\infty \leq \|A^{-1}\|_\infty \|A\|_\infty = \kappa_\infty(A)$ .

Note that  $\operatorname{cond}(A, x)$  can be much smaller than  $\kappa_\infty(A)$ ; the condition number of  $A^T$  and  $A^H$  might or might not be equal to  $\kappa_\infty(A)$ .

The approximate number of floating-point operations for one right-hand side vector  $b$  is  $2n^2$  for real flavors and  $8n^2$  for complex flavors.

To estimate the condition number  $\kappa_\infty(A)$ , call [?gecon](#).

To refine the solution and estimate the error, call [?gerfs](#).

## ?gbtrs

*Solves a system of linear equations with  
an LU-factored band matrix, with  
multiple right-hand sides.*

---

```
call sgbtrs (trans, n, kl, ku, nrhs, ab, ldab, ipiv, b, ldb, info)
call dgbtrs (trans, n, kl, ku, nrhs, ab, ldab, ipiv, b, ldb, info)
call cgbtrs (trans, n, kl, ku, nrhs, ab, ldab, ipiv, b, ldb, info)
call zgbtrs (trans, n, kl, ku, nrhs, ab, ldab, ipiv, b, ldb, info)
```

### Discussion

This routine solves for  $X$  the following systems of linear equations:

$$\begin{aligned} AX = B & \quad \text{if } \text{trans} = 'N', \\ A^T X = B & \quad \text{if } \text{trans} = 'T', \\ A^H X = B & \quad \text{if } \text{trans} = 'C' \text{ (for complex matrices only).} \end{aligned}$$

Here  $A$  is an  $LU$ -factored general band matrix of order  $n$  with  $kl$  non-zero sub-diagonals and  $ku$  non-zero super-diagonals. Before calling this routine, you must call [?gbtrf](#) to compute the  $LU$  factorization of  $A$ .

### Input Parameters

<i>trans</i>	CHARACTER*1. Must be ' <b>N</b> ' or ' <b>T</b> ' or ' <b>C</b> '.
<i>n</i>	INTEGER. The order of $A$ ; the number of rows in $B$ ( $n \geq 0$ ).
<i>kl</i>	INTEGER. The number of sub-diagonals within the band of $A$ ( $kl \geq 0$ ).
<i>ku</i>	INTEGER. The number of super-diagonals within the band of $A$ ( $ku \geq 0$ ).
<i>nrhs</i>	INTEGER. The number of right-hand sides ( $nrhs \geq 0$ ).
<i>ab, b</i>	REAL for <b>sgbtrs</b> DOUBLE PRECISION for <b>dgbtrs</b> COMPLEX for <b>cgbtrs</b> DOUBLE COMPLEX for <b>zgbtrs</b> . Arrays: <i>ab</i> ( <i>ldab,*</i> ), <i>b</i> ( <i>ldb,*</i> ).

The array  $\text{ab}$  contains the matrix  $A$  in *band storage* (see [Matrix Storage Schemes](#)).

The array  $\text{b}$  contains the matrix  $B$  whose columns are the right-hand sides for the systems of equations.

The second dimension of  $\text{ab}$  must be at least  $\max(1, \text{n})$ , the second dimension of  $\text{b}$  at least  $\max(1, \text{nrhs})$ .

$\text{ldab}$       INTEGER. The first dimension of the array  $\text{ab}$ .

( $\text{ldab} \geq 2\text{k}\text{l} + \text{k}\text{u} + 1$ ).

$\text{ldb}$       INTEGER. The first dimension of  $\text{b}$ ;  $\text{ldb} \geq \max(1, \text{n})$ .

$\text{ipiv}$       INTEGER. Array, DIMENSION at least  $\max(1, \text{n})$ .

The  $\text{ipiv}$  array, as returned by [?gbtrf](#).

#### Output Parameters

$\text{b}$       Overwritten by the solution matrix  $X$ .

$\text{info}$       INTEGER. If  $\text{info}=0$ , the execution is successful.

If  $\text{info} = -i$ , the  $i$ th parameter had an illegal value.

### Application Notes

For each right-hand side  $b$ , the computed solution is the exact solution of a perturbed system of equations  $(A + E)x = b$ , where

$$|E| \leq c(\text{k}\text{l} + \text{k}\text{u} + 1)\epsilon P|L||U|$$

$c(k)$  is a modest linear function of  $k$ , and  $\epsilon$  is the machine precision.

If  $x_0$  is the true solution, the computed solution  $x$  satisfies this error bound:

$$\frac{\|x - x_0\|_\infty}{\|x\|_\infty} \leq c(\text{k}\text{l} + \text{k}\text{u} + 1) \operatorname{cond}(A, x)\epsilon$$

where  $\operatorname{cond}(A, x) = \|A^{-1}\| |A| \|x\|_\infty / \|x\|_\infty \leq \|A^{-1}\|_\infty \|A\|_\infty = \kappa_\infty(A)$ .

Note that  $\operatorname{cond}(A, x)$  can be much smaller than  $\kappa_\infty(A)$ ; the condition number of  $A^T$  and  $A^H$  might or might not be equal to  $\kappa_\infty(A)$ .

The approximate number of floating-point operations for one right-hand side vector is  $2\text{n}(\text{k}\text{u} + 2\text{k}\text{l})$  for real flavors. The number of operations for complex flavors is 4 times greater. All these estimates assume that  $\text{k}\text{l}$  and  $\text{k}\text{u}$  are much less than  $\min(\text{m}, \text{n})$ .

To estimate the condition number  $\kappa_\infty(A)$ , call [?gbcon](#).

To refine the solution and estimate the error, call [?gbrfs](#).

---

## ?gttrs

*Solves a system of linear equations with  
a tridiagonal matrix using the LU  
factorization computed by ?gttrf.*

---

```
call sgttrs (trans, n, nrhs, dl, d, du, du2, ipiv, b, ldb, info)
call dgttrs (trans, n, nrhs, dl, d, du, du2, ipiv, b, ldb, info)
call cgttrs (trans, n, nrhs, dl, d, du, du2, ipiv, b, ldb, info)
call zgttrs (trans, n, nrhs, dl, d, du, du2, ipiv, b, ldb, info)
```

### Discussion

This routine solves for  $X$  the following systems of linear equations with multiple right hand sides:

$$\begin{aligned} AX = B & \quad \text{if } \text{trans} = 'N', \\ A^T X = B & \quad \text{if } \text{trans} = 'T', \\ A^H X = B & \quad \text{if } \text{trans} = 'C' \text{ (for complex matrices only).} \end{aligned}$$

Before calling this routine, you must call [?gttrf](#) to compute the *LU* factorization of  $A$ .

### Input Parameters

<i>trans</i>	CHARACTER*1. Must be ' <i>N</i> ' or ' <i>T</i> ' or ' <i>C</i> '. Indicates the form of the equations: If <i>trans</i> = ' <i>N</i> ', then $AX = B$ is solved for $X$ . If <i>trans</i> = ' <i>T</i> ', then $A^T X = B$ is solved for $X$ . If <i>trans</i> = ' <i>C</i> ', then $A^H X = B$ is solved for $X$ .
<i>n</i>	INTEGER. The order of $A$ ( <i>n</i> $\geq 0$ ).
<i>nrhs</i>	INTEGER. The number of right-hand sides, i.e., the number of columns in $B$ ( <i>nrhs</i> $\geq 0$ ).

*dl, d, du, du2, b*

REAL for *sgttrs*  
DOUBLE PRECISION for *dgttrs*  
COMPLEX for *cgttrs*  
DOUBLE COMPLEX for *zgttrs*.

Arrays:  $d1(n - 1)$ ,  $d(n)$ ,  $du(n - 1)$ ,  $du2(n - 2)$ ,  
 $b(1:ldb, nrhs)$ .

The array  $d1$  contains the  $(n - 1)$  multipliers that define the matrix  $L$  from the  $LU$  factorization of  $A$ .

The array  $d$  contains the  $n$  diagonal elements of the upper triangular matrix  $U$  from the  $LU$  factorization of  $A$ .

The array  $du$  contains the  $(n - 1)$  elements of the first super-diagonal of  $U$ .

The array  $du2$  contains the  $(n - 2)$  elements of the second super-diagonal of  $U$ .

The array  $b$  contains the matrix  $B$  whose columns are the right-hand sides for the systems of equations.

$ldb$  **INTEGER.** The leading dimension of  $b$ ;  $ldb \geq \max(1, n)$ .

$ipiv$  **INTEGER.**

Array, **DIMENSION (n).**

The  $ipiv$  array, as returned by [?gttrf](#).

## Output Parameters

$b$  Overwritten by the solution matrix  $X$ .

$info$  **INTEGER.** If  $info=0$ , the execution is successful.

If  $info = -i$ , the  $i$ th parameter had an illegal value.

## Application Notes

For each right-hand side  $b$ , the computed solution is the exact solution of a perturbed system of equations  $(A + E)x = b$  where

$$|E| \leq c(n) \varepsilon P |L| |U|$$

$c(n)$  is a modest linear function of  $n$ , and  $\varepsilon$  is the machine precision.

If  $x_0$  is the true solution, the computed solution  $x$  satisfies this error bound:

$$\frac{\|x - x_0\|_\infty}{\|x\|_\infty} \leq c(n) \operatorname{cond}(A, x) \varepsilon$$

where  $\operatorname{cond}(A, x) = \|A^{-1}\| |A| \|x\|_\infty / \|x\|_\infty \leq \|A^{-1}\|_\infty \|A\|_\infty = \kappa_\infty(A)$ .

Note that  $\operatorname{cond}(A, x)$  can be much smaller than  $\kappa_\infty(A)$ ; the condition number of  $A^T$  and  $A^H$  might or might not be equal to  $\kappa_\infty(A)$ .

The approximate number of floating-point operations for one right-hand side vector  $b$  is  $2\textcolor{red}{n}^2$  for real flavors and  $8\textcolor{red}{n}^2$  for complex flavors.

To estimate the condition number  $\kappa_\infty(A)$ , call [?gecon](#).

To refine the solution and estimate the error, call [?gerfs](#).

## ?potrs

Solves a system of linear equations with a Cholesky-factored symmetric (Hermitian) positive-definite matrix.

```
call spotrs ( uplo, n, nrhs, a, lda, b, ldb, info )
call dpotrs ( uplo, n, nrhs, a, lda, b, ldb, info )
call cpotrs ( uplo, n, nrhs, a, lda, b, ldb, info )
call zpotrs ( uplo, n, nrhs, a, lda, b, ldb, info )
```

### Discussion

This routine solves for  $X$  the system of linear equations  $AX = B$  with a symmetric positive-definite or, for complex data, Hermitian positive-definite matrix  $A$ , given the Cholesky factorization of  $A$ :

$$\begin{aligned} A &= U^H U && \text{if } \text{uplo} = 'U' \\ A &= LL^H && \text{if } \text{uplo} = 'L' \end{aligned}$$

where  $L$  is a lower triangular matrix and  $U$  is upper triangular. The system is solved with multiple right-hand sides stored in the columns of the matrix  $B$ .

Before calling this routine, you must call [?potrf](#) to compute the Cholesky factorization of  $A$ .

### Input Parameters

<i>uplo</i>	CHARACTER*1. Must be ' <i>U</i> ' or ' <i>L</i> '.
	Indicates how the input matrix $A$ has been factored:
	If <i>uplo</i> = ' <i>U</i> ', the array <i>a</i> stores the factor $U$ of the Cholesky factorization $A = U^H U$ .
	If <i>uplo</i> = ' <i>L</i> ', the array <i>a</i> stores the factor $L$ of the Cholesky factorization $A = LL^H$ .
<i>n</i>	INTEGER. The order of matrix $A$ ( <i>n</i> $\geq 0$ ).
<i>nrhs</i>	INTEGER. The number of right-hand sides ( <i>nrhs</i> $\geq 0$ ).

<i>a, b</i>	<small>REAL for <b>spotrs</b> DOUBLE PRECISION for <b>dpotrs</b> COMPLEX for <b>cpotrs</b> DOUBLE COMPLEX for <b>zpotrs</b>.</small> Arrays: <i>a( lda, * ), b( ldb, * )</i> . The array <i>a</i> contains the factor <i>U</i> or <i>L</i> (see <i>uplo</i> ). The array <i>b</i> contains the matrix <i>B</i> whose columns are the right-hand sides for the systems of equations. The second dimension of <i>a</i> must be at least $\max(1, n)$ , the second dimension of <i>b</i> at least $\max(1, nrhs)$ .
<i>lda</i>	<small>INTEGER. The first dimension of <i>a</i>; <i>lda</i> <math>\geq \max(1, n)</math>.</small>
<i>ldb</i>	<small>INTEGER. The first dimension of <i>b</i>; <i>ldb</i> <math>\geq \max(1, n)</math>.</small>

### Output Parameters

<i>b</i>	Overwritten by the solution matrix <i>X</i> .
<i>info</i>	<small>INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = <i>-i</i>, the <i>i</i>th parameter had an illegal value.</small>

### Application Notes

If *uplo* = 'U', the computed solution for each right-hand side *b* is the exact solution of a perturbed system of equations  $(A + E)x = b$ , where

$$|E| \leq c(n)\varepsilon|U^H||U|$$

*c(n)* is a modest linear function of *n*, and  $\varepsilon$  is the machine precision.

A similar estimate holds for *uplo* = 'L'.

If  $x_0$  is the true solution, the computed solution *x* satisfies this error bound:

$$\frac{\|x - x_0\|_\infty}{\|x\|_\infty} \leq c(n) \operatorname{cond}(A, x)\varepsilon$$

where  $\operatorname{cond}(A, x) = \| |A^{-1}| |A| |x| \|_\infty / \|x\|_\infty \leq \|A^{-1}\|_\infty \|A\|_\infty = \kappa_\infty(A)$ .

Note that  $\operatorname{cond}(A, x)$  can be much smaller than  $\kappa_\infty(A)$ .

The approximate number of floating-point operations for one right-hand side vector *b* is  $2n^2$  for real flavors and  $8n^2$  for complex flavors.

To estimate the condition number  $\kappa_\infty(A)$ , call [?pocon](#).

To refine the solution and estimate the error, call [?porfs](#).

## ?pptrs

Solves a system of linear equations with a packed Cholesky-factored symmetric (Hermitian) positive-definite matrix.

```
call spptrs ( uplo, n, nrhs, ap, b, ldb, info )
call dpptrs ( uplo, n, nrhs, ap, b, ldb, info )
call cpptrs ( uplo, n, nrhs, ap, b, ldb, info )
call zpptrs ( uplo, n, nrhs, ap, b, ldb, info )
```

### Discussion

This routine solves for  $X$  the system of linear equations  $AX = B$  with a packed symmetric positive-definite or, for complex data, Hermitian positive-definite matrix  $A$ , given the Cholesky factorization of  $A$ :

$$\begin{aligned} A &= U^H U && \text{if } \text{uplo} = 'U' \\ A &= LL^H && \text{if } \text{uplo} = 'L' \end{aligned}$$

where  $L$  is a lower triangular matrix and  $U$  is upper triangular. The system is solved with multiple right-hand sides stored in the columns of the matrix  $B$ .

Before calling this routine, you must call [?pptrf](#) to compute the Cholesky factorization of  $A$ .

### Input Parameters

<i>uplo</i>	CHARACTER*1. Must be ' <i>U</i> ' or ' <i>L</i> '.
	Indicates how the input matrix $A$ has been factored:
	If <i>uplo</i> = ' <i>U</i> ', the array <i>a</i> stores the packed factor $U$ of the Cholesky factorization $A = U^H U$ .
	If <i>uplo</i> = ' <i>L</i> ', the array <i>a</i> stores the packed factor $L$ of the Cholesky factorization $A = LL^H$ .
<i>n</i>	INTEGER. The order of matrix $A$ ( <i>n</i> $\geq 0$ ).
<i>nrhs</i>	INTEGER. The number of right-hand sides ( <i>nrhs</i> $\geq 0$ ).

<i>ap, b</i>	<small>REAL for <code>spptrs</code> DOUBLE PRECISION for <code>dpptrs</code> COMPLEX for <code>cptrs</code> DOUBLE COMPLEX for <code>zptrs</code>.</small>
	<small>Arrays: <i>ap</i>( * ), <i>b</i>( <i>ldb</i>, * ) The dimension of <i>ap</i> must be at least <math>\max(1, \textcolor{red}{n}(\textcolor{red}{n}+1)/2)</math>. The array <i>ap</i> contains the factor <i>U</i> or <i>L</i>, as specified by <i>uplo</i>, in <i>packed storage</i> (see <a href="#">Matrix Storage Schemes</a>). The array <i>b</i> contains the matrix <i>B</i> whose columns are the right-hand sides for the systems of equations. The second dimension of <i>b</i> must be at least <math>\max(1, \textcolor{red}{nrhs})</math>.</small>
<i>ldb</i>	<small>INTEGER. The first dimension of <i>b</i>; <math>\textcolor{red}{ldb} \geq \max(1, n)</math>.</small>

## Output Parameters

<i>b</i>	Overwritten by the solution matrix <i>X</i> .
<i>info</i>	<small>INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = <math>-\textcolor{red}{i}</math>, the <i>i</i>th parameter had an illegal value.</small>

## Application Notes

If *uplo* = 'U', the computed solution for each right-hand side *b* is the exact solution of a perturbed system of equations  $(A + E)x = b$ , where

$$|E| \leq c(n)\varepsilon|U^H||U|$$

$c(n)$  is a modest linear function of *n*, and  $\varepsilon$  is the machine precision.

A similar estimate holds for *uplo* = 'L'.

If  $x_0$  is the true solution, the computed solution  $x$  satisfies this error bound:

$$\frac{\|x - x_0\|_\infty}{\|x\|_\infty} \leq c(n) \operatorname{cond}(A, x)\varepsilon$$

where  $\operatorname{cond}(A, x) = \| |A^{-1}| |A| |x| \|_\infty / \|x\|_\infty \leq \|A^{-1}\|_\infty \|A\|_\infty = \kappa_\infty(A)$ .

Note that  $\operatorname{cond}(A, x)$  can be much smaller than  $\kappa_\infty(A)$ .

The approximate number of floating-point operations for one right-hand side vector *b* is  $2n^2$  for real flavors and  $8n^2$  for complex flavors.

To estimate the condition number  $\kappa_\infty(A)$ , call [?ppcon](#).

To refine the solution and estimate the error, call [?pprfs](#).

## ?pbtrs

Solves a system of linear equations with a Cholesky-factored symmetric (Hermitian) positive-definite band matrix.

```
call spbtrs (uplo, n, kd, nrhs, ab, ldab, b, ldb, info)
call dpbtrs (uplo, n, kd, nrhs, ab, ldab, b, ldb, info)
call cpbtrs (uplo, n, kd, nrhs, ab, ldab, b, ldb, info)
call zpbtrs (uplo, n, kd, nrhs, ab, ldab, b, ldb, info)
```

### Discussion

This routine solves for  $X$  the system of linear equations  $AX = B$  with a symmetric positive-definite or, for complex data, Hermitian positive-definite **band** matrix  $A$ , given the Cholesky factorization of  $A$ :

$$\begin{aligned} A &= U^H U && \text{if } \text{uplo} = 'U' \\ A &= LL^H && \text{if } \text{uplo} = 'L' \end{aligned}$$

where  $L$  is a lower triangular matrix and  $U$  is upper triangular. The system is solved with multiple right-hand sides stored in the columns of the matrix  $B$ .

Before calling this routine, you must call [?pbtrf](#) to compute the Cholesky factorization of  $A$  in the band storage form.

### Input Parameters

<i>uplo</i>	CHARACTER*1. Must be ' <b>U</b> ' or ' <b>L</b> '.
	Indicates how the input matrix $A$ has been factored:
	If <i>uplo</i> = ' <b>U</b> ', the array <i>a</i> stores the factor $U$ of the factorization $A = U^H U$ in the band storage form.
	If <i>uplo</i> = ' <b>L</b> ', the array <i>a</i> stores the factor $L$ of the factorization $A = LL^H$ in the band storage form.
<i>n</i>	INTEGER. The order of matrix $A$ ( <i>n</i> $\geq 0$ ).
<i>kd</i>	INTEGER. The number of super-diagonals or sub-diagonals in the matrix $A$ ( <i>kd</i> $\geq 0$ ).
<i>nrhs</i>	INTEGER. The number of right-hand sides ( <i>nrhs</i> $\geq 0$ ).

<i>ab, b</i>	REAL for <code>spbtrs</code> DOUBLE PRECISION for <code>dpbtrs</code> COMPLEX for <code>cpbtrs</code> DOUBLE COMPLEX for <code>zpbtrs</code> . Arrays: <i>ab( l<sub>dab</sub>, * ), b( l<sub>db</sub>, * )</i> .
	The array <i>ab</i> contains the Cholesky factor, as returned by the factorization routine, in <i>band storage</i> form.
	The array <i>b</i> contains the matrix <i>B</i> whose columns are the right-hand sides for the systems of equations.
	The second dimension of <i>ab</i> must be at least $\max(1, n)$ , the second dimension of <i>b</i> at least $\max(1, nrhs)$ .
<i>l<sub>dab</sub></i>	INTEGER. The first dimension of the array <i>ab</i> . ( $l_{dab} \geq kd + 1$ ).
<i>l<sub>db</sub></i>	INTEGER. The first dimension of <i>b</i> ; $l_{db} \geq \max(1, n)$ .

### Output Parameters

<i>b</i>	Overwritten by the solution matrix <i>X</i> .
<i>info</i>	INTEGER. If <i>info</i> =0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> th parameter had an illegal value.

### Application Notes

For each right-hand side *b*, the computed solution is the exact solution of a perturbed system of equations  $(A + E)x = b$ , where

$$|E| \leq c(kd + 1)\varepsilon P|U^H||U| \text{ or } |E| \leq c(kd + 1)\varepsilon P|L^H||L|$$

*c(k)* is a modest linear function of *k*, and  $\varepsilon$  is the machine precision.

If  $x_0$  is the true solution, the computed solution *x* satisfies this error bound:

$$\frac{\|x - x_0\|_\infty}{\|x\|_\infty} \leq c(kd + 1) \operatorname{cond}(A, x)\varepsilon$$

where  $\operatorname{cond}(A, x) = \|A^{-1}\| |A| \|x\|_\infty / \|x\|_\infty \leq \|A^{-1}\|_\infty \|A\|_\infty = \kappa_\infty(A)$ .

Note that  $\operatorname{cond}(A, x)$  can be much smaller than  $\kappa_\infty(A)$ .

The approximate number of floating-point operations for one right-hand side vector is  $4n^2kd$  for real flavors and  $16n^2kd$  for complex flavors.

To estimate the condition number  $\kappa_\infty(A)$ , call [?pbcon](#).

To refine the solution and estimate the error, call [?pbrfs](#).

## ?pttrs

Solves a system of linear equations with a symmetric (Hermitian) positive-definite tridiagonal matrix using the factorization computed by ?pttrf.

```
call spttrs (n, nrhs, d, e, b, ldb, info)
call dpttrs (n, nrhs, d, e, b, ldb, info)
call cpttrs (uplo, n, nrhs, d, e, b, ldb, info)
call zpttrs (uplo, n, nrhs, d, e, b, ldb, info)
```

### Discussion

This routine solves for  $X$  a system of linear equations  $AX = B$  with a symmetric (Hermitian) positive-definite tridiagonal matrix  $A$ .

Before calling this routine, you must call [?pttrf](#) to compute the  $LDL^H$  or  $U^HDU$  factorization of  $A$ .

### Input Parameters

<i>uplo</i>	CHARACTER*1. Used for <code>cpttrs/zpttrs</code> only. Must be ' <code>U</code> ' or ' <code>L</code> '. Specifies whether the superdiagonal or the subdiagonal of the tridiagonal matrix $A$ is stored and how $A$ is factored: If <i>uplo</i> = ' <code>U</code> ', the array <code>e</code> stores the superdiagonal of $A$ , and $A$ is factored as $U^HDU$ ; If <i>uplo</i> = ' <code>L</code> ', the array <code>e</code> stores the subdiagonal of $A$ , and $A$ is factored as $LDL^H$ .
<i>n</i>	INTEGER. The order of $A$ ( <i>n</i> $\geq 0$ ).
<i>nrhs</i>	INTEGER. The number of right-hand sides, i.e., the number of columns of the matrix $B$ ( <i>nrhs</i> $\geq 0$ ).

<i>d</i>	REAL for <code>spttrs</code> , <code>cpttrs</code> DOUBLE PRECISION for <code>dpttrs</code> , <code>zpttrs</code> . Array, dimension ( <i>n</i> ). Contains the diagonal elements of the diagonal matrix <i>D</i> from the factorization computed by <a href="#">?pttrf</a> .
<i>e</i> , <i>b</i>	REAL for <code>spttrs</code> DOUBLE PRECISION for <code>dpttrs</code> COMPLEX for <code>cpttrs</code> DOUBLE COMPLEX for <code>zpttrs</code> . Arrays: <i>e</i> ( <i>n</i> - 1), <i>b</i> ( <i>ldb</i> , <i>nrhs</i> ). The array <i>e</i> contains the ( <i>n</i> - 1) off-diagonal elements of the unit bidiagonal factor <i>U</i> or <i>L</i> from the factorization computed by <a href="#">?pttrf</a> (see <i>uplo</i> ). The array <i>b</i> contains the matrix <i>B</i> whose columns are the right-hand sides for the systems of equations.
<i>ldb</i>	INTEGER. The leading dimension of <i>b</i> ; <i>ldb</i> ≥ max(1, <i>n</i> ).

### Output Parameters

<i>b</i>	Overwritten by the solution matrix <i>X</i> .
<i>info</i>	INTEGER. If <i>info</i> =0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> th parameter had an illegal value.

## ?sytrs

Solves a system of linear equations with a UDU- or LDL-factored symmetric matrix.

```
call ssytrs (uplo, n, nrhs, a, lda, ipiv, b, ldb, info)
call dsytrs (uplo, n, nrhs, a, lda, ipiv, b, ldb, info)
call csytrs (uplo, n, nrhs, a, lda, ipiv, b, ldb, info)
call zsytrs (uplo, n, nrhs, a, lda, ipiv, b, ldb, info)
```

### Discussion

This routine solves for  $X$  the system of linear equations  $AX = B$  with a symmetric matrix  $A$ , given the Bunch-Kaufman factorization of  $A$ :

$$\begin{aligned} \text{if } uplo = 'U', \quad A &= PUDU^T P^T \\ \text{if } uplo = 'L', \quad A &= PLDL^T P^T \end{aligned}$$

where  $P$  is a permutation matrix,  $U$  and  $L$  are upper and lower triangular matrices with unit diagonal, and  $D$  is a symmetric block-diagonal matrix. The system is solved with multiple right-hand sides stored in the columns of the matrix  $B$ . You must supply to this routine the factor  $U$  (or  $L$ ) and the array  $ipiv$  returned by the factorization routine [?sytrf](#).

### Input Parameters

<i>uplo</i>	CHARACTER*1. Must be ' <i>U</i> ' or ' <i>L</i> '.
	Indicates how the input matrix $A$ has been factored:
If <i>uplo</i> = ' <i>U</i> ',	the array <i>a</i> stores the upper triangular factor $U$ of the factorization $A = PUDU^T P^T$ .
If <i>uplo</i> = ' <i>L</i> ',	the array <i>a</i> stores the lower triangular factor $L$ of the factorization $A = PLDL^T P^T$ .
<i>n</i>	INTEGER. The order of matrix $A$ ( <i>n</i> $\geq 0$ ).
<i>nrhs</i>	INTEGER. The number of right-hand sides ( <i>nrhs</i> $\geq 0$ ).
<i>ipiv</i>	INTEGER. Array, DIMENSION at least $\max(1, n)$ . The <i>ipiv</i> array, as returned by <a href="#">?sytrf</a> .

<i>a</i> , <i>b</i>	<small>REAL for ssytrs DOUBLE PRECISION for dsytrs COMPLEX for csytrs DOUBLE COMPLEX for zsytrs.</small> Arrays: <i>a</i> ( <i>lda</i> , *), <i>b</i> ( <i>ldb</i> , *). The array <i>a</i> contains the factor <i>U</i> or <i>L</i> (see <i>uplo</i> ). The array <i>b</i> contains the matrix <i>B</i> whose columns are the right-hand sides for the system of equations. The second dimension of <i>a</i> must be at least max(1, <i>n</i> ), the second dimension of <i>b</i> at least max(1, <i>nrhs</i> ).
<i>lda</i>	<small>INTEGER. The first dimension of <i>a</i>; <i>lda</i> ≥ max(1, <i>n</i>).</small>
<i>ldb</i>	<small>INTEGER. The first dimension of <i>b</i>; <i>ldb</i> ≥ max(1, <i>n</i>).</small>

### Output Parameters

<i>b</i>	Overwritten by the solution matrix <i>X</i> .
<i>info</i>	<small>INTEGER. If <i>info</i>=0, the execution is successful. If <i>info</i> = -<i>i</i>, the <i>i</i>th parameter had an illegal value.</small>

### Application Notes

For each right-hand side *b*, the computed solution is the exact solution of a perturbed system of equations  $(A + E)x = b$ , where

$$|E| \leq c(\textcolor{red}{n}) \varepsilon P |U| |D| |U^T| P^T \quad \text{or} \quad |E| \leq c(\textcolor{red}{n}) \varepsilon P |L| |D| |L^T| P^T$$

$c(\textcolor{red}{n})$  is a modest linear function of *n*, and  $\varepsilon$  is the machine precision.

If  $x_0$  is the true solution, the computed solution  $x$  satisfies this error bound:

$$\frac{\|x - x_0\|_\infty}{\|x\|_\infty} \leq c(\textcolor{red}{n}) \operatorname{cond}(A, x) \varepsilon$$

where  $\operatorname{cond}(A, x) = \|A^{-1}\| |A| \|x\|_\infty / \|x\|_\infty \leq \|A^{-1}\|_\infty \|A\|_\infty = \kappa_\infty(A)$ .

Note that  $\operatorname{cond}(A, x)$  can be much smaller than  $\kappa_\infty(A)$ .

The total number of floating-point operations for one right-hand side vector is approximately  $2\textcolor{red}{n}^2$  for real flavors or  $8\textcolor{red}{n}^2$  for complex flavors.

To estimate the condition number  $\kappa_\infty(A)$ , call [?sycon](#).

To refine the solution and estimate the error, call [?syref](#).

## ?hetrs

Solves a system of linear equations with a UDU- or LDL-factored Hermitian matrix.

```
call chetrs (uplo, n, nrhs, a, lda, ipiv, b, ldb, info)
call zhetrs (uplo, n, nrhs, a, lda, ipiv, b, ldb, info)
```

### Discussion

This routine solves for  $X$  the system of linear equations  $AX = B$  with a Hermitian matrix  $A$ , given the Bunch-Kaufman factorization of  $A$ :

$$\begin{aligned} \text{if } \text{uplo} = 'U', \quad A &= PUDU^H P^T \\ \text{if } \text{uplo} = 'L', \quad A &= PLDL^H P^T \end{aligned}$$

where  $P$  is a permutation matrix,  $U$  and  $L$  are upper and lower triangular matrices with unit diagonal, and  $D$  is a symmetric block-diagonal matrix. The system is solved with multiple right-hand sides stored in the columns of the matrix  $B$ . You must supply to this routine the factor  $U$  (or  $L$ ) and the array  $\text{ipiv}$  returned by the factorization routine [?hetrf](#).

### Input Parameters

<i>uplo</i>	CHARACTER*1. Must be ' <i>U</i> ' or ' <i>L</i> '.
	Indicates how the input matrix $A$ has been factored:
If <i>uplo</i> = ' <i>U</i> ',	the array <i>a</i> stores the upper triangular factor $U$ of the factorization $A = PUDU^H P^T$ .
If <i>uplo</i> = ' <i>L</i> ',	the array <i>a</i> stores the lower triangular factor $L$ of the factorization $A = PLDL^H P^T$ .
<i>n</i>	INTEGER. The order of matrix $A$ ( <i>n</i> $\geq 0$ ).
<i>nrhs</i>	INTEGER. The number of right-hand sides ( <i>nrhs</i> $\geq 0$ ).
<i>ipiv</i>	INTEGER. Array, DIMENSION at least max(1, <i>n</i> ). The <i>ipiv</i> array, as returned by <a href="#">?hetrf</a> .

<i>a</i> , <i>b</i>	COMPLEX for <code>chetrs</code> . DOUBLE COMPLEX for <code>zhetrs</code> . Arrays: <i>a</i> ( <i>lda</i> , *), <i>b</i> ( <i>ldb</i> , *). The array <i>a</i> contains the factor <i>U</i> or <i>L</i> (see <code>uplo</code> ). The array <i>b</i> contains the matrix <i>B</i> whose columns are the right-hand sides for the system of equations. The second dimension of <i>a</i> must be at least $\max(1, n)$ , the second dimension of <i>b</i> at least $\max(1, nrhs)$ .
<i>lda</i>	INTEGER. The first dimension of <i>a</i> ; <i>lda</i> $\geq \max(1, n)$ .
<i>ldb</i>	INTEGER. The first dimension of <i>b</i> ; <i>ldb</i> $\geq \max(1, n)$ .

## Output Parameters

<i>b</i>	Overwritten by the solution matrix <i>X</i> .
<i>info</i>	INTEGER. If <i>info</i> =0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> th parameter had an illegal value.

## Application Notes

For each right-hand side *b*, the computed solution is the exact solution of a perturbed system of equations  $(A + E)x = b$ , where

$$|E| \leq c(n)\varepsilon P|U||D|\|U^H|P^T \text{ or } |E| \leq c(n)\varepsilon P|L||D|\|L^H|P^T$$

*c(n)* is a modest linear function of *n*, and  $\varepsilon$  is the machine precision.

If  $x_0$  is the true solution, the computed solution *x* satisfies this error bound:

$$\frac{\|x - x_0\|_\infty}{\|x\|_\infty} \leq c(n) \operatorname{cond}(A, x)\varepsilon$$

where  $\operatorname{cond}(A, x) = \| |A^{-1}| |A| |x| \|_\infty / \|x\|_\infty \leq \|A^{-1}\|_\infty \|A\|_\infty = \kappa_\infty(A)$ .

Note that  $\operatorname{cond}(A, x)$  can be much smaller than  $\kappa_\infty(A)$ .

The total number of floating-point operations for one right-hand side vector is approximately  $8n^2$ .

To estimate the condition number  $\kappa_\infty(A)$ , call [?hecon](#).

To refine the solution and estimate the error, call [?herfs](#).

## ?sptrs

Solves a system of linear equations with a UDU- or LDL-factored symmetric matrix using packed storage.

```
call ssptrs ( uplo, n, nrhs, ap, ipiv, b, ldb, info )
call dsptrs ( uplo, n, nrhs, ap, ipiv, b, ldb, info )
call csptrs ( uplo, n, nrhs, ap, ipiv, b, ldb, info )
call zsptrs ( uplo, n, nrhs, ap, ipiv, b, ldb, info )
```

### Discussion

This routine solves for  $X$  the system of linear equations  $AX = B$  with a symmetric matrix  $A$ , given the Bunch-Kaufman factorization of  $A$ :

$$\begin{aligned} \text{if } \text{uplo} = 'U', \quad A &= PUDU^T P^T \\ \text{if } \text{uplo} = 'L', \quad A &= PLDL^T P^T \end{aligned}$$

where  $P$  is a permutation matrix,  $U$  and  $L$  are upper and lower **packed** triangular matrices with unit diagonal, and  $D$  is a symmetric block-diagonal matrix. The system is solved with multiple right-hand sides stored in the columns of the matrix  $B$ . You must supply the factor  $U$  (or  $L$ ) and the array  $ipiv$  returned by the factorization routine [?sptrf](#).

### Input Parameters

<i>uplo</i>	CHARACTER*1. Must be ' <b>U</b> ' or ' <b>L</b> '.
	Indicates how the input matrix $A$ has been factored:
	If <i>uplo</i> = ' <b>U</b> ', the array <i>ap</i> stores the packed factor $U$ of the factorization $A = PUDU^T P^T$ .
	If <i>uplo</i> = ' <b>L</b> ', the array <i>ap</i> stores the packed factor $L$ of the factorization $A = PLDL^T P^T$ .
<i>n</i>	INTEGER. The order of matrix $A$ ( <i>n</i> $\geq 0$ ).
<i>nrhs</i>	INTEGER. The number of right-hand sides ( <i>nrhs</i> $\geq 0$ ).
<i>ipiv</i>	INTEGER. Array, DIMENSION at least $\max(1, n)$ . The <i>ipiv</i> array, as returned by <a href="#">?sptrf</a> .

<i>ap, b</i>	<small>REAL for ssptrs DOUBLE PRECISION for dsptrs COMPLEX for csptrs DOUBLE COMPLEX for zsptrs.</small> Arrays: <i>ap</i> (*), <i>b</i> ( <i>ldb</i> ,*)
<i>ldb</i>	The dimension of <i>ap</i> must be at least $\max(1, \frac{n(n+1)}{2})$ . The array <i>ap</i> contains the factor <i>U</i> or <i>L</i> , as specified by <i>uplo</i> , in <i>packed storage</i> (see <a href="#">Matrix Storage Schemes</a> ). The array <i>b</i> contains the matrix <i>B</i> whose columns are the right-hand sides for the system of equations. The second dimension of <i>b</i> must be at least $\max(1, nrhs)$ .
	<small>INTEGER. The first dimension of <i>b</i>; <i>ldb</i> <math>\geq \max(1, n)</math>.</small>

## Output Parameters

<i>b</i>	Overwritten by the solution matrix <i>X</i> .
<i>info</i>	<small>INTEGER. If <i>info</i>=0, the execution is successful. If <i>info</i> = -<i>i</i>, the <i>i</i>th parameter had an illegal value.</small>

## Application Notes

For each right-hand side *b*, the computed solution is the exact solution of a perturbed system of equations  $(A + E)x = b$ , where

$$|E| \leq c(\textcolor{red}{n}) \varepsilon P |U| |D| |U^T| P^T \quad \text{or} \quad |E| \leq c(\textcolor{red}{n}) \varepsilon P |L| |D| |L^T| P^T$$

$c(\textcolor{red}{n})$  is a modest linear function of  $n$ , and  $\varepsilon$  is the machine precision.

If  $x_0$  is the true solution, the computed solution  $x$  satisfies this error bound:

$$\frac{\|x - x_0\|_\infty}{\|x\|_\infty} \leq c(\textcolor{red}{n}) \operatorname{cond}(A, x) \varepsilon$$

where  $\operatorname{cond}(A, x) = \| |A^{-1}| |A| \|_1 \|x\|_\infty / \|x\|_\infty \leq \|A^{-1}\|_\infty \|A\|_\infty = \kappa_\infty(A)$ .

Note that  $\operatorname{cond}(A, x)$  can be much smaller than  $\kappa_\infty(A)$ .

The total number of floating-point operations for one right-hand side vector is approximately  $2\textcolor{red}{n}^2$  for real flavors or  $8\textcolor{red}{n}^2$  for complex flavors.

To estimate the condition number  $\kappa_\infty(A)$ , call [?spcon](#).

To refine the solution and estimate the error, call [?sprfs](#).

## ?hptrs

Solves a system of linear equations with a UDU- or LDL-factored Hermitian matrix using packed storage.

```
call chptrs ( uplo, n, nrhs, ap, ipiv, b, ldb, info )
call zhptrs ( uplo, n, nrhs, ap, ipiv, b, ldb, info )
```

### Discussion

This routine solves for  $X$  the system of linear equations  $AX = B$  with a Hermitian matrix  $A$ , given the Bunch-Kaufman factorization of  $A$ :

$$\begin{aligned} \text{if } \text{uplo} = 'U', \quad A &= PUDU^H P^T \\ \text{if } \text{uplo} = 'L', \quad A &= PLDL^H P^T \end{aligned}$$

where  $P$  is a permutation matrix,  $U$  and  $L$  are upper and lower **packed** triangular matrices with unit diagonal, and  $D$  is a symmetric block-diagonal matrix. The system is solved with multiple right-hand sides stored in the columns of the matrix  $B$ .

You must supply to this routine the arrays `ap` (containing  $U$  or  $L$ ) and `ipiv` in the form returned by the factorization routine [?hptrf](#).

### Input Parameters

<code>uplo</code>	<b>CHARACTER*1</b> . Must be ' <code>U</code> ' or ' <code>L</code> '.
	Indicates how the input matrix $A$ has been factored:
	If <code>uplo</code> = ' <code>U</code> ', the array <code>ap</code> stores the packed factor $U$ of the factorization $A = PUDU^H P^T$ .
	If <code>uplo</code> = ' <code>L</code> ', the array <code>ap</code> stores the packed factor $L$ of the factorization $A = PLDL^H P^T$ .
<code>n</code>	<b>INTEGER</b> . The order of matrix $A$ ( $n \geq 0$ ).
<code>nrhs</code>	<b>INTEGER</b> . The number of right-hand sides ( $nrhs \geq 0$ ).
<code>ipiv</code>	<b>INTEGER</b> . Array, <b>DIMENSION</b> at least $\max(1, n)$ . The <code>ipiv</code> array, as returned by <a href="#">?hptrf</a> .

<i>ap</i> , <i>b</i>	<i>COMPLEX</i> for <a href="#">chptrs</a> . <i>DOUBLE COMPLEX</i> for <a href="#">zhptrs</a> .
	Arrays: <i>ap</i> (*), <i>b</i> ( <i>ldb</i> , *) The dimension of <i>ap</i> must be at least $\max(1, \frac{n(n+1)}{2})$ . The array <i>ap</i> contains the factor <i>U</i> or <i>L</i> , as specified by <i>uplo</i> , in <i>packed storage</i> (see <a href="#">Matrix Storage Schemes</a> ). The array <i>b</i> contains the matrix <i>B</i> whose columns are the right-hand sides for the system of equations. The second dimension of <i>b</i> must be at least $\max(1, \frac{nrhs}{2})$ .
<i>ldb</i>	<i>INTEGER</i> . The first dimension of <i>b</i> ; <i>ldb</i> $\geq \max(1, n)$ .

## Output Parameters

<i>b</i>	Overwritten by the solution matrix <i>X</i> .
<i>info</i>	<i>INTEGER</i> . If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> th parameter had an illegal value.

## Application Notes

For each right-hand side *b*, the computed solution is the exact solution of a perturbed system of equations  $(A + E)x = b$ , where

$$|E| \leq c(n)\varepsilon P|U||D|\|U^H\|P^T \quad \text{or} \quad |E| \leq c(n)\varepsilon P|L||D|\|L^H\|P^T$$

*c(n)* is a modest linear function of *n*, and  $\varepsilon$  is the machine precision.

If  $x_0$  is the true solution, the computed solution *x* satisfies this error bound:

$$\frac{\|x - x_0\|_\infty}{\|x\|_\infty} \leq c(n) \operatorname{cond}(A, x)\varepsilon$$

where  $\operatorname{cond}(A, x) = \|A^{-1}\| |A| \|x\|_\infty / \|x\|_\infty \leq \|A^{-1}\|_\infty \|A\|_\infty = \kappa_\infty(A)$ .

Note that  $\operatorname{cond}(A, x)$  can be much smaller than  $\kappa_\infty(A)$ .

The total number of floating-point operations for one right-hand side vector is approximately  $8n^2$  for complex flavors.

To estimate the condition number  $\kappa_\infty(A)$ , call [?hpcon](#).

To refine the solution and estimate the error, call [?hprfs](#).

## ?trtrs

Solves a system of linear equations with a triangular matrix, with multiple right-hand sides.

```
call strtrs (uplo,trans,diag,n,nrhs,a,lda,b,ldb,info)
call dtrtrs (uplo,trans,diag,n,nrhs,a,lda,b,ldb,info)
call ctrtrs (uplo,trans,diag,n,nrhs,a,lda,b,ldb,info)
call ztrtrs (uplo,trans,diag,n,nrhs,a,lda,b,ldb,info)
```

### Discussion

This routine solves for  $X$  the following systems of linear equations with a triangular matrix  $A$ , with multiple right-hand sides stored in  $B$ :

$$\begin{aligned} AX = B & \quad \text{if } \text{trans} = 'N', \\ A^T X = B & \quad \text{if } \text{trans} = 'T', \\ A^H X = B & \quad \text{if } \text{trans} = 'C' \text{ (for complex matrices only).} \end{aligned}$$

### Input Parameters

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. Indicates whether $A$ is upper or lower triangular: If <i>uplo</i> = 'U', then $A$ is upper triangular. If <i>uplo</i> = 'L', then $A$ is lower triangular.
<i>trans</i>	CHARACTER*1. Must be 'N' or 'T' or 'C'. If <i>trans</i> = 'N', then $AX = B$ is solved for $X$ . If <i>trans</i> = 'T', then $A^T X = B$ is solved for $X$ . If <i>trans</i> = 'C', then $A^H X = B$ is solved for $X$ .
<i>diag</i>	CHARACTER*1. Must be 'N' or 'U'. If <i>diag</i> = 'N', then $A$ is not a unit triangular matrix. If <i>diag</i> = 'U', then $A$ is unit triangular: diagonal elements of $A$ are assumed to be 1 and not referenced in the array <i>a</i> .
<i>n</i>	INTEGER. The order of $A$ ; the number of rows in $B$ ( $n \geq 0$ ).
<i>nrhs</i>	INTEGER. The number of right-hand sides ( $nrhs \geq 0$ ).

<i>a, b</i>	REAL for <code>strtrs</code> DOUBLE PRECISION for <code>dtrtrs</code> COMPLEX for <code>ctrtrs</code> DOUBLE COMPLEX for <code>ztrtrs</code> . Arrays: <i>a( lda, * ), b( ldb, * )</i> .
	The array <i>a</i> contains the matrix <i>A</i> . The array <i>b</i> contains the matrix <i>B</i> whose columns are the right-hand sides for the systems of equations.
	The second dimension of <i>a</i> must be at least $\max(1, n)$ , the second dimension of <i>b</i> at least $\max(1, nrhs)$ .
<i>lda</i>	INTEGER. The first dimension of <i>a</i> ; <i>lda</i> $\geq \max(1, n)$ .
<i>ldb</i>	INTEGER. The first dimension of <i>b</i> ; <i>ldb</i> $\geq \max(1, n)$ .

### Output Parameters

<i>b</i>	Overwritten by the solution matrix <i>X</i> .
<i>info</i>	INTEGER. If <i>info</i> =0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> th parameter had an illegal value.

### Application Notes

For each right-hand side *b*, the computed solution is the exact solution of a perturbed system of equations  $(A + E)x = b$  where

$$|E| \leq c(n)\varepsilon|A|$$

$c(n)$  is a modest linear function of *n*, and  $\varepsilon$  is the machine precision.  
If  $x_0$  is the true solution, the computed solution *x* satisfies this error bound:  

$$\frac{\|x - x_0\|_\infty}{\|x\|_\infty} \leq c(n) \operatorname{cond}(A, x)\varepsilon, \text{ provided } c(n) \operatorname{cond}(A, x)\varepsilon < 1$$

where  $\operatorname{cond}(A, x) = \| |A^{-1}| |A| |x| \|_\infty / \|x\|_\infty \leq \|A^{-1}\|_\infty \|A\|_\infty = \kappa_\infty(A)$ .

Note that  $\operatorname{cond}(A, x)$  can be much smaller than  $\kappa_\infty(A)$ ; the condition number of  $A^T$  and  $A^H$  might or might not be equal to  $\kappa_\infty(A)$ .

The approximate number of floating-point operations for one right-hand side vector *b* is  $n^2$  for real flavors and  $4n^2$  for complex flavors.

To estimate the condition number  $\kappa_\infty(A)$ , call [?trcon](#).

To estimate the error in the solution, call [?trrfs](#).

## ?tptrs

Solves a system of linear equations with a packed triangular matrix, with multiple right-hand sides.

```
call stptrs (uplo,trans,diag,n,nrhs,ap,b,ldb,info)
call dptrs (uplo,trans,diag,n,nrhs,ap,b,ldb,info)
call cptrs (uplo,trans,diag,n,nrhs,ap,b,ldb,info)
call zptrs (uplo,trans,diag,n,nrhs,ap,b,ldb,info)
```

### Discussion

This routine solves for  $X$  the following systems of linear equations with a packed triangular matrix  $A$ , with multiple right-hand sides stored in  $B$ :

$$\begin{array}{ll} AX = B & \text{if } \text{trans} = 'N', \\ A^T X = B & \text{if } \text{trans} = 'T', \\ A^H X = B & \text{if } \text{trans} = 'C' \text{ (for complex matrices only).} \end{array}$$

### Input Parameters

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. Indicates whether $A$ is upper or lower triangular: If <i>uplo</i> = 'U', then $A$ is upper triangular. If <i>uplo</i> = 'L', then $A$ is lower triangular.
<i>trans</i>	CHARACTER*1. Must be 'N' or 'T' or 'C'. If <i>trans</i> = 'N', then $AX = B$ is solved for $X$ . If <i>trans</i> = 'T', then $A^T X = B$ is solved for $X$ . If <i>trans</i> = 'C', then $A^H X = B$ is solved for $X$ .
<i>diag</i>	CHARACTER*1. Must be 'N' or 'U'. If <i>diag</i> = 'N', then $A$ is not a unit triangular matrix. If <i>diag</i> = 'U', then $A$ is unit triangular: diagonal elements are assumed to be 1 and not referenced in the array <i>ap</i> .
<i>n</i>	INTEGER. The order of $A$ ; the number of rows in $B$ ( $n \geq 0$ ).
<i>nrhs</i>	INTEGER. The number of right-hand sides ( $nrhs \geq 0$ ).

<i>ap, b</i>	<small>REAL for <code>stptrs</code> DOUBLE PRECISION for <code>dptrs</code> COMPLEX for <code>ctptrs</code> DOUBLE COMPLEX for <code>zptrs</code>.</small>
	<small>Arrays: <i>ap(*)</i>, <i>b(ldb,*)</i></small>
	<small>The dimension of <i>ap</i> must be at least <math>\max(1, n(n+1)/2)</math>. The array <i>ap</i> contains the matrix <i>A</i> in <i>packed storage</i> (see <a href="#">Matrix Storage Schemes</a>).</small>
	<small>The array <i>b</i> contains the matrix <i>B</i> whose columns are the right-hand sides for the system of equations. The second dimension of <i>b</i> must be at least <math>\max(1, nrhs)</math>.</small>
<i>ldb</i>	<small>INTEGER. The first dimension of <i>b</i>; <math>ldb \geq \max(1, n)</math>.</small>

## Output Parameters

<i>b</i>	Overwritten by the solution matrix <i>X</i> .
<i>info</i>	<small>INTEGER. If <i>info</i>=0, the execution is successful. If <i>info</i> = -<i>i</i>, the <i>i</i>th parameter had an illegal value.</small>

## Application Notes

For each right-hand side *b*, the computed solution is the exact solution of a perturbed system of equations  $(A + E)x = b$  where

$$|E| \leq c(n)\varepsilon|A|$$

$c(n)$  is a modest linear function of *n*, and  $\varepsilon$  is the machine precision.

If  $x_0$  is the true solution, the computed solution *x* satisfies this error bound:

$$\frac{\|x - x_0\|_\infty}{\|x\|_\infty} \leq c(n) \operatorname{cond}(A, x)\varepsilon, \text{ provided } c(n) \operatorname{cond}(A, x)\varepsilon < 1$$

where  $\operatorname{cond}(A, x) = \| |A^{-1}| |A| |x| \|_\infty / \|x\|_\infty \leq \|A^{-1}\|_\infty \|A\|_\infty = \kappa_\infty(A)$ .

Note that  $\operatorname{cond}(A, x)$  can be much smaller than  $\kappa_\infty(A)$ ; the condition number of  $A^T$  and  $A^H$  might or might not be equal to  $\kappa_\infty(A)$ .

The approximate number of floating-point operations for one right-hand side vector *b* is  $n^2$  for real flavors and  $4n^2$  for complex flavors.

To estimate the condition number  $\kappa_\infty(A)$ , call [`tpcon`](#).

To estimate the error in the solution, call [`tprfs`](#).

## ?tbtrs

Solves a system of linear equations with a band triangular matrix, with multiple right-hand sides.

```
call stbtrs (uplo, trans, diag, n, kd, nrhs, ab, ldab, b, ldb, info)
call dtbtrs (uplo, trans, diag, n, kd, nrhs, ab, ldab, b, ldb, info)
call ctbtrs (uplo, trans, diag, n, kd, nrhs, ab, ldab, b, ldb, info)
call ztbtrs (uplo, trans, diag, n, kd, nrhs, ab, ldab, b, ldb, info)
```

### Discussion

This routine solves for  $X$  the following systems of linear equations with a band triangular matrix  $A$ , with multiple right-hand sides stored in  $B$ :

$$\begin{aligned} AX = B & \quad \text{if } \text{trans} = 'N', \\ A^T X = B & \quad \text{if } \text{trans} = 'T', \\ A^H X = B & \quad \text{if } \text{trans} = 'C' \text{ (for complex matrices only).} \end{aligned}$$

### Input Parameters

<i>uplo</i>	CHARACTER*1. Must be ' <i>U</i> ' or ' <i>L</i> '.
	Indicates whether $A$ is upper or lower triangular: If <i>uplo</i> = ' <i>U</i> ', then $A$ is upper triangular. If <i>uplo</i> = ' <i>L</i> ', then $A$ is lower triangular.
<i>trans</i>	CHARACTER*1. Must be ' <i>N</i> ' or ' <i>T</i> ' or ' <i>C</i> '.
	If <i>trans</i> = ' <i>N</i> ', then $AX = B$ is solved for $X$ . If <i>trans</i> = ' <i>T</i> ', then $A^T X = B$ is solved for $X$ . If <i>trans</i> = ' <i>C</i> ', then $A^H X = B$ is solved for $X$ .
<i>diag</i>	CHARACTER*1. Must be ' <i>N</i> ' or ' <i>U</i> '.
	If <i>diag</i> = ' <i>N</i> ', then $A$ is not a unit triangular matrix. If <i>diag</i> = ' <i>U</i> ', then $A$ is unit triangular: diagonal elements are assumed to be 1 and not referenced in the array <i>ab</i> .
<i>n</i>	INTEGER. The order of $A$ ; the number of rows in $B$ ( <i>n</i> $\geq 0$ ).
<i>kd</i>	INTEGER. The number of super-diagonals or sub-diagonals in the matrix $A$ ( <i>kd</i> $\geq 0$ ).
<i>nrhs</i>	INTEGER. The number of right-hand sides ( <i>nrhs</i> $\geq 0$ ).

<i>ab</i> , <i>b</i>	<small>REAL for <b>stbtrs</b> DOUBLE PRECISION for <b>dtbtrs</b> COMPLEX for <b>ctbtrs</b> DOUBLE COMPLEX for <b>ztbtrs</b>.</small>
	<small>Arrays: <i>ab( 1dab, * ), b( 1db, * ).</i></small>
	The array <i>ab</i> contains the matrix <i>A</i> in <i>band storage</i> form. The array <i>b</i> contains the matrix <i>B</i> whose columns are the right-hand sides for the systems of equations.
	The second dimension of <i>ab</i> must be at least $\max(1, n)$ , the second dimension of <i>b</i> at least $\max(1, nrhs)$ .
<i>ldab</i>	<small>INTEGER. The first dimension of <i>ab</i>; <i>ldab</i> <math>\geq kd + 1</math>.</small>
<i>ldb</i>	<small>INTEGER. The first dimension of <i>b</i>; <i>ldb</i> <math>\geq \max(1, n)</math>.</small>

### Output Parameters

<i>b</i>	Overwritten by the solution matrix <i>X</i> .
<i>info</i>	<small>INTEGER. If <i>info</i>=0, the execution is successful. If <i>info</i> = -<i>i</i>, the <i>i</i>th parameter had an illegal value.</small>

### Application Notes

For each right-hand side *b*, the computed solution is the exact solution of a perturbed system of equations  $(A + E)x = b$  where

$$|E| \leq c(n)\varepsilon|A|$$

*c(n)* is a modest linear function of *n*, and  $\varepsilon$  is the machine precision.  
If  $x_0$  is the true solution, the computed solution *x* satisfies this error bound:

$$\frac{\|x - x_0\|_\infty}{\|x\|_\infty} \leq c(n) \operatorname{cond}(A, x)\varepsilon, \text{ provided } c(n) \operatorname{cond}(A, x)\varepsilon < 1$$

where  $\operatorname{cond}(A, x) = \| |A^{-1}| |A| |x| \|_\infty / \|x\|_\infty \leq \|A^{-1}\|_\infty \|A\|_\infty = \kappa_\infty(A)$ .

Note that  $\operatorname{cond}(A, x)$  can be much smaller than  $\kappa_\infty(A)$ ; the condition number of  $A^T$  and  $A^H$  might or might not be equal to  $\kappa_\infty(A)$ .

The approximate number of floating-point operations for one right-hand side vector *b* is  $2n * kd$  for real flavors and  $8n * kd$  for complex flavors.

To estimate the condition number  $\kappa_\infty(A)$ , call [?tbcon](#).

To estimate the error in the solution, call [?tbrfs](#).

## Routines for Estimating the Condition Number

This section describes the LAPACK routines for estimating the *condition number* of a matrix. The condition number is used for analyzing the errors in the solution of a system of linear equations (see [Error Analysis](#)). Since the condition number may be arbitrarily large when the matrix is nearly singular, the routines actually compute the *reciprocal* condition number.

### ?gecon

*Estimates the reciprocal of the condition number of a general matrix in either the 1-norm or the infinity-norm.*

```
call sgecon ( norm,n,a,lda,anorm,rcond,work,iwork,info )
call dgecon ( norm,n,a,lda,anorm,rcond,work,iwork,info )

call cgecon ( norm,n,a,lda,anorm,rcond,work,rwork,info )
call zgecon ( norm,n,a,lda,anorm,rcond,work,rwork,info )
```

#### Discussion

This routine estimates the reciprocal of the condition number of a general matrix  $A$  in either the 1-norm or infinity-norm:

$$\begin{aligned}\kappa_1(A) &= \|A\|_1 \|A^{-1}\|_1 = \kappa_\infty(A^T) = \kappa_\infty(A^H) \\ \kappa_\infty(A) &= \|A\|_\infty \|A^{-1}\|_\infty = \kappa_1(A^T) = \kappa_1(A^H).\end{aligned}$$

Before calling this routine:

- compute *anorm* (either  $\|A\|_1 = \max_j \sum_i |a_{ij}|$  or  $\|A\|_\infty = \max_i \sum_j |a_{ij}|$ )
- call [?getrf](#) to compute the *LU* factorization of  $A$ .

#### Input Parameters

<i>norm</i>	CHARACTER*1. Must be ' <b>1</b> ' or ' <b>O</b> ' or ' <b>I</b> '.
	If <i>norm</i> = ' <b>1</b> ' or ' <b>O</b> ', then the routine estimates $\kappa_1(A)$ .
	If <i>norm</i> = ' <b>I</b> ', then the routine estimates $\kappa_\infty(A)$ .
<i>n</i>	INTEGER. The order of the matrix $A$ ( $n \geq 0$ ).

<i>a, work</i>	<small>REAL for sgecon DOUBLE PRECISION for dgecon COMPLEX for cgecon DOUBLE COMPLEX for zgecon.</small> Arrays: <i>a</i> ( <i>lda</i> , *), <i>work</i> ( * ).
	The array <i>a</i> contains the <i>LU</i> -factored matrix <i>A</i> , as returned by <a href="#">?getrf</a> . The second dimension of <i>a</i> must be at least $\max(1, n)$ . The array <i>work</i> is a workspace for the routine. The dimension of <i>work</i> must be at least $\max(1, 4*n)$ for real flavors and $\max(1, 2*n)$ for complex flavors.
<i>anorm</i>	<small>REAL for single precision flavors. DOUBLE PRECISION for double precision flavors.</small> The norm of the <i>original</i> matrix <i>A</i> (see <a href="#">Discussion</a> ).
<i>lda</i>	<small>INTEGER.</small> The first dimension of <i>a</i> ; <i>lda</i> $\geq \max(1, n)$ .
<i>iwork</i>	<small>INTEGER.</small> Workspace array, <small>DIMENSION</small> at least $\max(1, n)$ .
<i>rwork</i>	<small>REAL for cgecon DOUBLE PRECISION for zgecon</small> Workspace array, <small>DIMENSION</small> at least $\max(1, 2*n)$ .

## Output Parameters

<i>rcond</i>	<small>REAL for single precision flavors. DOUBLE PRECISION for double precision flavors.</small> An estimate of the reciprocal of the condition number. The routine sets <i>rcond</i> =0 if the estimate underflows; in this case the matrix is singular (to working precision). However, anytime <i>rcond</i> is small compared to 1.0, for the working precision, the matrix may be poorly conditioned or even singular.
<i>info</i>	<small>INTEGER.</small> If <i>info</i> =0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> th parameter had an illegal value.

## Application Notes

The computed *rcond* is never less than  $\rho$  (the reciprocal of the true condition number) and in practice is nearly always less than  $10\rho$ . A call to this routine involves solving a number of systems of linear equations  $Ax = b$  or  $A^Hx = b$ ; the number is usually 4 or 5 and never more than 11. Each solution requires approximately  $2n^2$  floating-point operations for real flavors and  $8n^2$  for complex flavors.

## ?gbcon

Estimates the reciprocal of the condition number of a band matrix in either the 1-norm or the infinity-norm.

```
call sgbcon (norm,n,kl,ku,ab,ldab,ipiv,anorm,rcond,work,iwork,info)
call dgbcon (norm,n,kl,ku,ab,ldab,ipiv,anorm,rcond,work,iwork,info)
call cgbcon (norm,n,kl,ku,ab,ldab,ipiv,anorm,rcond,work,rwork,info)
call zgbcon (norm,n,kl,ku,ab,ldab,ipiv,anorm,rcond,work,rwork,info)
```

### Discussion

This routine estimates the reciprocal of the condition number of a general band matrix  $A$  in either the 1-norm or infinity-norm:

$$\begin{aligned}\kappa_1(A) &= \|A\|_1 \|A^{-1}\|_1 = \kappa_\infty(A^T) = \kappa_\infty(A^H) \\ \kappa_\infty(A) &= \|A\|_\infty \|A^{-1}\|_\infty = \kappa_1(A^T) = \kappa_1(A^H).\end{aligned}$$

Before calling this routine:

- compute *anorm* (either  $\|A\|_1 = \max_j \sum_i |a_{ij}|$  or  $\|A\|_\infty = \max_i \sum_j |a_{ij}|$ )
- call [?gbtrf](#) to compute the LU factorization of  $A$ .

### Input Parameters

<i>norm</i>	CHARACTER*1. Must be '1' or 'O' or 'I'. If <i>norm</i> = '1' or 'O', then the routine estimates $\kappa_1(A)$ . If <i>norm</i> = 'I', then the routine estimates $\kappa_\infty(A)$ .
<i>n</i>	INTEGER. The order of the matrix $A$ ( <i>n</i> $\geq$ 0).
<i>kl</i>	INTEGER. The number of sub-diagonals within the band of $A$ ( <i>kl</i> $\geq$ 0).
<i>ku</i>	INTEGER. The number of super-diagonals within the band of $A$ ( <i>ku</i> $\geq$ 0).
<i>ldab</i>	INTEGER. The first dimension of the array <i>ab</i> . ( <i>ldab</i> $\geq$ $2\text{kl} + \text{ku} + 1$ ).
<i>ipiv</i>	INTEGER. Array, DIMENSION at least $\max(1, n)$ . The <i>ipiv</i> array, as returned by <a href="#">?gbtrf</a> .

<i>ab</i> , <i>work</i>	<b>REAL</b> for <i>sgbcon</i> <b>DOUBLE PRECISION</b> for <i>dgbcon</i> <b>COMPLEX</b> for <i>cgbcon</i> <b>DOUBLE COMPLEX</b> for <i>zgbcon</i> . Arrays: <i>ab( 1:dab, * )</i> , <i>work( * )</i> .
	The array <i>ab</i> contains the factored band matrix <i>A</i> , as returned by <a href="#">?gbtrf</a> .
	The second dimension of <i>ab</i> must be at least $\max(1, n)$ . The array <i>work</i> is a workspace for the routine.
	The dimension of <i>work</i> must be at least $\max(1, 3 * n)$ for real flavors and $\max(1, 2 * n)$ for complex flavors.
<i>anorm</i>	<b>REAL</b> for single precision flavors. <b>DOUBLE PRECISION</b> for double precision flavors. The norm of the <i>original</i> matrix <i>A</i> (see <a href="#">Discussion</a> ).
<i>iwork</i>	<b>INTEGER</b> . Workspace array, <b>DIMENSION</b> at least $\max(1, n)$ .
<i>rwork</i>	<b>REAL</b> for <i>cgbcon</i> <b>DOUBLE PRECISION</b> for <i>zgbcon</i> Workspace array, <b>DIMENSION</b> at least $\max(1, 2 * n)$ .

## Output Parameters

<i>rcond</i>	<b>REAL</b> for single precision flavors. <b>DOUBLE PRECISION</b> for double precision flavors. An estimate of the reciprocal of the condition number. The routine sets <i>rcond</i> =0 if the estimate underflows; in this case the matrix is singular (to working precision). However, anytime <i>rcond</i> is small compared to 1.0, for the working precision, the matrix may be poorly conditioned or even singular.
<i>info</i>	<b>INTEGER</b> . If <i>info</i> =0, the execution is successful. If <i>info</i> = <i>-i</i> , the <i>i</i> th parameter had an illegal value.

## Application Notes

The computed  $rcond$  is never less than  $\rho$  (the reciprocal of the true condition number) and in practice is nearly always less than  $10\rho$ . A call to this routine involves solving a number of systems of linear equations  $Ax = b$  or  $A^Hx = b$ ; the number is usually 4 or 5 and never more than 11. Each solution requires approximately  $2n(ku + 2k1)$  floating-point operations for real flavors and  $8n(ku + 2k1)$  for complex flavors.

## ?gtcon

*Estimates the reciprocal of the condition number of a tridiagonal matrix using the factorization computed by ?gttrf.*

```
call sgtcon ( norm,n,dl,d,du,du2,ipiv,anorm,rcond,work,iwork,info )
call dgtcon ( norm,n,dl,d,du,du2,ipiv,anorm,rcond,work,iwork,info )

call cgtcon ( norm,n,dl,d,du,du2,ipiv,anorm,rcond,work,info )
call zgtcon ( norm,n,dl,d,du,du2,ipiv,anorm,rcond,work,info )
```

## Discussion

This routine estimates the reciprocal of the condition number of a real or complex tridiagonal matrix  $A$  in either the 1-norm or infinity-norm:

$$\begin{aligned}\kappa_1(A) &= \|A\|_1 \|A^{-1}\|_1 \\ \kappa_\infty(A) &= \|A\|_\infty \|A^{-1}\|_\infty\end{aligned}$$

An estimate is obtained for  $\|A^{-1}\|$ , and the reciprocal of the condition number is computed as  $rcond = 1 / (\|A\| \|A^{-1}\|)$ .

Before calling this routine:

- compute  $anorm$  (either  $\|A\|_1 = \max_j \sum_i |a_{ij}|$  or  $\|A\|_\infty = \max_i \sum_j |a_{ij}|$ )
- call [?gttrf](#) to compute the  $LU$  factorization of  $A$ .

## Input Parameters

<i>norm</i>	CHARACTER*1. Must be '1' or 'O' or 'I'. If <i>norm</i> = '1' or 'O', then the routine estimates $\kappa_1(A)$ . If <i>norm</i> = 'I', then the routine estimates $\kappa_\infty(A)$ .
<i>n</i>	INTEGER. The order of the matrix <i>A</i> ( <i>n</i> ≥ 0).
<i>dl, d, du, du2</i>	REAL for <i>sgtcon</i> DOUBLE PRECISION for <i>dgtcon</i> COMPLEX for <i>cgtcon</i> DOUBLE COMPLEX for <i>zgtcon</i> . Arrays: <i>dl</i> ( <i>n</i> - 1), <i>d</i> ( <i>n</i> ), <i>du</i> ( <i>n</i> - 1), <i>du2</i> ( <i>n</i> - 2). The array <i>dl</i> contains the ( <i>n</i> - 1) multipliers that define the matrix <i>L</i> from the <i>LU</i> factorization of <i>A</i> as computed by <a href="#">?gttrf</a> . The array <i>d</i> contains the <i>n</i> diagonal elements of the upper triangular matrix <i>U</i> from the <i>LU</i> factorization of <i>A</i> . The array <i>du</i> contains the ( <i>n</i> - 1) elements of the first super-diagonal of <i>U</i> . The array <i>du2</i> contains the ( <i>n</i> - 2) elements of the second super-diagonal of <i>U</i> .
<i>ipiv</i>	INTEGER. Array, DIMENSION ( <i>n</i> ). The array of pivot indices, as returned by <a href="#">?gttrf</a> .
<i>anorm</i>	REAL for single precision flavors. DOUBLE PRECISION for double precision flavors. The norm of the <i>original</i> matrix <i>A</i> (see <i>Discussion</i> ).
<i>work</i>	REAL for <i>sgtcon</i> DOUBLE PRECISION for <i>dgtcon</i> COMPLEX for <i>cgtcon</i> DOUBLE COMPLEX for <i>zgtcon</i> . Workspace array, DIMENSION (2 * <i>n</i> ).
<i>iwork</i>	INTEGER. Workspace array, DIMENSION ( <i>n</i> ). Used for real flavors only.

## Output Parameters

<i>rcond</i>	<code>REAL</code> for single precision flavors. <code>DOUBLE PRECISION</code> for double precision flavors.
	An estimate of the reciprocal of the condition number.
	The routine sets <i>rcond</i> =0 if the estimate underflows; in this case the matrix is singular (to working precision). However, anytime <i>rcond</i> is small compared to 1.0, for the working precision, the matrix may be poorly conditioned or even singular.
<i>info</i>	<code>INTEGER</code> . If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> th parameter had an illegal value.

## Application Notes

The computed *rcond* is never less than  $\rho$  (the reciprocal of the true condition number) and in practice is nearly always less than  $10\rho$ . A call to this routine involves solving a number of systems of linear equations  $Ax = b$ ; the number is usually 4 or 5 and never more than 11. Each solution requires approximately  $2n^2$  floating-point operations for real flavors and  $8n^2$  for complex flavors.

---

## ?pocon

*Estimates the reciprocal of the condition number of a symmetric (Hermitian) positive-definite matrix.*

---

```
call spocon ( uplo,n,a,lda,anorm,rcond,work,iwork,info )
call dpocon ( uplo,n,a,lda,anorm,rcond,work,iwork,info )
call cpocon ( uplo,n,a,lda,anorm,rcond,work,rwork,info )
call zpocon ( uplo,n,a,lda,anorm,rcond,work,rwork,info )
```

### Discussion

This routine estimates the reciprocal of the condition number of a symmetric (Hermitian) positive-definite matrix  $A$ :

$$\kappa_1(A) = \|A\|_1 \|A^{-1}\|_1 \quad (\text{since } A \text{ is symmetric or Hermitian}, \kappa_\infty(A) = \kappa_1(A)).$$

Before calling this routine:

- compute *anorm* (either  $\|A\|_1 = \max_j \sum_i |a_{ij}|$  or  $\|A\|_\infty = \max_i \sum_j |a_{ij}|$ )
- call [?potrf](#) to compute the Cholesky factorization of  $A$ .

### Input Parameters

<i>uplo</i>	CHARACTER*1. Must be ' <i>U</i> ' or ' <i>L</i> '.
	Indicates how the input matrix $A$ has been factored:
	If <i>uplo</i> = ' <i>U</i> ', the array <i>a</i> stores the upper triangular factor $U$ of the factorization $A = U^H U$ .
	If <i>uplo</i> = ' <i>L</i> ', the array <i>a</i> stores the lower triangular factor $L$ of the factorization $A = LL^H$ .
<i>n</i>	INTEGER. The order of the matrix $A$ ( $n \geq 0$ ).
<i>a, work</i>	REAL for <i>spocon</i> DOUBLE PRECISION for <i>dpocon</i> COMPLEX for <i>cpocon</i> DOUBLE COMPLEX for <i>zpocon</i> . Arrays: <i>a</i> ( <i>lda</i> , *), <i>work</i> (*).

The array  $a$  contains the factored matrix  $A$ , as returned by [?potrf](#).

The second dimension of  $a$  must be at least  $\max(1, n)$ .

The array  $work$  is a workspace for the routine.

The dimension of  $work$  must be at least  $\max(1, 3 * n)$  for real flavors and  $\max(1, 2 * n)$  for complex flavors.

$lda$  **INTEGER**. The first dimension of  $a$ ;  $lda \geq \max(1, n)$ .

$anorm$  **REAL** for single precision flavors.

**DOUBLE PRECISION** for double precision flavors.

The norm of the *original* matrix  $A$  (see *Discussion*).

$iwork$  **INTEGER**.

Workspace array, **DIMENSION** at least  $\max(1, n)$ .

$rwork$  **REAL** for `cpocon`

**DOUBLE PRECISION** for `zpocon`

Workspace array, **DIMENSION** at least  $\max(1, n)$ .

## Output Parameters

$rcond$  **REAL** for single precision flavors.

**DOUBLE PRECISION** for double precision flavors.

An estimate of the reciprocal of the condition number.

The routine sets  $rcond = 0$  if the estimate underflows; in this case the matrix is singular (to working precision).

However, anytime  $rcond$  is small compared to 1.0, for the working precision, the matrix may be poorly conditioned or even singular.

$info$  **INTEGER**.

If  $info = 0$ , the execution is successful.

If  $info = -i$ , the  $i$ th parameter had an illegal value.

## Application Notes

The computed  $rcond$  is never less than  $\rho$  (the reciprocal of the true condition number) and in practice is nearly always less than  $10\rho$ . A call to this routine involves solving a number of systems of linear equations  $Ax = b$ ; the number is usually 4 or 5 and never more than 11. Each solution requires approximately  $2n^2$  floating-point operations for real flavors and  $8n^2$  for complex flavors.

## ?ppcon

*Estimates the reciprocal of the condition number of a packed symmetric (Hermitian) positive-definite matrix.*

---

```
call sppcon ( uplo,n,ap,anorm,rcond,work,iwork,info )
call dppcon ( uplo,n,ap,anorm,rcond,work,iwork,info )

call cppcon ( uplo,n,ap,anorm,rcond,work,rwork,info )
call zppcon ( uplo,n,ap,anorm,rcond,work,rwork,info )
```

### Discussion

This routine estimates the reciprocal of the condition number of a packed symmetric (Hermitian) positive-definite matrix  $A$ :

$$\kappa_1(A) = \|A\|_1 \|A^{-1}\|_1 \text{ (since } A \text{ is symmetric or Hermitian, } \kappa_\infty(A) = \kappa_1(A)).$$

Before calling this routine:

- compute `anorm` (either  $\|A\|_1 = \max_j \sum_i |a_{ij}|$  or  $\|A\|_\infty = \max_i \sum_j |a_{ij}|$ )
- call [?pptrf](#) to compute the Cholesky factorization of  $A$ .

### Input Parameters

<code>uplo</code>	CHARACTER*1. Must be ' <code>U</code> ' or ' <code>L</code> '.
	Indicates how the input matrix $A$ has been factored:
	If <code>uplo</code> = ' <code>U</code> ', the array <code>ap</code> stores the upper triangular factor $U$ of the factorization $A = U^H U$ .
	If <code>uplo</code> = ' <code>L</code> ', the array <code>ap</code> stores the lower triangular factor $L$ of the factorization $A = L L^H$ .
<code>n</code>	INTEGER. The order of the matrix $A$ ( $n \geq 0$ ).
<code>ap, work</code>	REAL for <code>sppcon</code> DOUBLE PRECISION for <code>dppcon</code> COMPLEX for <code>cppcon</code> DOUBLE COMPLEX for <code>zppcon</code> . Arrays: <code>ap(*)</code> , <code>work(*)</code> .

The array  $\text{ap}$  contains the packed factored matrix  $A$ , as returned by [?pptrf](#).

The dimension of  $\text{ap}$  must be at least  $\max(1, n(n+1)/2)$ .

The array  $\text{work}$  is a workspace for the routine.

The dimension of  $\text{work}$  must be at least  $\max(1, 3 * n)$  for real flavors and  $\max(1, 2 * n)$  for complex flavors.

$\text{anorm}$

**REAL** for single precision flavors.

**DOUBLE PRECISION** for double precision flavors.

The norm of the *original* matrix  $A$  (see *Discussion*).

$iwork$

**INTEGER**.

Workspace array, **DIMENSION** at least  $\max(1, n)$ .

$rwork$

**REAL** for `cppcon`

**DOUBLE PRECISION** for `zppcon`

Workspace array, **DIMENSION** at least  $\max(1, n)$ .

## Output Parameters

$rcond$

**REAL** for single precision flavors.

**DOUBLE PRECISION** for double precision flavors.

An estimate of the reciprocal of the condition number.

The routine sets  $rcond = 0$  if the estimate underflows; in this case the matrix is singular (to working precision).

However, anytime  $rcond$  is small compared to 1.0, for the working precision, the matrix may be poorly conditioned or even singular.

$info$

**INTEGER**.

If  $info = 0$ , the execution is successful.

If  $info = -i$ , the  $i$ th parameter had an illegal value.

## Application Notes

The computed  $rcond$  is never less than  $\rho$  (the reciprocal of the true condition number) and in practice is nearly always less than  $10\rho$ . A call to this routine involves solving a number of systems of linear equations  $Ax = b$ ; the number is usually 4 or 5 and never more than 11. Each solution requires approximately  $2n^2$  floating-point operations for real flavors and  $8n^2$  for complex flavors.

---

## ?pbcon

*Estimates the reciprocal of the condition number of a symmetric (Hermitian) positive-definite band matrix.*

---

```
call spbcon (uplo, n, kd, ab, ldab, anorm, rcond, work, iwork, info)
call dpbcon (uplo, n, kd, ab, ldab, anorm, rcond, work, iwork, info)
call cpbcon (uplo, n, kd, ab, ldab, anorm, rcond, work, rwork, info)
call zpbcon (uplo, n, kd, ab, ldab, anorm, rcond, work, rwork, info)
```

### Discussion

This routine estimates the reciprocal of the condition number of a symmetric (Hermitian) positive-definite band matrix  $A$ :

$$\kappa_1(A) = \|A\|_1 \|A^{-1}\|_1 \quad (\text{since } A \text{ is symmetric or Hermitian}, \kappa_\infty(A) = \kappa_1(A)).$$

Before calling this routine:

- compute *anorm* (either  $\|A\|_1 = \max_j \sum_i |a_{ij}|$  or  $\|A\|_\infty = \max_i \sum_j |a_{ij}|$ )
- call [?pbtrf](#) to compute the Cholesky factorization of  $A$ .

### Input Parameters

<i>uplo</i>	CHARACTER*1. Must be ' <b>U</b> ' or ' <b>L</b> '.
	Indicates how the input matrix $A$ has been factored:
	If <i>uplo</i> = ' <b>U</b> ', the array <i>ab</i> stores the upper triangular factor $U$ of the Cholesky factorization $A = U^H U$ .
	If <i>uplo</i> = ' <b>L</b> ', the array <i>ab</i> stores the lower triangular factor $L$ of the factorization $A = L L^H$ .
<i>n</i>	INTEGER. The order of the matrix $A$ ( <i>n</i> $\geq 0$ ).
<i>kd</i>	INTEGER. The number of super-diagonals or sub-diagonals in the matrix $A$ ( <i>kd</i> $\geq 0$ ).
<i>ldab</i>	INTEGER. The first dimension of the array <i>ab</i> . ( <i>ldab</i> $\geq$ <i>kd</i> + 1).
<i>ab, work</i>	REAL for <b>spbcon</b> DOUBLE PRECISION for <b>dpbcon</b> COMPLEX for <b>cpbcon</b> DOUBLE COMPLEX for <b>zpbcon</b> .

Arrays:  $\text{ab}(\text{ldab}, *), \text{work}(*)$ .

The array  $\text{ab}$  contains the factored matrix  $A$  in band form, as returned by [?pbtrf](#).

The second dimension of  $\text{ab}$  must be at least  $\max(1, n)$ ,

The array  $\text{work}$  is a workspace for the routine.

The dimension of  $\text{work}$  must be at least  $\max(1, 3*n)$  for real flavors and  $\max(1, 2*n)$  for complex flavors.

$\text{anorm}$

`REAL` for single precision flavors.

`DOUBLE PRECISION` for double precision flavors.

The norm of the *original* matrix  $A$  (see *Discussion*).

$iwork$

`INTEGER`.

Workspace array, `DIMENSION` at least  $\max(1, n)$ .

$rwork$

`REAL` for `cpbcon`

`DOUBLE PRECISION` for `zpbcon`.

Workspace array, `DIMENSION` at least  $\max(1, n)$ .

## Output Parameters

$rcond$

`REAL` for single precision flavors.

`DOUBLE PRECISION` for double precision flavors.

An estimate of the reciprocal of the condition number.

The routine sets  $rcond=0$  if the estimate underflows; in this case the matrix is singular (to working precision).

However, anytime  $rcond$  is small compared to 1.0, for the working precision, the matrix may be poorly conditioned or even singular.

$info$

`INTEGER`. If  $info=0$ , the execution is successful.

If  $info = -i$ , the  $i$ th parameter had an illegal value.

## Application Notes

The computed  $rcond$  is never less than  $\rho$  (the reciprocal of the true condition number) and in practice is nearly always less than  $10\rho$ . A call to this routine involves solving a number of systems of linear equations  $Ax = b$ ; the number is usually 4 or 5 and never more than 11. Each solution requires approximately  $4n(kd + 1)$  floating-point operations for real flavors and  $16n(kd + 1)$  for complex flavors.

---

## ?ptcon

*Estimates the reciprocal of the condition number of a symmetric (Hermitian) positive-definite tridiagonal matrix.*

---

```
call sptcon (n, d, e, anorm, rcond, work, info)
call dptcon (n, d, e, anorm, rcond, work, info)

call cptcon (n, d, e, anorm, rcond, work, info)
call zptcon (n, d, e, anorm, rcond, work, info)
```

### Discussion

This routine computes the reciprocal of the condition number (in the 1-norm) of a real symmetric or complex Hermitian positive-definite tridiagonal matrix using the factorization  $A = LDL^H$  or  $A = U^H D U$  computed by [?ptrf](#) :

$$\kappa_1(A) = \|A\|_1 \|A^{-1}\|_1 \text{ (since } A \text{ is symmetric or Hermitian, } \kappa_\infty(A) = \kappa_1(A)).$$

The norm  $\|A^{-1}\|$  is computed by a direct method, and the reciprocal of the condition number is computed as  $rcond = 1 / (\|A\| \|A^{-1}\|)$ .

Before calling this routine:

- compute *anorm* as  $\|A\|_1 = \max_j \sum_i |a_{ij}|$
- call [?ptrf](#) to compute the factorization of *A*.

### Input Parameters

<i>n</i>	INTEGER. The order of the matrix <i>A</i> ( <i>n</i> $\geq 0$ ).
<i>d</i> , <i>work</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Arrays, dimension ( <i>n</i> ). The array <i>d</i> contains the <i>n</i> diagonal elements of the diagonal matrix <i>D</i> from the factorization of <i>A</i> , as computed by <a href="#">?ptrf</a> ; <i>work</i> is a workspace array.

---

<i>e</i>	<code>REAL</code> for <code>sptcon</code> <code>DOUBLE PRECISION</code> for <code>dptcon</code> <code>COMPLEX</code> for <code>cptcon</code> <code>DOUBLE COMPLEX</code> for <code>zptcon</code> . Array, <code>DIMENSION (n - 1)</code> . Contains off-diagonal elements of the unit bidiagonal factor <i>U</i> or <i>L</i> from the factorization computed by <a href="#">?pttrf</a> .
<i>anorm</i>	<code>REAL</code> for single precision flavors. <code>DOUBLE PRECISION</code> for double precision flavors. The 1- norm of the <i>original</i> matrix <i>A</i> (see <i>Discussion</i> ).

## Output Parameters

<i>rcond</i>	<code>REAL</code> for single precision flavors. <code>DOUBLE PRECISION</code> for double precision flavors. An estimate of the reciprocal of the condition number. The routine sets <i>rcond</i> =0 if the estimate underflows; in this case the matrix is singular (to working precision). However, anytime <i>rcond</i> is small compared to 1.0, for the working precision, the matrix may be poorly conditioned or even singular.
<i>info</i>	<code>INTEGER</code> . If <i>info</i> = 0, the execution is successful. If <i>info</i> = <i>-i</i> , the <i>i</i> th parameter had an illegal value.

## Application Notes

The computed *rcond* is never less than  $\rho$  (the reciprocal of the true condition number) and in practice is nearly always less than  $10\rho$ . A call to this routine involves solving a number of systems of linear equations  $Ax = b$ ; the number is usually 4 or 5 and never more than 11. Each solution requires approximately  $4n(kd + 1)$  floating-point operations for real flavors and  $16n(kd + 1)$  for complex flavors.

---

## ?sycon

*Estimates the reciprocal of the condition number of a symmetric matrix.*

---

```
call ssycon (uplo, n, a, lda, ipiv, anorm, rcond, work, iwork, info)
call dsycon (uplo, n, a, lda, ipiv, anorm, rcond, work, iwork, info)
call csycon (uplo, n, a, lda, ipiv, anorm, rcond, work, rwork, info)
call zsycon (uplo, n, a, lda, ipiv, anorm, rcond, work, rwork, info)
```

### Discussion

This routine estimates the reciprocal of the condition number of a symmetric matrix  $A$ :

$$\kappa_1(A) = \|A\|_1 \|A^{-1}\|_1 \quad (\text{since } A \text{ is symmetric}, \kappa_\infty(A) = \kappa_1(A)).$$

Before calling this routine:

- compute *anorm* (either  $\|A\|_1 = \max_j \sum_i |a_{ij}|$  or  $\|A\|_\infty = \max_i \sum_j |a_{ij}|$ )
- call [?sytrf](#) to compute the factorization of  $A$ .

### Input Parameters

<i>uplo</i>	CHARACTER*1. Must be ' <i>U</i> ' or ' <i>L</i> '.
	Indicates how the input matrix $A$ has been factored:
	If <i>uplo</i> = ' <i>U</i> ', the array <i>a</i> stores the upper triangular factor $U$ of the factorization $A = PUDU^TP^T$ .
	If <i>uplo</i> = ' <i>L</i> ', the array <i>a</i> stores the lower triangular factor $L$ of the factorization $A = PLDL^TP^T$ .
<i>n</i>	INTEGER. The order of matrix $A$ ( <i>n</i> $\geq 0$ ).
<i>a</i> , <i>work</i>	REAL for <b>ssycon</b> DOUBLE PRECISION for <b>dsycon</b> COMPLEX for <b>csycon</b> DOUBLE COMPLEX for <b>zsycon</b> . Arrays: <i>a</i> ( <i>lda</i> , *), <i>work</i> ( * ).
	The array <i>a</i> contains the factored matrix $A$ , as returned by <a href="#">?sytrf</a> .
	The second dimension of <i>a</i> must be at least $\max(1, n)$ .

The array `work` is a workspace for the routine.

The dimension of `work` must be at least  $\max(1, 2 * n)$ .

`lda`

`INTEGER`. The first dimension of `a`;  $lda \geq \max(1, n)$ .

`ipiv`

`INTEGER`. Array, `DIMENSION` at least  $\max(1, n)$ .

The array `ipiv`, as returned by `?sytrf`.

`anorm`

`REAL` for single precision flavors.

`DOUBLE PRECISION` for double precision flavors.

The norm of the *original* matrix  $A$  (see *Discussion*).

`iwork`

`INTEGER`.

Workspace array, `DIMENSION` at least  $\max(1, n)$ .

`rwork`

`REAL` for `csycon`

`DOUBLE PRECISION` for `zsycon`.

Workspace array, `DIMENSION` at least  $\max(1, n)$ .

## Output Parameters

`rcond`

`REAL` for single precision flavors.

`DOUBLE PRECISION` for double precision flavors.

An estimate of the reciprocal of the condition number.

The routine sets `rcond`=0 if the estimate underflows; in this case the matrix is singular (to working precision).

However, anytime `rcond` is small compared to 1.0, for the working precision, the matrix may be poorly conditioned or even singular.

`info`

`INTEGER`.

If `info` = 0, the execution is successful.

If `info` =  $-i$ , the  $i$ th parameter had an illegal value.

## Application Notes

The computed `rcond` is never less than  $\rho$  (the reciprocal of the true condition number) and in practice is nearly always less than  $10\rho$ . A call to this routine involves solving a number of systems of linear equations  $Ax = b$ ; the number is usually 4 or 5 and never more than 11. Each solution requires approximately  $2n^2$  floating-point operations for real flavors and  $8n^2$  for complex flavors.

## ?hecon

*Estimates the reciprocal of the condition number of a Hermitian matrix.*

---

```
call checon (uplo, n, a, lda, ipiv, anorm, rcond, work, rwork, info)
call zhecon (uplo, n, a, lda, ipiv, anorm, rcond, work, rwork, info)
```

### Discussion

This routine estimates the reciprocal of the condition number of a Hermitian matrix  $A$ :

$$\kappa_1(A) = \|A\|_1 \|A^{-1}\|_1 \text{ (since } A \text{ is Hermitian, } \kappa_\infty(A) = \kappa_1(A)).$$

Before calling this routine:

- compute *anorm* (either  $\|A\|_1 = \max_j \sum_i |a_{ij}|$  or  $\|A\|_\infty = \max_i \sum_j |a_{ij}|$ )
- call [?hetrf](#) to compute the factorization of  $A$ .

### Input Parameters

<i>uplo</i>	CHARACTER*1. Must be ' <b>U</b> ' or ' <b>L</b> '.
	Indicates how the input matrix $A$ has been factored:
	If <i>uplo</i> = ' <b>U</b> ', the array <i>a</i> stores the upper triangular factor $U$ of the factorization $A = PUDU^H P^T$ .
	If <i>uplo</i> = ' <b>L</b> ', the array <i>a</i> stores the lower triangular factor $L$ of the factorization $A = PLDL^H P^T$ .
<i>n</i>	INTEGER. The order of matrix $A$ ( <i>n</i> $\geq 0$ ).
<i>a</i> , <i>work</i>	COMPLEX for <b>checon</b> DOUBLE COMPLEX for <b>zhecon</b> . Arrays: <i>a</i> ( <i>lda</i> , *), <i>work</i> (*).
	The array <i>a</i> contains the factored matrix $A$ , as returned by <a href="#">?hetrf</a> .
	The second dimension of <i>a</i> must be at least $\max(1, \sqrt{n})$ .
	The array <i>work</i> is a workspace for the routine.
	The dimension of <i>work</i> must be at least $\max(1, 2 * n)$ .

---

<i>lda</i>	<b>INTEGER.</b> The first dimension of <i>a</i> ; <i>lda</i> $\geq \max(1, n)$ .
<i>ipiv</i>	<b>INTEGER.</b> Array, <b>DIMENSION</b> at least $\max(1, n)$ . The array <i>ipiv</i> , as returned by <a href="#">?hetrf</a> .
<i>anorm</i>	<b>REAL</b> for single precision flavors. <b>DOUBLE PRECISION</b> for double precision flavors. The norm of the <i>original</i> matrix <i>A</i> (see <i>Discussion</i> ).
<i>rwork</i>	<b>REAL</b> for <i>checon</i> <b>DOUBLE PRECISION</b> for <i>zhecon</i> Workspace array, <b>DIMENSION</b> at least $\max(1, n)$ .

## Output Parameters

<i>rcond</i>	<b>REAL</b> for single precision flavors. <b>DOUBLE PRECISION</b> for double precision flavors. An estimate of the reciprocal of the condition number. The routine sets <i>rcond</i> =0 if the estimate underflows; in this case the matrix is singular (to working precision). However, anytime <i>rcond</i> is small compared to 1.0, for the working precision, the matrix may be poorly conditioned or even singular.
<i>info</i>	<b>INTEGER.</b> If <i>info</i> = 0, the execution is successful. If <i>info</i> = <i>-i</i> , the <i>i</i> th parameter had an illegal value.

## Application Notes

The computed *rcond* is never less than  $\rho$  (the reciprocal of the true condition number) and in practice is nearly always less than  $10\rho$ . A call to this routine involves solving a number of systems of linear equations  $Ax = b$ ; the number is usually 5 and never more than 11. Each solution requires approximately  $8n^2$  floating-point operations.

---

## ?spcon

*Estimates the reciprocal of the condition number of a packed symmetric matrix.*

---

```
call sspcon ( uplo, n, ap, ipiv, anorm, rcond, work, iwork, info )
call dspcon ( uplo, n, ap, ipiv, anorm, rcond, work, iwork, info )
call cspcon ( uplo, n, ap, ipiv, anorm, rcond, work, rwork, info )
call zspcon ( uplo, n, ap, ipiv, anorm, rcond, work, rwork, info )
```

### Discussion

This routine estimates the reciprocal of the condition number of a packed symmetric matrix  $A$ :

$$\kappa_1(A) = \|A\|_1 \|A^{-1}\|_1 \text{ (since } A \text{ is symmetric, } \kappa_\infty(A) = \kappa_1(A)).$$

Before calling this routine:

- compute *anorm* (either  $\|A\|_1 = \max_j \sum_i |a_{ij}|$  or  $\|A\|_\infty = \max_i \sum_j |a_{ij}|$ )
- call [?sptrf](#) to compute the factorization of  $A$ .

### Input Parameters

<i>uplo</i>	CHARACTER*1. Must be ' <i>U</i> ' or ' <i>L</i> '.
	Indicates how the input matrix $A$ has been factored: If <i>uplo</i> = ' <i>U</i> ', the array <i>ap</i> stores the packed upper triangular factor $U$ of the factorization $A = PUDU^TP^T$ . If <i>uplo</i> = ' <i>L</i> ', the array <i>ap</i> stores the packed lower triangular factor $L$ of the factorization $A = PLDL^TP^T$ .
<i>n</i>	INTEGER. The order of matrix $A$ ( <i>n</i> $\geq 0$ ).
<i>ap, work</i>	REAL for <b>sspcon</b> DOUBLE PRECISION for <b>dspcon</b> COMPLEX for <b>cspcon</b> DOUBLE COMPLEX for <b>zspcon</b> . Arrays: <i>ap(*)</i> , <i>work(*)</i> .
	The array <i>ap</i> contains the packed factored matrix $A$ , as returned by <a href="#">?sptrf</a> . The dimension of <i>ap</i> must be at least $\max(1, n(n+1)/2)$ .

The array `work` is a workspace for the routine.

The dimension of `work` must be at least  $\max(1, 2 * n)$ .

`ipiv`

`INTEGER`. Array, `DIMENSION` at least  $\max(1, n)$ .

The array `ipiv`, as returned by [?sptrf](#).

`anorm`

`REAL` for single precision flavors.

`DOUBLE PRECISION` for double precision flavors.

The norm of the *original* matrix  $A$  (see *Discussion*).

`iwork`

`INTEGER`.

Workspace array, `DIMENSION` at least  $\max(1, n)$ .

`rwork`

`REAL` for `cspcon`

`DOUBLE PRECISION` for `zspcon`

Workspace array, `DIMENSION` at least  $\max(1, n)$ .

## Output Parameters

`rcond`

`REAL` for single precision flavors.

`DOUBLE PRECISION` for double precision flavors.

An estimate of the reciprocal of the condition number.

The routine sets `rcond`=0 if the estimate underflows; in this case the matrix is singular (to working precision).

However, anytime `rcond` is small compared to 1.0, for the working precision, the matrix may be poorly conditioned or even singular.

`info`

`INTEGER`.

If `info` = 0, the execution is successful.

If `info` =  $-i$ , the  $i$ th parameter had an illegal value.

## Application Notes

The computed `rcond` is never less than  $\rho$  (the reciprocal of the true condition number) and in practice is nearly always less than  $10\rho$ . A call to this routine involves solving a number of systems of linear equations  $Ax = b$ ; the number is usually 4 or 5 and never more than 11. Each solution requires approximately  $2n^2$  floating-point operations for real flavors and  $8n^2$  for complex flavors.

## ?hpcon

*Estimates the reciprocal of the condition number of a packed Hermitian matrix.*

---

```
call chpcon ( uplo, n, ap, ipiv, anorm, rcond, work, rwork, info )
call zhpcon ( uplo, n, ap, ipiv, anorm, rcond, work, rwork, info )
```

### Discussion

This routine estimates the reciprocal of the condition number of a Hermitian matrix  $A$ :

$$\kappa_1(A) = \|A\|_1 \|A^{-1}\|_1 \text{ (since } A \text{ is Hermitian, } \kappa_\infty(A) = \kappa_1(A)).$$

Before calling this routine:

- compute `anorm` (either  $\|A\|_1 = \max_j \sum_i |a_{ij}|$  or  $\|A\|_\infty = \max_i \sum_j |a_{ij}|$ )
- call `?hptrf` to compute the factorization of  $A$ .

### Input Parameters

`uplo`            CHARACTER\*1. Must be '`U`' or '`L`'.

Indicates how the input matrix  $A$  has been factored:

If `uplo` = '`U`', the array `ap` stores the packed upper triangular factor  $U$  of the factorization  $A = PUDU^TP^T$ .

If `uplo` = '`L`', the array `ap` stores the packed lower triangular factor  $L$  of the factorization  $A = PLDL^TP^T$ .

`n`            INTEGER. The order of matrix  $A$  ( $n \geq 0$ ).

`ap, work`      COMPLEX for `chpcon`

DOUBLE COMPLEX for `zhpcon`.

Arrays: `ap(*)`, `work(*)`.

The array `ap` contains the packed factored matrix  $A$ , as returned by `?hptrf`.

The dimension of `ap` must be at least  $\max(1, n(n+1)/2)$ .

The array `work` is a workspace for the routine.

The dimension of `work` must be at least  $\max(1, 2*n)$ .

---

<i>ipiv</i>	<b>INTEGER.</b> Array, <b>DIMENSION</b> at least max(1, <i>n</i> ). The array <i>ipiv</i> , as returned by <a href="#">?hptrf</a> .
<i>anorm</i>	<b>REAL</b> for single precision flavors. <b>DOUBLE PRECISION</b> for double precision flavors. The norm of the <i>original</i> matrix <i>A</i> (see <i>Discussion</i> ).
<i>rwork</i>	<b>REAL</b> for <i>chpcon</i> <b>DOUBLE PRECISION</b> for <i>zhpcon</i> . Workspace array, <b>DIMENSION</b> at least max(1, <i>n</i> ).

### Output Parameters

<i>rcond</i>	<b>REAL</b> for single precision flavors. <b>DOUBLE PRECISION</b> for double precision flavors. An estimate of the reciprocal of the condition number. The routine sets <i>rcond</i> =0 if the estimate underflows; in this case the matrix is singular (to working precision). However, anytime <i>rcond</i> is small compared to 1.0, for the working precision, the matrix may be poorly conditioned or even singular.
<i>info</i>	<b>INTEGER.</b> If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> th parameter had an illegal value.

### Application Notes

The computed *rcond* is never less than  $\rho$  (the reciprocal of the true condition number) and in practice is nearly always less than  $10\rho$ . A call to this routine involves solving a number of systems of linear equations  $Ax = b$ ; the number is usually 5 and never more than 11. Each solution requires approximately  $8n^2$  floating-point operations.

---

## ?trcon

*Estimates the reciprocal of the condition number of a triangular matrix.*

---

```
call strcon (norm, uplo, diag, n, a, lda, rcond, work, iwork, info)
call dtrcon (norm, uplo, diag, n, a, lda, rcond, work, iwork, info)

call ctrcon (norm, uplo, diag, n, a, lda, rcond, work, rwork, info)
call ztrcon (norm, uplo, diag, n, a, lda, rcond, work, rwork, info)
```

### Discussion

This routine estimates the reciprocal of the condition number of a triangular matrix  $A$  in either the 1-norm or infinity-norm:

$$\begin{aligned}\kappa_1(A) &= \|A\|_1 \|A^{-1}\|_1 = \kappa_\infty(A^T) = \kappa_\infty(A^H) \\ \kappa_\infty(A) &= \|A\|_\infty \|A^{-1}\|_\infty = \kappa_1(A^T) = \kappa_1(A^H).\end{aligned}$$

### Input Parameters

<i>norm</i>	CHARACTER*1. Must be '1' or 'O' or 'I'. If <i>norm</i> = '1' or 'O', then the routine estimates $\kappa_1(A)$ . If <i>norm</i> = 'I', then the routine estimates $\kappa_\infty(A)$ .
<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. Indicates whether $A$ is upper or lower triangular: If <i>uplo</i> = 'U', the array <i>a</i> stores the upper triangle of $A$ , other array elements are not referenced. If <i>uplo</i> = 'L', the array <i>a</i> stores the lower triangle of $A$ , other array elements are not referenced.
<i>diag</i>	CHARACTER*1. Must be 'N' or 'U'. If <i>diag</i> = 'N', then $A$ is not a unit triangular matrix. If <i>diag</i> = 'U', then $A$ is unit triangular: diagonal elements are assumed to be 1 and not referenced in the array <i>a</i> .
<i>n</i>	INTEGER. The order of the matrix $A$ ( $n \geq 0$ ).

<i>a, work</i>	<code>REAL</code> for <code>strcon</code> <code>DOUBLE PRECISION</code> for <code>dtrcon</code> <code>COMPLEX</code> for <code>ctrcon</code> <code>DOUBLE COMPLEX</code> for <code>ztrcon</code> . Arrays: <i>a</i> ( <i>lda</i> , *), <i>work</i> (*).
	The array <i>a</i> contains the matrix <i>A</i> . The second dimension of <i>a</i> must be at least <code>max(1, n)</code> . The array <i>work</i> is a workspace for the routine. The dimension of <i>work</i> must be at least <code>max(1, 3 * n)</code> for real flavors and <code>max(1, 2 * n)</code> for complex flavors.
<i>lda</i>	<code>INTEGER</code> . The first dimension of <i>a</i> ; <i>lda</i> $\geq \max(1, n)$ .
<i>iwork</i>	<code>INTEGER</code> . Workspace array, <code>DIMENSION</code> at least <code>max(1, n)</code> .
<i>rwork</i>	<code>REAL</code> for <code>ctrcon</code> <code>DOUBLE PRECISION</code> for <code>ztrcon</code> . Workspace array, <code>DIMENSION</code> at least <code>max(1, n)</code> .

## Output Parameters

<i>rcond</i>	<code>REAL</code> for single precision flavors. <code>DOUBLE PRECISION</code> for double precision flavors. An estimate of the reciprocal of the condition number. The routine sets <i>rcond</i> =0 if the estimate underflows; in this case the matrix is singular (to working precision). However, anytime <i>rcond</i> is small compared to 1.0, for the working precision, the matrix may be poorly conditioned or even singular.
<i>info</i>	<code>INTEGER</code> . If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> th parameter had an illegal value.

## Application Notes

The computed *rcond* is never less than  $\rho$  (the reciprocal of the true condition number) and in practice is nearly always less than  $10\rho$ . A call to this routine involves solving a number of systems of linear equations  $Ax = b$ ; the number is usually 4 or 5 and never more than 11. Each solution requires approximately  $n^2$  floating-point operations for real flavors and  $4n^2$  operations for complex flavors.

---

## ?tpcon

*Estimates the reciprocal of the condition number of a packed triangular matrix.*

---

```
call stpcon (norm, uplo, diag, n, ap, rcond, work, iwork, info)
call dtpcon (norm, uplo, diag, n, ap, rcond, work, iwork, info)

call ctpcon (norm, uplo, diag, n, ap, rcond, work, rwork, info)
call ztpcon (norm, uplo, diag, n, ap, rcond, work, rwork, info)
```

### Discussion

This routine estimates the reciprocal of the condition number of a packed triangular matrix  $A$  in either the 1-norm or infinity-norm:

$$\begin{aligned}\kappa_1(A) &= \|A\|_1 \|A^{-1}\|_1 = \kappa_\infty(A^T) = \kappa_\infty(A^H) \\ \kappa_\infty(A) &= \|A\|_\infty \|A^{-1}\|_\infty = \kappa_1(A^T) = \kappa_1(A^H).\end{aligned}$$

### Input Parameters

<i>norm</i>	CHARACTER*1. Must be '1' or 'O' or 'I'. If <i>norm</i> = '1' or 'O', then the routine estimates $\kappa_1(A)$ . If <i>norm</i> = 'I', then the routine estimates $\kappa_\infty(A)$ .
<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. Indicates whether $A$ is upper or lower triangular: If <i>uplo</i> = 'U', the array <i>ap</i> stores the upper triangle of $A$ in packed form. If <i>uplo</i> = 'L', the array <i>ap</i> stores the lower triangle of $A$ in packed form.
<i>diag</i>	CHARACTER*1. Must be 'N' or 'U'. If <i>diag</i> = 'N', then $A$ is not a unit triangular matrix. If <i>diag</i> = 'U', then $A$ is unit triangular: diagonal elements are assumed to be 1 and not referenced in the array <i>ap</i> .
<i>n</i>	INTEGER. The order of the matrix $A$ ( $n \geq 0$ ).

---

<i>ap</i> , <i>work</i>	<b>REAL</b> for <i>stpcon</i> <b>DOUBLE PRECISION</b> for <i>dtpcon</i> <b>COMPLEX</b> for <i>ctpcon</i> <b>DOUBLE COMPLEX</b> for <i>ztpcon</i> . Arrays: <i>ap</i> (*), <i>work</i> (*).
	The array <i>ap</i> contains the packed matrix <i>A</i> .
	The dimension of <i>ap</i> must be at least $\max(1, n(n+1)/2)$ .
	The array <i>work</i> is a workspace for the routine.
	The dimension of <i>work</i> must be at least $\max(1, 3 * n)$ for real flavors and $\max(1, 2 * n)$ for complex flavors.
<i>iwork</i>	<b>INTEGER</b> . Workspace array, <b>DIMENSION</b> at least $\max(1, n)$ .
<i>rwork</i>	<b>REAL</b> for <i>ctpcon</i> <b>DOUBLE PRECISION</b> for <i>ztpcon</i> Workspace array, <b>DIMENSION</b> at least $\max(1, n)$ .

## Output Parameters

<i>rcond</i>	<b>REAL</b> for single precision flavors. <b>DOUBLE PRECISION</b> for double precision flavors. An estimate of the reciprocal of the condition number. The routine sets <i>rcond</i> =0 if the estimate underflows; in this case the matrix is singular (to working precision). However, anytime <i>rcond</i> is small compared to 1.0, for the working precision, the matrix may be poorly conditioned or even singular.
<i>info</i>	<b>INTEGER</b> . If <i>info</i> = 0, the execution is successful. If <i>info</i> = <i>-i</i> , the <i>i</i> th parameter had an illegal value.

## Application Notes

The computed *rcond* is never less than  $\rho$  (the reciprocal of the true condition number) and in practice is nearly always less than  $10\rho$ . A call to this routine involves solving a number of systems of linear equations  $Ax = b$ ; the number is usually 4 or 5 and never more than 11. Each solution requires approximately  $n^2$  floating-point operations for real flavors and  $4n^2$  operations for complex flavors.

---

## ?tbcon

*Estimates the reciprocal of the condition number of a triangular band matrix.*

---

```
call stbcon ( norm, uplo, diag, n, kd, ab, ldab, rcond, work, iwork, info )
call dtbcon ( norm, uplo, diag, n, kd, ab, ldab, rcond, work, iwork, info )
call ctbcon ( norm, uplo, diag, n, kd, ab, ldab, rcond, work, rwork, info )
call ztbcon ( norm, uplo, diag, n, kd, ab, ldab, rcond, work, rwork, info )
```

### Discussion

This routine estimates the reciprocal of the condition number of a triangular band matrix  $A$  in either the 1-norm or infinity-norm:

$$\begin{aligned}\kappa_1(A) &= \|A\|_1 \|A^{-1}\|_1 = \kappa_\infty(A^T) = \kappa_\infty(A^H) \\ \kappa_\infty(A) &= \|A\|_\infty \|A^{-1}\|_\infty = \kappa_1(A^T) = \kappa_1(A^H).\end{aligned}$$

### Input Parameters

<i>norm</i>	CHARACTER*1. Must be '1' or 'O' or 'I'. If <i>norm</i> = '1' or 'O', then the routine estimates $\kappa_1(A)$ . If <i>norm</i> = 'I', then the routine estimates $\kappa_\infty(A)$ .
<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. Indicates whether $A$ is upper or lower triangular: If <i>uplo</i> = 'U', the array <i>ap</i> stores the upper triangle of $A$ in packed form. If <i>uplo</i> = 'L', the array <i>ap</i> stores the lower triangle of $A$ in packed form.
<i>diag</i>	CHARACTER*1. Must be 'N' or 'U'. If <i>diag</i> = 'N', then $A$ is not a unit triangular matrix. If <i>diag</i> = 'U', then $A$ is unit triangular: diagonal elements are assumed to be 1 and not referenced in the array <i>ab</i> .
<i>n</i>	INTEGER. The order of the matrix $A$ ( $n \geq 0$ ).
<i>kd</i>	INTEGER. The number of super-diagonals or sub-diagonals in the matrix $A$ ( $kd \geq 0$ ).

*ab*, *work*      REAL for *stbcon*  
                   DOUBLE PRECISION for *dtbcon*  
                   COMPLEX for *ctbcon*  
                   DOUBLE COMPLEX for *ztbcon*.  
                   Arrays: *ab(1:dab, \*)*, *work(\*)*.  
                   The array *ab* contains the band matrix *A*.  
                   The second dimension of *ab* must be at least  $\max(1, n)$ .  
                   The array *work* is a workspace for the routine.  
                   The dimension of *work* must be at least  $\max(1, 3 * n)$  for real flavors and  $\max(1, 2 * n)$  for complex flavors.

*ldab*            INTEGER. The first dimension of the array *ab*.  
                   (*ldab*  $\geq kd + 1$ ).

*iwork*            INTEGER.  
                   Workspace array, DIMENSION at least  $\max(1, n)$ .

*rwork*            REAL for *ctbcon*  
                   DOUBLE PRECISION for *ztbcon*.  
                   Workspace array, DIMENSION at least  $\max(1, n)$ .

## Output Parameters

*rcond*            REAL for single precision flavors.  
                   DOUBLE PRECISION for double precision flavors.  
                   An estimate of the reciprocal of the condition number.  
                   The routine sets *rcond*=0 if the estimate underflows; in this case the matrix is singular (to working precision).  
                   However, anytime *rcond* is small compared to 1.0, for the working precision, the matrix may be poorly conditioned or even singular.

*info*            INTEGER. If *info* = 0, the execution is successful.  
                   If *info* =  $-i$ , the *i*th parameter had an illegal value.

## Application Notes

The computed *rcond* is never less than  $\rho$  (the reciprocal of the true condition number) and in practice is nearly always less than  $10\rho$ . A call to this routine involves solving a number of systems of linear equations  $Ax = b$ ; the number is usually 4 or 5 and never more than 11. Each solution requires approximately  $2n(kd + 1)$  floating-point operations for real flavors and  $8n(kd + 1)$  operations for complex flavors.

## Refining the Solution and Estimating Its Error

This section describes the LAPACK routines for refining the computed solution of a system of linear equations and estimating the solution error. You can call these routines after factorizing the matrix of the system of equations and computing the solution (see [Routines for Matrix Factorization](#) and [Routines for Solving Systems of Linear Equations](#)).

---

### ?gerfs

*Refines the solution of a system of linear equations with a general matrix and estimates its error.*

---

```
call sgerfs ( trans,n,nrhs,a,lda,af,ldaf,ipiv,b,ldb,
              x,ldx,ferr,berr,work,iwork,info)
call dgerfs ( trans,n,nrhs,a,lda,af,ldaf,ipiv,b,ldb,
              x,ldx,ferr,berr,work,iwork,info)

call cgerfs ( trans,n,nrhs,a,lda,af,ldaf,ipiv,b,ldb,
              x,ldx,ferr,berr,work,rwork,info)
call zgerfs ( trans,n,nrhs,a,lda,af,ldaf,ipiv,b,ldb,
              x,ldx,ferr,berr,work,rwork,info)
```

#### Discussion

This routine performs an iterative refinement of the solution to a system of linear equations  $AX = B$  or  $A^T X = B$  or  $A^H X = B$  with a general matrix  $A$ , with multiple right-hand sides. For each computed solution vector  $x$ , the routine computes the *component-wise backward error*  $\beta$ . This error is the smallest relative perturbation in elements of  $A$  and  $b$  such that  $x$  is the exact solution of the perturbed system:

$$|\delta a_{ij}|/|a_{ij}| \leq \beta |a_{ij}|, \quad |\delta b_i|/|b_i| \leq \beta |b_i| \text{ such that } (A + \delta A)x = (b + \delta b).$$

Finally, the routine estimates the *component-wise forward error* in the computed solution  $\|x - x_e\|_\infty / \|x\|_\infty$  (here  $x_e$  is the exact solution).

Before calling this routine:

- call the factorization routine [?getrf](#)
- call the solver routine [?getrs](#).

### Input Parameters

*trans*            CHARACTER\*1. Must be 'N' or 'T' or 'C'.

Indicates the form of the equations:

If *trans* = 'N', the system has the form  $AX = B$ .

If *trans* = 'T', the system has the form  $A^T X = B$ .

If *trans* = 'C', the system has the form  $A^H X = B$ .

*n*            INTEGER. The order of the matrix *A* (*n*  $\geq 0$ ).

*nrhs*          INTEGER. The number of right-hand sides (*nrhs*  $\geq 0$ ).

*a, af, b, x, work*    REAL for [sgerfs](#)  
                           DOUBLE PRECISION for [dgerfs](#)  
                           COMPLEX for [cgerfs](#)  
                           DOUBLE COMPLEX for [zgerfs](#).

Arrays:

*a( lda, \* )* contains the original matrix *A*, as supplied to [?getrf](#).

*af( ldaf, \* )* contains the factored matrix *A*, as returned by [?getrf](#).

*b( ldb, \* )* contains the right-hand side matrix *B*.

*x( ldx, \* )* contains the solution matrix *X*.

*work* (\*) is a workspace array.

The second dimension of *a* and *af* must be at least  $\max(1, n)$ ; the second dimension of *b* and *x* must be at least  $\max(1, nrhs)$ ; the dimension of *work* must be at least  $\max(1, 3*n)$  for real flavors and  $\max(1, 2*n)$  for complex flavors.

*lda*            INTEGER. The first dimension of *a*; *lda*  $\geq \max(1, n)$ .

*ldaf*            INTEGER. The first dimension of *af*; *ldaf*  $\geq \max(1, n)$ .

*ldb*            INTEGER. The first dimension of *b*; *ldb*  $\geq \max(1, n)$ .

*ldx*            INTEGER. The first dimension of *x*; *ldx*  $\geq \max(1, n)$ .

<i>ipiv</i>	<b>INTEGER.</b> Array, <b>DIMENSION</b> at least max(1, <i>n</i> ). The <i>ipiv</i> array, as returned by <a href="#">?getrf</a> .
<i>iwork</i>	<b>INTEGER.</b> Workspace array, <b>DIMENSION</b> at least max(1, <i>n</i> ).
<i>rwork</i>	<b>REAL</b> for <i>cgerfs</i> <b>DOUBLE PRECISION</b> for <i>zgerfs</i> . Workspace array, <b>DIMENSION</b> at least max(1, <i>n</i> ).

### Output Parameters

<i>x</i>	The refined solution matrix <i>X</i> .
<i>ferr</i> , <i>berr</i>	<b>REAL</b> for single precision flavors. <b>DOUBLE PRECISION</b> for double precision flavors. Arrays, <b>DIMENSION</b> at least max(1, <i>nrhs</i> ). Contain the component-wise forward and backward errors, respectively, for each solution vector.
<i>info</i>	<b>INTEGER.</b> If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> th parameter had an illegal value.

### Application Notes

The bounds returned in *ferr* are not rigorous, but in practice they almost always overestimate the actual error.

For each right-hand side, computation of the backward error involves a minimum of  $4n^2$  floating-point operations (for real flavors) or  $16n^2$  operations (for complex flavors). In addition, each step of iterative refinement involves  $6n^2$  operations (for real flavors) or  $24n^2$  operations (for complex flavors); the number of iterations may range from 1 to 5. Estimating the forward error involves solving a number of systems of linear equations  $Ax = b$ ; the number is usually 4 or 5 and never more than 11. Each solution requires approximately  $2n^2$  floating-point operations for real flavors or  $8n^2$  for complex flavors.

## ?gbrfs

*Refines the solution of a system of linear equations with a general band matrix and estimates its error.*

```

call sgbrfs (trans,n,k1,ku,nrhs,ab,ldab,afb,ldafb,ipiv,
             b,ldb,x,ldx,ferr,berr,work,iwork,info)

call dgbrfs (trans,n,k1,ku,nrhs,ab,ldab,afb,ldafb,ipiv,
             b,ldb,x,ldx,ferr,berr,work,iwork,info)

call cgbrfs (trans,n,k1,ku,nrhs,ab,ldab,afb,ldafb,ipiv,
             b,ldb,x,ldx,ferr,berr,work,rwork,info)

call zgbrfs (trans,n,k1,ku,nrhs,ab,ldab,afb,ldafb,ipiv,
             b,ldb,x,ldx,ferr,berr,work,rwork,info)

```

### Discussion

This routine performs an iterative refinement of the solution to a system of linear equations  $AX = B$  or  $A^T X = B$  or  $A^H X = B$  with a band matrix  $A$ , with multiple right-hand sides. For each computed solution vector  $x$ , the routine computes the *component-wise backward error*  $\beta$ . This error is the smallest relative perturbation in elements of  $A$  and  $b$  such that  $x$  is the exact solution of the perturbed system:

$$|\delta a_{ij}|/|a_{ij}| \leq \beta |a_{ij}|, \quad |\delta b_i|/|b_i| \leq \beta |b_i| \text{ such that } (A + \delta A)x = (b + \delta b).$$

Finally, the routine estimates the *component-wise forward error* in the computed solution  $\|x - x_e\|_\infty / \|x\|_\infty$  (here  $x_e$  is the exact solution).

Before calling this routine:

- call the factorization routine [?gbtrf](#)
- call the solver routine [?gbtrs](#).

## Input Parameters

<i>trans</i>	CHARACTER*1. Must be 'N' or 'T' or 'C'. Indicates the form of the equations: If <i>trans</i> = 'N', the system has the form $AX = B$ . If <i>trans</i> = 'T', the system has the form $A^T X = B$ . If <i>trans</i> = 'C', the system has the form $A^H X = B$ .
<i>n</i>	INTEGER. The order of the matrix <i>A</i> ( <i>n</i> $\geq 0$ ).
<i>kl</i>	INTEGER. The number of sub-diagonals within the band of <i>A</i> ( <i>kl</i> $\geq 0$ ).
<i>ku</i>	INTEGER. The number of super-diagonals within the band of <i>A</i> ( <i>ku</i> $\geq 0$ ).
<i>nrhs</i>	INTEGER. The number of right-hand sides ( <i>nrhs</i> $\geq 0$ ).
<i>ab,afb,b,x,work</i>	REAL for sgbrfs DOUBLE PRECISION for dgbrfs COMPLEX for cgbrfs DOUBLE COMPLEX for zgbrfs.
Arrays:	
<i>ab(1dab,*)</i>	contains the original band matrix <i>A</i> , as supplied to <a href="#">?gbtrf</a> , but stored in rows from 1 to <i>kl</i> + <i>ku</i> + 1.
<i>afb(1dafb,*)</i>	contains the factored band matrix <i>A</i> , as returned by <a href="#">?gbtrf</a> .
<i>b(1db,*)</i>	contains the right-hand side matrix <i>B</i> .
<i>x(1dx,*)</i>	contains the solution matrix <i>X</i> .
<i>work (*)</i>	is a workspace array. The second dimension of <i>ab</i> and <i>afb</i> must be at least $\max(1, n)$ ; the second dimension of <i>b</i> and <i>x</i> must be at least $\max(1, nrhs)$ ; the dimension of <i>work</i> must be at least $\max(1, 3 * n)$ for real flavors and $\max(1, 2 * n)$ for complex flavors.
<i>1dab</i>	INTEGER. The first dimension of <i>ab</i> .
<i>1dafb</i>	INTEGER. The first dimension of <i>afb</i> .
<i>1db</i>	INTEGER. The first dimension of <i>b</i> ; <i>1db</i> $\geq \max(1, n)$ .
<i>1dx</i>	INTEGER. The first dimension of <i>x</i> ; <i>1dx</i> $\geq \max(1, n)$ .

---

<i>ipiv</i>	<code>INTEGER.</code> Array, <code>DIMENSION</code> at least $\max(1, n)$ . The <i>ipiv</i> array, as returned by <a href="#">?gbtrf</a> .
<i>iwork</i>	<code>INTEGER.</code> Workspace array, <code>DIMENSION</code> at least $\max(1, n)$ .
<i>rwork</i>	<code>REAL</code> for <code>cgbrfs</code> <code>DOUBLE PRECISION</code> for <code>zgbrfs</code> Workspace array, <code>DIMENSION</code> at least $\max(1, n)$ .

### Output Parameters

<i>x</i>	The refined solution matrix $X$ .
<i>ferr</i> , <i>berr</i>	<code>REAL</code> for single precision flavors. <code>DOUBLE PRECISION</code> for double precision flavors. Arrays, <code>DIMENSION</code> at least $\max(1, nrhs)$ . Contain the component-wise forward and backward errors, respectively, for each solution vector.
<i>info</i>	<code>INTEGER.</code> If <i>info</i> = 0, the execution is successful. If <i>info</i> = $-i$ , the <i>i</i> th parameter had an illegal value.

### Application Notes

The bounds returned in *ferr* are not rigorous, but in practice they almost always overestimate the actual error.

For each right-hand side, computation of the backward error involves a minimum of  $4n(kl + ku)$  floating-point operations (for real flavors) or  $16n(kl + ku)$  operations (for complex flavors). In addition, each step of iterative refinement involves  $2n(4kl + 3ku)$  operations (for real flavors) or  $8n(4kl + 3ku)$  operations (for complex flavors); the number of iterations may range from 1 to 5. Estimating the forward error involves solving a number of systems of linear equations  $Ax = b$ ; the number is usually 4 or 5 and never more than 11. Each solution requires approximately  $2n^2$  floating-point operations for real flavors or  $8n^2$  for complex flavors.

## ?gtrfs

*Refines the solution of a system of linear equations with a tridiagonal matrix and estimates its error.*

---

```
call sgtrfs (trans, n, nrhs, dl, d, du, dlf, df, duf, du2, ipiv, b,
            ldb, x, ldx, ferr, berr, work, iwork, info)
call dgtrfs (trans, n, nrhs, dl, d, du, dlf, df, duf, du2, ipiv, b,
            ldb, x, ldx, ferr, berr, work, iwork, info)
call cgtrfs (trans, n, nrhs, dl, d, du, dlf, df, duf, du2, ipiv, b,
            ldb, x, ldx, ferr, berr, work, rwork, info)
call zgtrfs (trans, n, nrhs, dl, d, du, dlf, df, duf, du2, ipiv, b,
            ldb, x, ldx, ferr, berr, work, rwork, info)
```

### Discussion

This routine performs an iterative refinement of the solution to a system of linear equations  $AX = B$  or  $A^T X = B$  or  $A^H X = B$  with a tridiagonal matrix  $A$ , with multiple right-hand sides. For each computed solution vector  $x$ , the routine computes the *component-wise backward error*  $\beta$ . This error is the smallest relative perturbation in elements of  $A$  and  $b$  such that  $x$  is the exact solution of the perturbed system:

$$|\delta a_{ij}|/|a_{ij}| \leq \beta |a_{ij}|, \quad |\delta b_i|/|b_i| \leq \beta |b_i| \text{ such that } (A + \delta A)x = (b + \delta b).$$

Finally, the routine estimates the *component-wise forward error* in the computed solution  $\|x - x_e\|_\infty / \|x\|_\infty$  (here  $x_e$  is the exact solution).

Before calling this routine:

- call the factorization routine [?gttrf](#)
- call the solver routine [?gttrs](#).

### Input Parameters

*trans*            CHARACTER\*1. Must be '**N**' or '**T**' or '**C**'.

Indicates the form of the equations:

If *trans* = '**N**', the system has the form  $AX = B$ .

If *trans* = '**T**', the system has the form  $A^T X = B$ .

If *trans* = '**C**', the system has the form  $A^H X = B$ .

`n`                    INTEGER. The order of the matrix  $A$  ( $n \geq 0$ ).  
`nrhs`                INTEGER. The number of right-hand sides , i.e., the  
                          number of columns of the matrix  $B$  ( $nrhs \geq 0$ ).  
`d1, d, du, dlf, df,`  
`duf, du2, b, x, work` REAL for `sgtrfs`  
                          DOUBLE PRECISION for `dgtrfs`  
                          COMPLEX for `cgtrfs`  
                          DOUBLE COMPLEX for `zgtrfs`.  
    Arrays:  
`d1`, dimension ( $n - 1$ ), contains the subdiagonal  
                          elements of  $A$ .  
`d`, dimension ( $n$ ), contains the diagonal elements of  $A$ .  
`du`, dimension ( $n - 1$ ), contains the superdiagonal  
                          elements of  $A$ .  
`dlf`, dimension ( $n - 1$ ), contains the ( $n - 1$ ) multipliers  
                          that define the matrix  $L$  from the  $LU$  factorization of  $A$   
                          as computed by [?gttrf](#).  
`df`, dimension ( $n$ ), contains the  $n$  diagonal elements  
                          of the upper triangular matrix  $U$  from the  $LU$   
                          factorization of  $A$ .  
`duf`, dimension ( $n - 1$ ), contains the ( $n - 1$ ) elements  
                          of the first super-diagonal of  $U$ .  
`du2`, dimension ( $n - 2$ ), contains the ( $n - 2$ ) elements  
                          of the second super-diagonal of  $U$ .  
`b( ldb, nrhs)` contains the right-hand side matrix  $B$ .  
`x( ldx, nrhs)` contains the solution matrix  $X$ , as  
                          computed by [?gttrs](#).  
`work (*)` is a workspace array;  
                          the dimension of `work` must be at least  $\max(1, 3*n)$  for  
                          real flavors and  $\max(1, 2*n)$  for complex flavors.  
`ldb`                 INTEGER. The first dimension of `b`;  $ldb \geq \max(1, n)$ .  
`ldx`                 INTEGER. The first dimension of `x`;  $ldx \geq \max(1, n)$ .  
`ipiv`                INTEGER.  
                          Array, DIMENSION at least  $\max(1, n)$ .  
                          The `ipiv` array, as returned by [?gttrf](#).

<i>iwork</i>	<b>INTEGER.</b> Workspace array, <b>DIMENSION (n)</b> . Used for real flavors only.
<i>rwork</i>	<b>REAL</b> for <b>cgtrfs</b> <b>DOUBLE PRECISION</b> for <b>zgtrfs</b> . Workspace array, <b>DIMENSION (n)</b> . Used for complex flavors only.

## Output Parameters

<i>x</i>	The refined solution matrix $X$ .
<i>ferr</i> , <i>berr</i>	<b>REAL</b> for single precision flavors. <b>DOUBLE PRECISION</b> for double precision flavors. Arrays, <b>DIMENSION</b> at least $\max(1, nrhs)$ . Contain the component-wise forward and backward errors, respectively, for each solution vector.
<i>info</i>	<b>INTEGER</b> . If <i>info</i> = 0, the execution is successful. If <i>info</i> = <i>-i</i> , the <i>i</i> th parameter had an illegal value.

## ?porfs

*Refines the solution of a system of linear equations with a symmetric (Hermitian) positive-definite matrix and estimates its error.*

```
call sporfs (uplo,n,nrhs,a,lda,af,ldaf,b,ldb,
             x,ldx,ferr,berr,work,iwork,info)

call dporfs (uplo,n,nrhs,a,lda,af,ldaf,b,ldb,
             x,ldx,ferr,berr,work,iwork,info)

call cpors (uplo,n,nrhs,a,lda,af,ldaf,b,ldb,
             x,ldx,ferr,berr,work,rwork,info)

call zporfs (uplo,n,nrhs,a,lda,af,ldaf,b,ldb,
             x,ldx,ferr,berr,work,rwork,info)
```

### Discussion

This routine performs an iterative refinement of the solution to a system of linear equations  $AX = B$  with a symmetric (Hermitian) positive definite matrix  $A$ , with multiple right-hand sides. For each computed solution vector  $x$ , the routine computes the *component-wise backward error*  $\beta$ . This error is the smallest relative perturbation in elements of  $A$  and  $b$  such that  $x$  is the exact solution of the perturbed system:

$$|\delta a_{ij}|/|a_{ij}| \leq \beta |a_{ij}|, \quad |\delta b_i|/|b_i| \leq \beta |b_i| \text{ such that } (A + \delta A)x = (b + \delta b).$$

Finally, the routine estimates the *component-wise forward error* in the computed solution  $\|x - x_e\|_\infty / \|x\|_\infty$  (here  $x_e$  is the exact solution).

Before calling this routine:

- call the factorization routine [?potrf](#)
- call the solver routine [?potrs](#).

## Input Parameters

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. Indicates how the input matrix <i>A</i> has been factored: If <i>uplo</i> = 'U', the array <i>af</i> stores the factor <i>U</i> of the Cholesky factorization $A = U^H U$ . If <i>uplo</i> = 'L', the array <i>af</i> stores the factor <i>L</i> of the Cholesky factorization $A = LL^H$ .
<i>n</i>	INTEGER. The order of the matrix <i>A</i> ( <i>n</i> $\geq 0$ ).
<i>nrhs</i>	INTEGER. The number of right-hand sides ( <i>nrhs</i> $\geq 0$ ).
<i>a</i> , <i>af</i> , <i>b</i> , <i>x</i> , <i>work</i>	REAL for <i>sporfs</i> DOUBLE PRECISION for <i>dporfs</i> COMPLEX for <i>cporfs</i> DOUBLE COMPLEX for <i>zporfs</i> .
Arrays:	
<i>a</i> ( <i>lda</i> ,*)	contains the original matrix <i>A</i> , as supplied to <a href="#">?potrf</a> .
<i>af</i> ( <i>ldaf</i> ,*)	contains the factored matrix <i>A</i> , as returned by <a href="#">?potrf</a> .
<i>b</i> ( <i>ldb</i> ,*)	contains the right-hand side matrix <i>B</i> .
<i>x</i> ( <i>idx</i> ,*)	contains the solution matrix <i>X</i> .
<i>work</i> (*)	is a workspace array. The second dimension of <i>a</i> and <i>af</i> must be at least $\max(1, n)$ ; the second dimension of <i>b</i> and <i>x</i> must be at least $\max(1, nrhs)$ ; the dimension of <i>work</i> must be at least $\max(1, 3 * n)$ for real flavors and $\max(1, 2 * n)$ for complex flavors.
<i>lda</i>	INTEGER. The first dimension of <i>a</i> ; <i>lda</i> $\geq \max(1, n)$ .
<i>ldaf</i>	INTEGER. The first dimension of <i>af</i> ; <i>ldaf</i> $\geq \max(1, n)$ .
<i>ldb</i>	INTEGER. The first dimension of <i>b</i> ; <i>ldb</i> $\geq \max(1, n)$ .
<i>idx</i>	INTEGER. The first dimension of <i>x</i> ; <i>idx</i> $\geq \max(1, n)$ .
<i>iwork</i>	INTEGER. Workspace array, DIMENSION at least $\max(1, n)$ .
<i>rwork</i>	REAL for <i>cporfs</i> DOUBLE PRECISION for <i>zporfs</i> Workspace array, DIMENSION at least $\max(1, n)$ .

## Output Parameters

<i>x</i>	The refined solution matrix $X$ .
<i>ferr</i> , <i>berr</i>	<i>REAL</i> for single precision flavors. <i>DOUBLE PRECISION</i> for double precision flavors.
	Arrays, <i>DIMENSION</i> at least $\max(1, \text{nrhs})$ . Contain the component-wise forward and backward errors, respectively, for each solution vector.
<i>info</i>	<i>INTEGER</i> . If <i>info</i> = 0, the execution is successful. If <i>info</i> = $-i$ , the <i>i</i> th parameter had an illegal value.

## Application Notes

The bounds returned in *ferr* are not rigorous, but in practice they almost always overestimate the actual error.

For each right-hand side, computation of the backward error involves a minimum of  $4n^2$  floating-point operations (for real flavors) or  $16n^2$  operations (for complex flavors). In addition, each step of iterative refinement involves  $6n^2$  operations (for real flavors) or  $24n^2$  operations (for complex flavors); the number of iterations may range from 1 to 5.

Estimating the forward error involves solving a number of systems of linear equations  $Ax = b$ ; the number is usually 4 or 5 and never more than 11. Each solution requires approximately  $2n^2$  floating-point operations for real flavors or  $8n^2$  for complex flavors.

## ?pprfs

*Refines the solution of a system of linear equations with a packed symmetric (Hermitian) positive-definite matrix and estimates its error.*

---

```
call spprfs (uplo,n,nrhs,ap,afp,b,ldb,x,ldx,
             ferr,berr,work,iwork,info)
call dpprfs (uplo,n,nrhs,ap,afp,b,ldb,x,ldx,
             ferr,berr,work,iwork,info)

call cpprfs (uplo,n,nrhs,ap,afp,b,ldb,x,ldx,
             ferr,berr,work,rwork,info)
call zpprfs (uplo,n,nrhs,ap,afp,b,ldb,x,ldx,
             ferr,berr,work,rwork,info)
```

### Discussion

This routine performs an iterative refinement of the solution to a system of linear equations  $AX = B$  with a packed symmetric (Hermitian) positive definite matrix  $A$ , with multiple right-hand sides. For each computed solution vector  $x$ , the routine computes the *component-wise backward error*  $\beta$ . This error is the smallest relative perturbation in elements of  $A$  and  $b$  such that  $x$  is the exact solution of the perturbed system:

$$|\delta a_{ij}|/|a_{ij}| \leq \beta |a_{ij}|, \quad |\delta b_i|/|b_i| \leq \beta |b_i| \text{ such that } (A + \delta A)x = (b + \delta b).$$

Finally, the routine estimates the *component-wise forward error* in the computed solution  $\|x - x_e\|_\infty/\|x\|_\infty$  (here  $x_e$  is the exact solution).

Before calling this routine:

- call the factorization routine [?pptrf](#)
- call the solver routine [?pptrs](#).

### Input Parameters

*uplo*

CHARACTER\*1. Must be 'U' or 'L'.

Indicates how the input matrix  $A$  has been factored:

If  $\text{uplo} = \text{'U'}$ , the array  $\text{afp}$  stores the packed factor  $U$  of the Cholesky factorization  $A = U^H U$ .

If  $\text{uplo} = \text{'L'}$ , the array  $\text{afp}$  stores the packed factor  $L$  of the Cholesky factorization  $A = LL^H$ .

$n$  INTEGER. The order of the matrix  $A$  ( $n \geq 0$ ).

$nrhs$  INTEGER. The number of right-hand sides ( $nrhs \geq 0$ ).

$ap, afp, b, x, work$  REAL for spprfs  
 DOUBLE PRECISION for dpprfs  
 COMPLEX for cprfs  
 DOUBLE COMPLEX for zpprfs.

Arrays:

$ap(*)$  contains the original packed matrix  $A$ , as supplied to [?pptrf](#).

$afp(*)$  contains the factored packed matrix  $A$ , as returned by [?pptrf](#).

$b(ldb, *)$  contains the right-hand side matrix  $B$ .

$x(lidx, *)$  contains the solution matrix  $X$ .

$work(*)$  is a workspace array.

The dimension of arrays  $ap$  and  $afp$  must be at least  $\max(1, n(n+1)/2)$ ; the second dimension of  $b$  and  $x$  must be at least  $\max(1, nrhs)$ ; the dimension of  $work$  must be at least  $\max(1, 3*n)$  for real flavors and  $\max(1, 2*n)$  for complex flavors.

$ldb$  INTEGER. The first dimension of  $b$ ;  $ldb \geq \max(1, n)$ .

$lidx$  INTEGER. The first dimension of  $x$ ;  $lidx \geq \max(1, n)$ .

$iwork$  INTEGER.

Workspace array, DIMENSION at least  $\max(1, n)$ .

$rwork$  REAL for cprfs

DOUBLE PRECISION for zpprfs

Workspace array, DIMENSION at least  $\max(1, n)$ .

## Output Parameters

$x$  The refined solution matrix  $X$ .

<i>ferr</i> , <i>berr</i>	<b>REAL</b> for single precision flavors. <b>DOUBLE PRECISION</b> for double precision flavors.
	Arrays, <b>DIMENSION</b> at least $\max(1, \text{nrhs})$ . Contain the component-wise forward and backward errors, respectively, for each solution vector.
<i>info</i>	<b>INTEGER</b> . If <i>info</i> =0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> th parameter had an illegal value.

## Application Notes

The bounds returned in *ferr* are not rigorous, but in practice they almost always overestimate the actual error.

For each right-hand side, computation of the backward error involves a minimum of  $4n^2$  floating-point operations (for real flavors) or  $16n^2$  operations (for complex flavors). In addition, each step of iterative refinement involves  $6n^2$  operations (for real flavors) or  $24n^2$  operations (for complex flavors); the number of iterations may range from 1 to 5.

Estimating the forward error involves solving a number of systems of linear equations  $Ax = b$ ; the number of systems is usually 4 or 5 and never more than 11. Each solution requires approximately  $2n^2$  floating-point operations for real flavors or  $8n^2$  for complex flavors.

## ?pbrfs

Refines the solution of a system of linear equations with a band symmetric (Hermitian) positive-definite matrix and estimates its error.

```

call spbrfs (uplo,n,kd,nrhs,ab,ldab,afb,ldafb,
             b,ldb,x,ldx,ferr,berr,work,iwork,info)
call dpbrfs (uplo,n,kd,nrhs,ab,ldab,afb,ldafb,
             b,ldb,x,ldx,ferr,berr,work,iwork,info)

call cpbrfs (uplo,n,kd,nrhs,ab,ldab,afb,ldafb,
             b,ldb,x,ldx,ferr,berr,work,rwork,info)
call zpbrfs (uplo,n,kd,nrhs,ab,ldab,afb,ldafb,
             b,ldb,x,ldx,ferr,berr,work,rwork,info)

```

### Discussion

This routine performs an iterative refinement of the solution to a system of linear equations  $AX = B$  with a symmetric (Hermitian) positive definite band matrix  $A$ , with multiple right-hand sides. For each computed solution vector  $x$ , the routine computes the *component-wise backward error*  $\beta$ . This error is the smallest relative perturbation in elements of  $A$  and  $b$  such that  $x$  is the exact solution of the perturbed system:

$$|\delta a_{ij}|/|a_{ij}| \leq \beta |a_{ij}|, \quad |\delta b_i|/|b_i| \leq \beta |b_i| \text{ such that } (A + \delta A)x = (b + \delta b).$$

Finally, the routine estimates the *component-wise forward error* in the computed solution  $\|x - x_e\|_\infty/\|x\|_\infty$  (here  $x_e$  is the exact solution).

Before calling this routine:

- call the factorization routine [?pbtrf](#)
- call the solver routine [?pbtrs](#).

### Input Parameters

*uplo*                    CHARACTER\*1. Must be '**U**' or '**L**'.

Indicates how the input matrix  $A$  has been factored:

If  $\text{uplo} = \text{'U'}$ , the array  $\text{afb}$  stores the factor  $U$  of the Cholesky factorization  $A = U^H U$ .

If  $\text{uplo} = \text{'L'}$ , the array  $\text{afb}$  stores the factor  $L$  of the Cholesky factorization  $A = LL^H$ .

$n$	<b>INTEGER.</b> The order of the matrix $A$ ( $n \geq 0$ ).
$kd$	<b>INTEGER.</b> The number of super-diagonals or sub-diagonals in the matrix $A$ ( $kd \geq 0$ ).
$nrhs$	<b>INTEGER.</b> The number of right-hand sides ( $nrhs \geq 0$ ).
$ab, afb, b, x, work$	<b>REAL</b> for <code>spbrfs</code> <b>DOUBLE PRECISION</b> for <code>dpbrfs</code> <b>COMPLEX</b> for <code>cpbrfs</code> <b>DOUBLE COMPLEX</b> for <code>zpbrfs</code> .

Arrays:

$ab(1dab, *)$  contains the original band matrix  $A$ , as supplied to `?pbtrf`.

$afb(1dafb, *)$  contains the factored band matrix  $A$ , as returned by `?pbtrf`.

$b(1db, *)$  contains the right-hand side matrix  $B$ .

$x(1dx, *)$  contains the solution matrix  $X$ .

$work (*)$  is a workspace array.

The second dimension of  $ab$  and  $afb$  must be at least  $\max(1, n)$ ; the second dimension of  $b$  and  $x$  must be at least  $\max(1, nrhs)$ ; the dimension of  $work$  must be at least  $\max(1, 3 * n)$  for real flavors and  $\max(1, 2 * n)$  for complex flavors.

$1dab$	<b>INTEGER.</b> The first dimension of $ab$ ; $1dab \geq kd + 1$ .
$1dafb$	<b>INTEGER.</b> The first dimension of $afb$ ; $1dafb \geq kd + 1$ .
$1db$	<b>INTEGER.</b> The first dimension of $b$ ; $1db \geq \max(1, n)$ .
$1dx$	<b>INTEGER.</b> The first dimension of $x$ ; $1dx \geq \max(1, n)$ .
$iwork$	<b>INTEGER.</b> Workspace array, <b>DIMENSION</b> at least $\max(1, n)$ .
$rwork$	<b>REAL</b> for <code>cpbrfs</code> <b>DOUBLE PRECISION</b> for <code>zpbrfs</code> Workspace array, <b>DIMENSION</b> at least $\max(1, n)$ .

## Output Parameters

<code>x</code>	The refined solution matrix $X$ .
<code>ferr, berr</code>	<code>REAL</code> for single precision flavors. <code>DOUBLE PRECISION</code> for double precision flavors.
	Arrays, <code>DIMENSION</code> at least $\max(1, nrhs)$ . Contain the component-wise forward and backward errors, respectively, for each solution vector.
<code>info</code>	<code>INTEGER</code> . If <code>info</code> = 0, the execution is successful. If <code>info</code> = $-i$ , the $i$ th parameter had an illegal value.

## Application Notes

The bounds returned in `ferr` are not rigorous, but in practice they almost always overestimate the actual error.

For each right-hand side, computation of the backward error involves a minimum of  $8n * kd$  floating-point operations (for real flavors) or  $32n * kd$  operations (for complex flavors). In addition, each step of iterative refinement involves  $12n * kd$  operations (for real flavors) or  $48n * kd$  operations (for complex flavors); the number of iterations may range from 1 to 5.

Estimating the forward error involves solving a number of systems of linear equations  $Ax = b$ ; the number is usually 4 or 5 and never more than 11. Each solution requires approximately  $4n * kd$  floating-point operations for real flavors or  $16n * kd$  for complex flavors.

## ?ptrfs

*Refines the solution of a system of linear equations with a symmetric (Hermitian) positive-definite tridiagonal matrix and estimates its error.*

```
call sptrfs (n,nrhs,d,e,df,ef,b,ldb,x,ldx,ferr,berr,work,info)
call dptrfs (n,nrhs,d,e,df,ef,b,ldb,x,ldx,ferr,berr,work,info)
call cptrfs (uplo,n,nrhs,d,e,df,ef,b,ldb,x,ldx,ferr,berr,
            work,rwork,info)
call cpqrfs (uplo,n,nrhs,d,e,df,ef,b,ldb,x,ldx,ferr,berr,
            work,rwork,info)
```

### Discussion

This routine performs an iterative refinement of the solution to a system of linear equations  $AX = B$  with a symmetric (Hermitian) positive definite tridiagonal matrix  $A$ , with multiple right-hand sides. For each computed solution vector  $x$ , the routine computes the *component-wise backward error*  $\beta$ . This error is the smallest relative perturbation in elements of  $A$  and  $b$  such that  $x$  is the exact solution of the perturbed system:

$$|\delta a_{ij}|/|a_{ij}| \leq \beta |a_{ij}|, \quad |\delta b_i|/|b_i| \leq \beta |b_i| \text{ such that } (A + \delta A)x = (b + \delta b).$$

Finally, the routine estimates the *component-wise forward error* in the computed solution  $\|x - x_e\|_\infty / \|x\|_\infty$  (here  $x_e$  is the exact solution).

Before calling this routine:

- call the factorization routine [?pttrf](#)
- call the solver routine [?ptrs](#).

### Input Parameters

<i>uplo</i>	<b>CHARACTER*1.</b> Used for complex flavors only. Must be ' <b>U</b> ' or ' <b>L</b> '. Specifies whether the superdiagonal or the subdiagonal of the tridiagonal matrix $A$ is stored and how $A$ is factored:
-------------	--

If  $\text{uplo} = \text{'U'}$ , the array  $e$  stores the superdiagonal of  $A$ , and  $A$  is factored as  $U^H D U$ ;

If  $\text{uplo} = \text{'L'}$ , the array  $e$  stores the subdiagonal of  $A$ , and  $A$  is factored as  $L D L^H$ .

*n*

**INTEGER.** The order of the matrix  $A$  ( $n \geq 0$ ).

*nrhs*

**INTEGER.** The number of right-hand sides ( $nrhs \geq 0$ ).

*d, df, rwork*

**REAL** for single precision flavors

**DOUBLE PRECISION** for double precision flavors

Arrays:  $d(n)$ ,  $df(n)$ ,  $rwork(n)$ .

The array  $d$  contains the  $n$  diagonal elements of the tridiagonal matrix  $A$ .

The array  $df$  contains the  $n$  diagonal elements of the diagonal matrix  $D$  from the factorization of  $A$  as computed by [?ptrf](#).

The array  $rwork$  is a workspace array used for complex flavors only.

*e, ef, b, x, work*

**REAL** for [sptrfs](#)

**DOUBLE PRECISION** for [dptrfs](#)

**COMPLEX** for [cptrfs](#)

**DOUBLE COMPLEX** for [zptrfs](#).

Arrays:  $e(n - 1)$ ,  $ef(n - 1)$ ,  $b(ldb, nrhs)$ ,  $x(lidx, nrhs)$ ,  $work(*)$ .

The array  $e$  contains the  $(n - 1)$  off-diagonal elements of the tridiagonal matrix  $A$  (see  $\text{uplo}$ ).

The array  $ef$  contains the  $(n - 1)$  off-diagonal elements of the unit bidiagonal factor  $U$  or  $L$  from the factorization computed by [?ptrf](#) (see  $\text{uplo}$ ).

The array  $b$  contains the matrix  $B$  whose columns are the right-hand sides for the systems of equations.

The array  $x$  contains the solution matrix  $X$  as computed by [?ptrs](#).

The array  $work$  is a workspace array. The dimension of  $work$  must be at least  $2 * n$  for real flavors, and at least  $n$  for complex flavors.

*ldb*

**INTEGER.** The leading dimension of  $b$ ;  $ldb \geq \max(1, n)$ .

*lidx*

**INTEGER.** The leading dimension of  $x$ ;  $lidx \geq \max(1, n)$ .

## Output Parameters

<i>x</i>	The refined solution matrix $X$ .
<i>ferr</i> , <i>berr</i>	<b>REAL</b> for single precision flavors. <b>DOUBLE PRECISION</b> for double precision flavors.
	Arrays, <b>DIMENSION</b> at least $\max(1, nrhs)$ . Contain the component-wise forward and backward errors, respectively, for each solution vector.
<i>info</i>	<b>INTEGER</b> . If <i>info</i> = 0, the execution is successful. If <i>info</i> = $-i$ , the <i>i</i> th parameter had an illegal value.

## ?syrf<sub>s</sub>

Refines the solution of a system of linear equations with a symmetric matrix and estimates its error.

```
call ssyrfs (uplo,n,nrhs,a,lda,af,ldaf,ipiv,b,ldb,
               x,ldx,ferr,berr,work,iwork,info)
call dsyrfs (uplo,n,nrhs,a,lda,af,ldaf,ipiv,b,ldb,
               x,ldx,ferr,berr,work,iwork,info)

call csyrfs (uplo,n,nrhs,a,lda,af,ldaf,ipiv,b,ldb,
               x,ldx,ferr,berr,work,rwork,info)
call zsyrfs (uplo,n,nrhs,a,lda,af,ldaf,ipiv,b,ldb,
               x,ldx,ferr,berr,work,rwork,info)
```

### Discussion

This routine performs an iterative refinement of the solution to a system of linear equations  $AX = B$  with a symmetric full-storage matrix  $A$ , with multiple right-hand sides. For each computed solution vector  $x$ , the routine computes the *component-wise backward error*  $\beta$ . This error is the smallest relative perturbation in elements of  $A$  and  $b$  such that  $x$  is the exact solution of the perturbed system:

$$|\delta a_{ij}|/|a_{ij}| \leq \beta |a_{ij}|, \quad |\delta b_i|/|b_i| \leq \beta |b_i| \text{ such that } (A + \delta A)x = (b + \delta b).$$

Finally, the routine estimates the *component-wise forward error* in the computed solution  $\|x - x_e\|_\infty/\|x\|_\infty$  (here  $x_e$  is the exact solution).

Before calling this routine:

- call the factorization routine [?sytrf](#)
- call the solver routine [?sytrs](#).

### Input Parameters

*uplo*                    CHARACTER\*1. Must be '**U**' or '**L**'.  
                           Indicates how the input matrix  $A$  has been factored:

If  $\text{uplo} = \text{'U'}$ , the array  $\text{af}$  stores the Bunch-Kaufman factorization  $A = PUDU^TP^T$ .

If  $\text{uplo} = \text{'L'}$ , the array  $\text{af}$  stores the Bunch-Kaufman factorization  $A = PLDL^TP^T$ .

$n$	<code>INTEGER</code> . The order of the matrix $A$ ( $n \geq 0$ ).
$nrhs$	<code>INTEGER</code> . The number of right-hand sides ( $nrhs \geq 0$ ).
$a, af, b, x, work$	<code>REAL</code> for <code>ssyrf</code> <code>DOUBLE PRECISION</code> for <code>dsyrf</code> <code>COMPLEX</code> for <code>csyrf</code> <code>DOUBLE COMPLEX</code> for <code>zsyrf</code> .

Arrays:

$a(\text{lda}, *)$  contains the original matrix  $A$ , as supplied to [?sytrf](#).

$af(\text{ldaf}, *)$  contains the factored matrix  $A$ , as returned by [?sytrf](#).

$b(\text{ldb}, *)$  contains the right-hand side matrix  $B$ .

$x(\text{idx}, *)$  contains the solution matrix  $X$ .

$work (*)$  is a workspace array.

The second dimension of  $a$  and  $af$  must be at least  $\max(1, n)$ ; the second dimension of  $b$  and  $x$  must be at least  $\max(1, nrhs)$ ; the dimension of  $work$  must be at least  $\max(1, 3 * n)$  for real flavors and  $\max(1, 2 * n)$  for complex flavors.

$lda$	<code>INTEGER</code> . The first dimension of $a$ ; $lda \geq \max(1, n)$ .
$ldaf$	<code>INTEGER</code> . The first dimension of $af$ ; $ldaf \geq \max(1, n)$ .
$ldb$	<code>INTEGER</code> . The first dimension of $b$ ; $ldb \geq \max(1, n)$ .
$idx$	<code>INTEGER</code> . The first dimension of $x$ ; $idx \geq \max(1, n)$ .
$ipiv$	<code>INTEGER</code> . Array, <code>DIMENSION</code> at least $\max(1, n)$ . The $ipiv$ array, as returned by <a href="#">?sytrf</a> .
$iwork$	<code>INTEGER</code> . Workspace array, <code>DIMENSION</code> at least $\max(1, n)$ .
$rwork$	<code>REAL</code> for <code>csyrf</code> <code>DOUBLE PRECISION</code> for <code>zsyrf</code> . Workspace array, <code>DIMENSION</code> at least $\max(1, n)$ .

## Output Parameters

<i>x</i>	The refined solution matrix $X$ .
<i>ferr</i> , <i>berr</i>	<i>REAL</i> for single precision flavors. <i>DOUBLE PRECISION</i> for double precision flavors.
	Arrays, <i>DIMENSION</i> at least $\max(1, \text{nrhs})$ . Contain the component-wise forward and backward errors, respectively, for each solution vector.
<i>info</i>	<i>INTEGER</i> . If <i>info</i> = 0, the execution is successful. If <i>info</i> = $-i$ , the <i>i</i> th parameter had an illegal value.

## Application Notes

The bounds returned in *ferr* are not rigorous, but in practice they almost always overestimate the actual error.

For each right-hand side, computation of the backward error involves a minimum of  $4n^2$  floating-point operations (for real flavors) or  $16n^2$  operations (for complex flavors). In addition, each step of iterative refinement involves  $6n^2$  operations (for real flavors) or  $24n^2$  operations (for complex flavors); the number of iterations may range from 1 to 5.

Estimating the forward error involves solving a number of systems of linear equations  $Ax = b$ ; the number is usually 4 or 5 and never more than 11. Each solution requires approximately  $2n^2$  floating-point operations for real flavors or  $8n^2$  for complex flavors.

## ?herfs

*Refines the solution of a system of linear equations with a complex Hermitian matrix and estimates its error.*

---

```
call cherfs (uplo,n,nrhs,a,lda,af,ldaf,ipiv,b,ldb,
             x,ldx,ferr,berr,work,rwork,info)
call zherfs (uplo,n,nrhs,a,lda,af,ldaf,ipiv,b,ldb,
             x,ldx,ferr,berr,work,rwork,info)
```

### Discussion

This routine performs an iterative refinement of the solution to a system of linear equations  $AX = B$  with a complex Hermitian full-storage matrix  $A$ , with multiple right-hand sides. For each computed solution vector  $x$ , the routine computes the *component-wise backward error*  $\beta$ . This error is the smallest relative perturbation in elements of  $A$  and  $b$  such that  $x$  is the exact solution of the perturbed system:

$$|\delta a_{ij}|/|a_{ij}| \leq \beta |a_{ij}|, \quad |\delta b_i|/|b_i| \leq \beta |b_i| \text{ such that } (A + \delta A)x = (b + \delta b).$$

Finally, the routine estimates the *component-wise forward error* in the computed solution  $\|x - x_e\|_\infty / \|x\|_\infty$  (here  $x_e$  is the exact solution).

Before calling this routine:

- call the factorization routine [?hetrf](#)
- call the solver routine [?hetrs](#).

### Input Parameters

<i>uplo</i>	CHARACTER*1. Must be ' <b>U</b> ' or ' <b>L</b> '.
	Indicates how the input matrix $A$ has been factored:
	If <i>uplo</i> = ' <b>U</b> ', the array <i>af</i> stores the Bunch-Kaufman factorization $A = PUDU^HPT$ .
	If <i>uplo</i> = ' <b>L</b> ', the array <i>af</i> stores the Bunch-Kaufman factorization $A = PLDL^HPT$ .

*n* INTEGER. The order of the matrix  $A$  ( $n \geq 0$ ).

*nrhs*                    INTEGER. The number of right-hand sides (*nrhs*  $\geq 0$ ).

*a, af, b, x, work*    COMPLEX for *cherfs*  
                           DOUBLE COMPLEX for *zherfs*.

Arrays:

*a( lda, \* )* contains the original matrix *A*, as supplied  
                           to *?hetrf*.

*af( ldaf, \* )* contains the factored matrix *A*, as returned  
                           by *?hetrf*.

*b( ldb, \* )* contains the right-hand side matrix *B*.

*x( idx, \* )* contains the solution matrix *X*.

*work ( \* )* is a workspace array.

The second dimension of *a* and *af* must be at least  
 $\max(1, n)$ ; the second dimension of *b* and *x* must be at  
                           least  $\max(1, \text{nrhs})$ ; the dimension of *work* must be at  
                           least  $\max(1, 2 * n)$ .

*lda*                    INTEGER. The first dimension of *a*; *lda*  $\geq \max(1, n)$ .

*ldaf*                    INTEGER. The first dimension of *af*; *ldaf*  $\geq \max(1, n)$ .

*ldb*                    INTEGER. The first dimension of *b*; *ldb*  $\geq \max(1, n)$ .

*idx*                    INTEGER. The first dimension of *x*; *idx*  $\geq \max(1, n)$ .

*ipiv*                    INTEGER.

Array, DIMENSION at least  $\max(1, n)$ .

The *ipiv* array, as returned by *?hetrf*.

*rwork*                REAL for *cherfs*

DOUBLE PRECISION for *zherfs*.

Workspace array, DIMENSION at least  $\max(1, n)$ .

## Output Parameters

<i>x</i>	The refined solution matrix $X$ .
<i>ferr</i> , <i>berr</i>	<i>REAL</i> for <i>cherf</i> s <i>DOUBLE PRECISION</i> for <i>zherf</i> s. Arrays, <i>DIMENSION</i> at least $\max(1, \text{n} \times \text{rhs})$ . Contain the component-wise forward and backward errors, respectively, for each solution vector.
<i>info</i>	<i>INTEGER</i> . If <i>info</i> = 0, the execution is successful. If <i>info</i> = $-i$ , the <i>i</i> th parameter had an illegal value.

## Application Notes

The bounds returned in *ferr* are not rigorous, but in practice they almost always overestimate the actual error.

For each right-hand side, computation of the backward error involves a minimum of  $16n^2$  operations. In addition, each step of iterative refinement involves  $24n^2$  operations; the number of iterations may range from 1 to 5.

Estimating the forward error involves solving a number of systems of linear equations  $Ax = b$ ; the number is usually 4 or 5 and never more than 11. Each solution requires approximately  $8n^2$  floating-point operations.

The real counterpart of this routine is *ssyrf*s / *dsyrf*s.

## ?sprfs

*Refines the solution of a system of linear equations with a packed symmetric matrix and estimates the solution error.*

```
call ssprfs (uplo,n,nrhs,ap,afp,ipiv,b,ldb,x,ldx,
             ferr,berr,work,iwork,info)
call dsprfs (uplo,n,nrhs,ap,afp,ipiv,b,ldb,x,ldx,
             ferr,berr,work,iwork,info)

call csprfs (uplo,n,nrhs,ap,afp,ipiv,b,ldb,x,ldx,
             ferr,berr,work,rwork,info)
call zsprfs (uplo,n,nrhs,ap,afp,ipiv,b,ldb,x,ldx,
             ferr,berr,work,rwork,info)
```

### Discussion

This routine performs an iterative refinement of the solution to a system of linear equations  $AX = B$  with a packed symmetric matrix  $A$ , with multiple right-hand sides. For each computed solution vector  $x$ , the routine computes the *component-wise backward error*  $\beta$ . This error is the smallest relative perturbation in elements of  $A$  and  $b$  such that  $x$  is the exact solution of the perturbed system:

$$|\delta a_{ij}|/|a_{ij}| \leq \beta |a_{ij}|, \quad |\delta b_i|/|b_i| \leq \beta |b_i| \text{ such that } (A + \delta A)x = (b + \delta b).$$

Finally, the routine estimates the *component-wise forward error* in the computed solution  $\|x - x_e\|_\infty/\|x\|_\infty$  (here  $x_e$  is the exact solution).

Before calling this routine:

- call the factorization routine [?sptrf](#)
- call the solver routine [?sptrs](#).

### Input Parameters

*uplo*                    CHARACTER\*1. Must be '**U**' or '**L**'.  
 Indicates how the input matrix  $A$  has been factored:

If  $\text{uplo} = \text{'U'}$ , the array  $\text{afp}$  stores the packed Bunch-Kaufman factorization  $A = PUDU^TP^T$ .

If  $\text{uplo} = \text{'L'}$ , the array  $\text{afp}$  stores the packed Bunch-Kaufman factorization  $A = PLDL^TP^T$ .

$n$  INTEGER. The order of the matrix  $A$  ( $n \geq 0$ ).  
 $nrhs$  INTEGER. The number of right-hand sides ( $nrhs \geq 0$ ).  
 $ap, afp, b, x, work$  REAL for ssprfs  
 DOUBLE PRECISION for dsprfs  
 COMPLEX for csprfs  
 DOUBLE COMPLEX for zsprfs.

Arrays:

$ap(*)$  contains the original packed matrix  $A$ , as supplied to [?sptrf](#).

$afp(*)$  contains the factored packed matrix  $A$ , as returned by [?sptrf](#).

$b(ldb, *)$  contains the right-hand side matrix  $B$ .

$x(idx, *)$  contains the solution matrix  $X$ .

$work(*)$  is a workspace array.

The dimension of arrays  $ap$  and  $afp$  must be at least  $\max(1, n(n+1)/2)$ ; the second dimension of  $b$  and  $x$  must be at least  $\max(1, nrhs)$ ; the dimension of  $work$  must be at least  $\max(1, 3*n)$  for real flavors and  $\max(1, 2*n)$  for complex flavors..

$ldb$  INTEGER. The first dimension of  $b$ ;  $ldb \geq \max(1, n)$ .  
 $idx$  INTEGER. The first dimension of  $x$ ;  $idx \geq \max(1, n)$ .  
 $ipiv$  INTEGER.  
 Array, DIMENSION at least  $\max(1, n)$ .  
 The  $ipiv$  array, as returned by [?sptrf](#).  
 $iwork$  INTEGER.  
 Workspace array, DIMENSION at least  $\max(1, n)$ .  
 $rwork$  REAL for csprfs  
 DOUBLE PRECISION for zsprfs  
 Workspace array, DIMENSION at least  $\max(1, n)$ .

## Output Parameters

<i>x</i>	The refined solution matrix $X$ .
<i>ferr</i> , <i>berr</i>	<i>REAL</i> for single precision flavors. <i>DOUBLE PRECISION</i> for double precision flavors.
	Arrays, <i>DIMENSION</i> at least $\max(1, \text{nrhs})$ . Contain the component-wise forward and backward errors, respectively, for each solution vector.
<i>info</i>	<i>INTEGER</i> . If <i>info</i> = 0, the execution is successful. If <i>info</i> = $-i$ , the <i>i</i> th parameter had an illegal value.

## Application Notes

The bounds returned in *ferr* are not rigorous, but in practice they almost always overestimate the actual error.

For each right-hand side, computation of the backward error involves a minimum of  $4n^2$  floating-point operations (for real flavors) or  $16n^2$  operations (for complex flavors). In addition, each step of iterative refinement involves  $6n^2$  operations (for real flavors) or  $24n^2$  operations (for complex flavors); the number of iterations may range from 1 to 5.

Estimating the forward error involves solving a number of systems of linear equations  $Ax = b$ ; the number of systems is usually 4 or 5 and never more than 11. Each solution requires approximately  $2n^2$  floating-point operations for real flavors or  $8n^2$  for complex flavors.

## ?hprfs

*Refines the solution of a system of linear equations with a packed complex Hermitian matrix and estimates the solution error.*

---

```
call chprfs (uplo,n,nrhs,ap,afp,ipiv,b,ldb,x,ldx,
             ferr,berr,work,rwork,info)
call zhprfs (uplo,n,nrhs,ap,afp,ipiv,b,ldb,x,ldx,
             ferr,berr,work,rwork,info)
```

### Discussion

This routine performs an iterative refinement of the solution to a system of linear equations  $AX = B$  with a packed complex Hermitian matrix  $A$ , with multiple right-hand sides. For each computed solution vector  $x$ , the routine computes the *component-wise backward error*  $\beta$ . This error is the smallest relative perturbation in elements of  $A$  and  $b$  such that  $x$  is the exact solution of the perturbed system:

$$|\delta a_{ij}|/|a_{ij}| \leq \beta |a_{ij}|, \quad |\delta b_i|/|b_i| \leq \beta |b_i| \text{ such that } (A + \delta A)x = (b + \delta b).$$

Finally, the routine estimates the *component-wise forward error* in the computed solution  $\|x - x_e\|_\infty/\|x\|_\infty$  (here  $x_e$  is the exact solution).

Before calling this routine:

- call the factorization routine [?hptrf](#)
- call the solver routine [?hptrs](#).

### Input Parameters

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'.
	Indicates how the input matrix $A$ has been factored: If <i>uplo</i> = 'U', the array <i>afp</i> stores the packed Bunch-Kaufman factorization $A = PUDU^HPT$ . If <i>uplo</i> = 'L', the array <i>afp</i> stores the packed Bunch-Kaufman factorization $A = PLDL^HPT$ .
<i>n</i>	INTEGER. The order of the matrix $A$ ( $n \geq 0$ ).
<i>nrhs</i>	INTEGER. The number of right-hand sides ( $nrhs \geq 0$ ).

---

*ap, afp, b, x, work* COMPLEX for **chprfs**  
DOUBLE COMPLEX for **zhprfs**.

Arrays:

*ap*(\*) contains the original packed matrix *A*, as supplied to [?hptrf](#).

*afp*(\*) contains the factored packed matrix *A*, as returned by [?hptrf](#).

*b*(*ldb*,\*) contains the right-hand side matrix *B*.

*x*(*lidx*,\*) contains the solution matrix *X*.

*work* (\*) is a workspace array.

The dimension of arrays *ap* and *afp* must be at least  $\max(1, n(n+1)/2)$ ; the second dimension of *b* and *x* must be at least  $\max(1, nrhs)$ ; the dimension of *work* must be at least  $\max(1, 2 * n)$ .

<i>ldb</i>	INTEGER. The first dimension of <i>b</i> ; <i>ldb</i> $\geq \max(1, n)$ .
<i>lidx</i>	INTEGER. The first dimension of <i>x</i> ; <i>lidx</i> $\geq \max(1, n)$ .
<i>ipiv</i>	INTEGER. Array, DIMENSION at least $\max(1, n)$ . The <i>ipiv</i> array, as returned by <a href="#">?hptrf</a> .
<i>rwork</i>	REAL for <b>chprfs</b> DOUBLE PRECISION for <b>zhprfs</b> Workspace array, DIMENSION at least $\max(1, n)$ .

## Output Parameters

<i>x</i>	The refined solution matrix <i>X</i> .
<i>ferr, berr</i>	REAL for <b>chprfs</b> . DOUBLE PRECISION for <b>zhprfs</b> . Arrays, DIMENSION at least $\max(1, nrhs)$ . Contain the component-wise forward and backward errors, respectively, for each solution vector.
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = <i>-i</i> , the <i>i</i> th parameter had an illegal value.

## Application Notes

The bounds returned in `ferr` are not rigorous, but in practice they almost always overestimate the actual error.

For each right-hand side, computation of the backward error involves a minimum of  $16n^2$  operations. In addition, each step of iterative refinement involves  $24n^2$  operations; the number of iterations may range from 1 to 5.

Estimating the forward error involves solving a number of systems of linear equations  $Ax = b$ ; the number is usually 4 or 5 and never more than 11. Each solution requires approximately  $8n^2$  floating-point operations.

The real counterpart of this routine is `ssprfs` / `dsprfs`.

---

## ?trrfs

*Estimates the error in the solution of  
a system of linear equations with a  
triangular matrix.*

---

```

call strrfs (uplo,trans,diag,n,nrhs,a,lda,b,ldb,
             x,ldx,ferr,berr,work,iwork,info)
call dtrrfs (uplo,trans,diag,n,nrhs,a,lda,b,ldb,
             x,ldx,ferr,berr,work,iwork,info)

call ctrrfs (uplo,trans,diag,n,nrhs,a,lda,b,ldb,
             x,ldx,ferr,berr,work,rwork,info)
call ztrrfs (uplo,trans,diag,n,nrhs,a,lda,b,ldb,
             x,ldx,ferr,berr,work,rwork,info)

```

### Discussion

This routine estimates the errors in the solution to a system of linear equations  $AX = B$  or  $A^T X = B$  or  $A^H X = B$  with a triangular matrix  $A$ , with multiple right-hand sides. For each computed solution vector  $x$ , the routine computes the *component-wise backward error*  $\beta$ . This error is the smallest relative perturbation in elements of  $A$  and  $b$  such that  $x$  is the exact solution of the perturbed system:

$$|\delta a_{ij}|/|a_{ij}| \leq \beta |a_{ij}|, \quad |\delta b_i|/|b_i| \leq \beta |b_i| \text{ such that } (A + \delta A)x = (b + \delta b).$$

The routine also estimates the *component-wise forward error* in the computed solution  $\|x - x_e\|_\infty / \|x\|_\infty$  (here  $x_e$  is the exact solution).

Before calling this routine, call the solver routine [?trtrs](#).

### Input Parameters

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. Indicates whether $A$ is upper or lower triangular: If <i>uplo</i> = 'U', then $A$ is upper triangular. If <i>uplo</i> = 'L', then $A$ is lower triangular.
<i>trans</i>	CHARACTER*1. Must be 'N' or 'T' or 'C'. Indicates the form of the equations: If <i>trans</i> = 'N', the system has the form $AX = B$ . If <i>trans</i> = 'T', the system has the form $A^T X = B$ . If <i>trans</i> = 'C', the system has the form $A^H X = B$ .
<i>diag</i>	CHARACTER*1. Must be 'N' or 'U'. If <i>diag</i> = 'N', then $A$ is not a unit triangular matrix. If <i>diag</i> = 'U', then $A$ is unit triangular: diagonal elements of $A$ are assumed to be 1 and not referenced in the array <i>a</i> .
<i>n</i>	INTEGER. The order of the matrix $A$ ( <i>n</i> $\geq 0$ ).
<i>nrhs</i>	INTEGER. The number of right-hand sides ( <i>nrhs</i> $\geq 0$ ).
<i>a</i> , <i>b</i> , <i>x</i> , <i>work</i>	REAL for <i>strrfs</i> DOUBLE PRECISION for <i>dtrrfs</i> COMPLEX for <i>cstrrfs</i> DOUBLE COMPLEX for <i>zstrrfs</i> .
Arrays:	
<i>a</i> ( <i>lda</i> ,*)	
contains the upper or lower triangular matrix $A$ , as specified by <i>uplo</i> .	
<i>b</i> ( <i>ldb</i> ,*)	
contains the right-hand side matrix $B$ .	
<i>x</i> ( <i>ldx</i> ,*)	
contains the solution matrix $X$ .	
<i>work</i> (*)	
is a workspace array.	
The second dimension of <i>a</i> must be at least $\max(1, n)$ ;	
the second dimension of <i>b</i> and <i>x</i> must be at least $\max(1, nrhs)$ ;	
the dimension of <i>work</i> must be at least $\max(1, 3 * n)$ for real flavors and $\max(1, 2 * n)$ for complex flavors.	

<i>lda</i>	INTEGER. The first dimension of <i>a</i> ; <i>lda</i> $\geq \max(1, n)$ .
<i>ldb</i>	INTEGER. The first dimension of <i>b</i> ; <i>ldb</i> $\geq \max(1, n)$ .
<i>ldx</i>	INTEGER. The first dimension of <i>x</i> ; <i>ldx</i> $\geq \max(1, n)$ .
<i>iwork</i>	INTEGER.
	Workspace array, DIMENSION at least $\max(1, n)$ .
<i>rwork</i>	REAL for <i>cstrrfs</i> DOUBLE PRECISION for <i>zstrrfs</i> Workspace array, DIMENSION at least $\max(1, n)$ .

### Output Parameters

<i>ferr</i> , <i>berr</i>	REAL for single precision flavors. DOUBLE PRECISION for double precision flavors. Arrays, DIMENSION at least $\max(1, nrhs)$ . Contain the component-wise forward and backward errors, respectively, for each solution vector.
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = <i>-i</i> , the <i>i</i> th parameter had an illegal value.

### Application Notes

The bounds returned in *ferr* are not rigorous, but in practice they almost always overestimate the actual error.

A call to this routine involves, for each right-hand side, solving a number of systems of linear equations  $Ax = b$ ; the number of systems is usually 4 or 5 and never more than 11. Each solution requires approximately  $n^2$  floating-point operations for real flavors or  $4n^2$  for complex flavors.

## ?tprhs

*Estimates the error in the solution of  
a system of linear equations with a  
packed triangular matrix.*

```
call stprfs (uplo,trans,diag,n,nrhs,ap,b,ldb,
             x,ldx,ferr,berr,work,iwork,info)

call dtprhs (uplo,trans,diag,n,nrhs,ap,b,ldb,
             x,ldx,ferr,berr,work,iwork,info)

call ctprhs (uplo,trans,diag,n,nrhs,ap,b,ldb,
             x,ldx,ferr,berr,work,rwork,info)

call ztprhs (uplo,trans,diag,n,nrhs,ap,b,ldb,
             x,ldx,ferr,berr,work,rwork,info)
```

### Discussion

This routine estimates the errors in the solution to a system of linear equations  $AX = B$  or  $A^T X = B$  or  $A^H X = B$  with a packed triangular matrix  $A$ , with multiple right-hand sides. For each computed solution vector  $x$ , the routine computes the *component-wise backward error*  $\beta$ . This error is the smallest relative perturbation in elements of  $A$  and  $b$  such that  $x$  is the exact solution of the perturbed system:

$$|\delta a_{ij}|/|a_{ij}| \leq \beta |a_{ij}|, \quad |\delta b_i|/|b_i| \leq \beta |b_i| \text{ such that } (A + \delta A)x = (b + \delta b).$$

The routine also estimates the *component-wise forward error* in the computed solution  $\|x - x_e\|_\infty / \|x\|_\infty$  (here  $x_e$  is the exact solution).

Before calling this routine, call the solver routine [?tptrs](#).

## Input Parameters

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. Indicates whether $A$ is upper or lower triangular: If <i>uplo</i> = 'U', then $A$ is upper triangular. If <i>uplo</i> = 'L', then $A$ is lower triangular.
<i>trans</i>	CHARACTER*1. Must be 'N' or 'T' or 'C'. Indicates the form of the equations: If <i>trans</i> = 'N', the system has the form $AX = B$ . If <i>trans</i> = 'T', the system has the form $A^TX = B$ . If <i>trans</i> = 'C', the system has the form $A^HX = B$ .
<i>diag</i>	CHARACTER*1. Must be 'N' or 'U'. If <i>diag</i> = 'N', $A$ is not a unit triangular matrix. If <i>diag</i> = 'U', $A$ is unit triangular: diagonal elements of $A$ are assumed to be 1 and not referenced in the array <i>ap</i> .
<i>n</i>	INTEGER. The order of the matrix $A$ ( <i>n</i> $\geq$ 0).
<i>nrhs</i>	INTEGER. The number of right-hand sides ( <i>nrhs</i> $\geq$ 0).
<i>ap</i> , <i>b</i> , <i>x</i> , <i>work</i>	REAL for <i>strrfs</i> DOUBLE PRECISION for <i>dtrrfs</i> COMPLEX for <i>ctrdfs</i> DOUBLE COMPLEX for <i>ztrrfs</i> .
Arrays:	
<i>ap</i> (*)	contains the upper or lower triangular matrix $A$ , as specified by <i>uplo</i> .
<i>b</i> ( <i>ldb</i> ,*)	contains the right-hand side matrix $B$ .
<i>x</i> ( <i>ldx</i> ,*)	contains the solution matrix $X$ .
<i>work</i> (*)	is a workspace array. The dimension of <i>ap</i> must be at least $\max(1, n(n+1)/2)$ ; the second dimension of <i>b</i> and <i>x</i> must be at least $\max(1, nrhs)$ ; the dimension of <i>work</i> must be at least $\max(1, 3*n)$ for real flavors and $\max(1, 2*n)$ for complex flavors.
<i>ldb</i>	INTEGER. The first dimension of <i>b</i> ; <i>ldb</i> $\geq \max(1, n)$ .
<i>ldx</i>	INTEGER. The first dimension of <i>x</i> ; <i>ldx</i> $\geq \max(1, n)$ .
<i>iwork</i>	INTEGER. Workspace array, DIMENSION at least $\max(1, n)$ .

---

*rwork*      **REAL** for **cstrrfs**  
**DOUBLE PRECISION** for **zstrrfs**  
 Workspace array, **DIMENSION** at least  $\max(1, n)$ .

### Output Parameters

*ferr*, *berr*      **REAL** for single precision flavors.  
**DOUBLE PRECISION** for double precision flavors.  
 Arrays, **DIMENSION** at least  $\max(1, \text{nrhs})$ . Contain the component-wise forward and backward errors, respectively, for each solution vector.

*info*      **INTEGER**.  
 If *info* = 0, the execution is successful.  
 If *info* = *-i*, the *i*th parameter had an illegal value.

### Application Notes

The bounds returned in *ferr* are not rigorous, but in practice they almost always overestimate the actual error.

A call to this routine involves, for each right-hand side, solving a number of systems of linear equations  $Ax = b$ ; the number of systems is usually 4 or 5 and never more than 11. Each solution requires approximately  $n^2$  floating-point operations for real flavors or  $4n^2$  for complex flavors.

---

## ?tbrfs

*Estimates the error in the solution of  
a system of linear equations with a  
triangular band matrix.*

---

```
call stbrfs (uplo,trans,diag,n,kd,nrhs,ab,ldab,b,ldb,  
            x,ldx,ferr,berr,work,iwork,info)

call dtbrfs (uplo,trans,diag,n,kd,nrhs,ab,ldab,b,ldb,  
            x,ldx,ferr,berr,work,iwork,info)

call ctbrfs (uplo,trans,diag,n,kd,nrhs,ab,ldab,b,ldb,  
            x,ldx,ferr,berr,work,rwork,info)

call ztbrfs (uplo,trans,diag,n,kd,nrhs,ab,ldab,b,ldb,  
            x,ldx,ferr,berr,work,rwork,info)
```

### Discussion

This routine estimates the errors in the solution to a system of linear equations  $AX = B$  or  $A^T X = B$  or  $A^H X = B$  with a triangular band matrix  $A$ , with multiple right-hand sides. For each computed solution vector  $x$ , the routine computes the *component-wise backward error*  $\beta$ . This error is the smallest relative perturbation in elements of  $A$  and  $b$  such that  $x$  is the exact solution of the perturbed system:

$$|\delta a_{ij}|/|a_{ij}| \leq \beta |a_{ij}|, \quad |\delta b_i|/|b_i| \leq \beta |b_i| \text{ such that } (A + \delta A)x = (b + \delta b).$$

The routine also estimates the *component-wise forward error* in the computed solution  $\|x - x_e\|_\infty / \|x\|_\infty$  (here  $x_e$  is the exact solution).

Before calling this routine, call the solver routine [?tbtrs](#).

## Input Parameters

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. Indicates whether $A$ is upper or lower triangular: If <i>uplo</i> = 'U', then $A$ is upper triangular. If <i>uplo</i> = 'L', then $A$ is lower triangular.
<i>trans</i>	CHARACTER*1. Must be 'N' or 'T' or 'C'. Indicates the form of the equations: If <i>trans</i> = 'N', the system has the form $AX = B$ . If <i>trans</i> = 'T', the system has the form $A^T X = B$ . If <i>trans</i> = 'C', the system has the form $A^H X = B$ .
<i>diag</i>	CHARACTER*1. Must be 'N' or 'U'. If <i>diag</i> = 'N', $A$ is not a unit triangular matrix. If <i>diag</i> = 'U', $A$ is unit triangular: diagonal elements of $A$ are assumed to be 1 and not referenced in the array <i>ab</i> .
<i>n</i>	INTEGER. The order of the matrix $A$ ( <i>n</i> $\geq$ 0).
<i>kd</i>	INTEGER. The number of super-diagonals or sub-diagonals in the matrix $A$ ( <i>kd</i> $\geq$ 0).
<i>nrhs</i>	INTEGER. The number of right-hand sides ( <i>nrhs</i> $\geq$ 0).
<i>ab</i> , <i>b</i> , <i>x</i> , <i>work</i>	REAL for <i>stbrfs</i> DOUBLE PRECISION for <i>dtbrfs</i> COMPLEX for <i>ctbrfs</i> DOUBLE COMPLEX for <i>ztbrfs</i> .
Arrays:	
	<i>ab</i> ( <i>ldab</i> , *) contains the upper or lower triangular matrix $A$ , as specified by <i>uplo</i> , in band storage format.
	<i>b</i> ( <i>ldb</i> , *) contains the right-hand side matrix $B$ .
	<i>x</i> ( <i>idx</i> , *) contains the solution matrix $X$ .
	<i>work</i> (*) is a workspace array.
	The second dimension of <i>a</i> must be at least $\max(1, n)$ ; the second dimension of <i>b</i> and <i>x</i> must be at least $\max(1, nrhs)$ .
	The dimension of <i>work</i> must be at least $\max(1, 3 * n)$ for real flavors and $\max(1, 2 * n)$ for complex flavors.
<i>ldab</i>	INTEGER. The first dimension of the array <i>ab</i> . ( <i>ldab</i> $\geq$ <i>kd</i> + 1).

<i>ldb</i>	INTEGER. The first dimension of <i>b</i> ; $ldb \geq \max(1, n)$ .
<i>ldx</i>	INTEGER. The first dimension of <i>x</i> ; $ldx \geq \max(1, n)$ .
<i>iwork</i>	INTEGER.
	Workspace array, DIMENSION at least $\max(1, n)$ .
<i>rwork</i>	REAL for <i>ctbrfs</i> DOUBLE PRECISION for <i>ztbrfs</i> Workspace array, DIMENSION at least $\max(1, n)$ .

## Output Parameters

<i>ferr</i> , <i>berr</i>	REAL for single precision flavors. DOUBLE PRECISION for double precision flavors. Arrays, DIMENSION at least $\max(1, nrhs)$ . Contain the component-wise forward and backward errors, respectively, for each solution vector.
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = <i>-i</i> , the <i>i</i> th parameter had an illegal value.

## Application Notes

The bounds returned in *ferr* are not rigorous, but in practice they almost always overestimate the actual error.

A call to this routine involves, for each right-hand side, solving a number of systems of linear equations  $Ax = b$ ; the number of systems is usually 4 or 5 and never more than 11. Each solution requires approximately  $2n * kd$  floating-point operations for real flavors or  $8n * kd$  operations for complex flavors.

## Routines for Matrix Inversion

It is seldom necessary to compute an explicit inverse of a matrix.

In particular, do not attempt to solve a system of equations  $Ax = b$  by first computing  $A^{-1}$  and then forming the matrix-vector product  $x = A^{-1}b$ .

Call a solver routine instead (see [Routines for Solving Systems of Linear Equations](#)); this is more efficient and more accurate.

However, matrix inversion routines are provided for the rare occasions when an explicit inverse matrix is needed.

### ?getri

*Computes the inverse of an LU-factored general matrix.*

```
call sgetri (n, a, lda, ipiv, work, lwork, info)
call dgetri (n, a, lda, ipiv, work, lwork, info)
call cgetri (n, a, lda, ipiv, work, lwork, info)
call zgetri (n, a, lda, ipiv, work, lwork, info)
```

#### Discussion

This routine computes the inverse ( $A^{-1}$ ) of a general matrix  $A$ .

Before calling this routine, call [?geqrf](#) to factorize  $A$ .

#### Input Parameters

<i>n</i>	<b>INTEGER.</b> The order of the matrix $A$ ( $n \geq 0$ ).
<i>a, work</i>	<b>REAL</b> for <i>sgetri</i> <b>DOUBLE PRECISION</b> for <i>dgetri</i> <b>COMPLEX</b> for <i>cgetri</i> <b>DOUBLE COMPLEX</b> for <i>zgetri</i> . Arrays: <i>a(lda,*)</i> , <i>work(lwork)</i> . <i>a(lda,*)</i> contains the factorization of the matrix $A$ , as returned by <a href="#">?geqrf</a> : $A = PLU$ . The second dimension of <i>a</i> must be at least $\max(1,n)$ . <i>work(lwork)</i> is a workspace array.

<i>lda</i>	<b>INTEGER.</b> The first dimension of <i>a</i> ; <i>lda</i> $\geq \max(1, n)$ .
<i>ipiv</i>	<b>INTEGER.</b> Array, <b>DIMENSION</b> at least $\max(1, n)$ . The <i>ipiv</i> array, as returned by <a href="#">?getrf</a> .
<i>lwork</i>	<b>INTEGER.</b> The size of the <i>work</i> array ( <i>lwork</i> $\geq n$ ) See <i>Application notes</i> for the suggested value of <i>lwork</i> .

## Output Parameters

<i>a</i>	Overwritten by the <i>n</i> by <i>n</i> matrix $A^{-1}$ .
<i>work(1)</i>	If <i>info</i> = 0, on exit <i>work(1)</i> contains the minimum value of <i>lwork</i> required for optimum performance. Use this <i>lwork</i> for subsequent runs.
<i>info</i>	<b>INTEGER.</b> If <i>info</i> = 0, the execution is successful. If <i>info</i> = <i>-i</i> , the <i>i</i> th parameter had an illegal value. If <i>info</i> = <i>i</i> , the <i>i</i> th diagonal element of the factor <i>U</i> is zero, <i>U</i> is singular, and the inversion could not be completed.

## Application Notes

For better performance, try using *lwork* = *n*\**blocksize*, where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run. On exit, examine *work(1)* and use this value for subsequent runs.

The computed inverse *X* satisfies the following error bound:

$$|XA - I| \leq c(n)\epsilon|X||P||L||U|$$

where  $c(n)$  is a modest linear function of *n*;  $\epsilon$  is the machine precision; *I* denotes the identity matrix; *P*, *L*, and *U* are the factors of the matrix factorization  $A = PLU$ .

The total number of floating-point operations is approximately  $(4/3)n^3$  for real flavors and  $(16/3)n^3$  for complex flavors.

## ?potri

Computes the inverse of a symmetric (Hermitian) positive-definite matrix.

```
call spotri (uplo, n, a, lda, info)
call dpotri (uplo, n, a, lda, info)
call cpotri (uplo, n, a, lda, info)
call zpotri (uplo, n, a, lda, info)
```

### Discussion

This routine computes the inverse ( $A^{-1}$ ) of a symmetric positive definite or, for complex flavors, Hermitian positive-definite matrix  $A$ .

Before calling this routine, call [?potrf](#) to factorize  $A$ .

### Input Parameters

<i>uplo</i>	CHARACTER*1. Must be ' <b>U</b> ' or ' <b>L</b> '.
	Indicates how the input matrix $A$ has been factored:
	If <i>uplo</i> = ' <b>U</b> ', the array <i>a</i> stores the factor $U$ of the Cholesky factorization $A = U^H U$ .
	If <i>uplo</i> = ' <b>L</b> ', the array <i>a</i> stores the factor $L$ of the Cholesky factorization $A = LL^H$ .
<i>n</i>	INTEGER. The order of the matrix $A$ ( <i>n</i> $\geq$ 0).
<i>a</i>	REAL for <b>spotri</b> DOUBLE PRECISION for <b>dpotri</b> COMPLEX for <b>cpotri</b> DOUBLE COMPLEX for <b>zpotri</b> . Array: <i>a</i> ( <i>lda</i> , *).
	Contains the factorization of the matrix $A$ , as returned by <a href="#">?potrf</a> .
	The second dimension of <i>a</i> must be at least $\max(1, n)$ .
<i>lda</i>	INTEGER. The first dimension of <i>a</i> ; <i>lda</i> $\geq \max(1, n)$ .

## Output Parameters

<i>a</i>	Overwritten by the <i>n</i> by <i>n</i> matrix $A^{-1}$ .
<i>info</i>	<i>INTEGER</i> . If <i>info</i> = 0, the execution is successful. If <i>info</i> = <i>-i</i> , the <i>i</i> th parameter had an illegal value. If <i>info</i> = <i>i</i> , the <i>i</i> th diagonal element of the Cholesky factor (and hence the factor itself) is zero, and the inversion could not be completed.

## Application Notes

The computed inverse  $X$  satisfies the following error bounds:

$$\|XA - I\|_2 \leq c(\textcolor{red}{n}) \varepsilon \kappa_2(A), \quad \|AX - I\|_2 \leq c(\textcolor{red}{n}) \varepsilon \kappa_2(A)$$

where  $c(\textcolor{red}{n})$  is a modest linear function of *n*, and  $\varepsilon$  is the machine precision;  $I$  denotes the identity matrix.

The 2-norm  $\|A\|_2$  of a matrix  $A$  is defined by  $\|A\|_2 = \max_{x:x=1} (Ax \cdot Ax)^{1/2}$ , and the condition number  $\kappa_2(A)$  is defined by  $\kappa_2(A) = \|A\|_2 \|A^{-1}\|_2$ .

The total number of floating-point operations is approximately  $(2/3)\textcolor{red}{n}^3$  for real flavors and  $(8/3)\textcolor{red}{n}^3$  for complex flavors.

## ?pptri

*Computes the inverse of a packed symmetric (Hermitian) positive-definite matrix*

```
call spptri (uplo, n, ap, info)
call dpptri (uplo, n, ap, info)
call cpptri (uplo, n, ap, info)
call zpptri (uplo, n, ap, info)
```

### Discussion

This routine computes the inverse ( $A^{-1}$ ) of a symmetric positive definite or, for complex flavors, Hermitian positive-definite matrix  $A$  in *packed* form. Before calling this routine, call [?pptrf](#) to factorize  $A$ .

### Input Parameters

<i>uplo</i>	CHARACTER*1. Must be ' <b>U</b> ' or ' <b>L</b> '.
	Indicates how the input matrix $A$ has been factored:
	If <i>uplo</i> = ' <b>U</b> ', the array <i>ap</i> stores the packed factor $U$ of the Cholesky factorization $A = U^H U$ .
	If <i>uplo</i> = ' <b>L</b> ', the array <i>ap</i> stores the packed factor $L$ of the Cholesky factorization $A = L L^H$ .
<i>n</i>	INTEGER. The order of the matrix $A$ ( $n \geq 0$ ).
<i>ap</i>	REAL for <b>spptri</b> DOUBLE PRECISION for <b>dpptri</b> COMPLEX for <b>cpptri</b> DOUBLE COMPLEX for <b>zpptri</b> . Array, DIMENSION at least $\max(1, n(n+1)/2)$ . Contains the factorization of the packed matrix $A$ , as returned by <a href="#">?pptrf</a> . The dimension <i>ap</i> must be at least $\max(1, n(n+1)/2)$ .

## Output Parameters

<i>ap</i>	Overwritten by the packed $n$ by $n$ matrix $A^{-1}$ .
<i>info</i>	<i>INTEGER</i> . If $\text{info} = 0$ , the execution is successful. If $\text{info} = -i$ , the $i$ th parameter had an illegal value. If $\text{info} = i$ , the $i$ th diagonal element of the Cholesky factor (and hence the factor itself) is zero, and the inversion could not be completed.

## Application Notes

The computed inverse  $X$  satisfies the following error bounds:

$$\|XA - I\|_2 \leq c(n)\varepsilon\kappa_2(A), \quad \|AX - I\|_2 \leq c(n)\varepsilon\kappa_2(A)$$

where  $c(n)$  is a modest linear function of  $n$ , and  $\varepsilon$  is the machine precision;  $I$  denotes the identity matrix.

The 2-norm  $\|A\|_2$  of a matrix  $A$  is defined by  $\|A\|_2 = \max_{x:x=1}(Ax \cdot Ax)^{1/2}$ , and the condition number  $\kappa_2(A)$  is defined by  $\kappa_2(A) = \|A\|_2 \|A^{-1}\|_2$ .

The total number of floating-point operations is approximately  $(2/3)n^3$  for real flavors and  $(8/3)n^3$  for complex flavors.

## ?sytri

Computes the inverse of a symmetric matrix.

```
call ssytri (uplo, n, a, lda, ipiv, work, info)
call dsytri (uplo, n, a, lda, ipiv, work, info)
call csytri (uplo, n, a, lda, ipiv, work, info)
call zsytri (uplo, n, a, lda, ipiv, work, info)
```

### Discussion

This routine computes the inverse ( $A^{-1}$ ) of a symmetric matrix  $A$ .

Before calling this routine, call [?sytrf](#) to factorize  $A$ .

### Input Parameters

<i>uplo</i>	CHARACTER*1. Must be ' <b>U</b> ' or ' <b>L</b> '.
	Indicates how the input matrix $A$ has been factored:
	If <i>uplo</i> = ' <b>U</b> ', the array <i>a</i> stores the Bunch-Kaufman factorization $A = PUDU^TP^T$ .
	If <i>uplo</i> = ' <b>L</b> ', the array <i>a</i> stores the Bunch-Kaufman factorization $A = PLDL^TP^T$ .
<i>n</i>	INTEGER. The order of the matrix $A$ ( <i>n</i> $\geq$ 0).
<i>a</i> , <i>work</i>	REAL for <b>ssytri</b> DOUBLE PRECISION for <b>dsytri</b> COMPLEX for <b>csytri</b> DOUBLE COMPLEX for <b>zsytri</b> . Arrays: <i>a</i> ( <i>lda</i> , *) contains the factorization of the matrix $A$ , as returned by <a href="#">?sytrf</a> . The second dimension of <i>a</i> must be at least max(1, <i>n</i> ). <i>work</i> (*) is a workspace array. The dimension of <i>work</i> must be at least max(1,2* <i>n</i> ). <i>lda</i> INTEGER. The first dimension of <i>a</i> ; <i>lda</i> $\geq$ max(1, <i>n</i> ).

*ipiv***INTEGER.**Array, **DIMENSION** at least  $\max(1, n)$ .The *ipiv* array, as returned by [?sytrf](#).

## Output Parameters

*a*Overwritten by the *n* by *n* matrix  $A^{-1}$ .*info***INTEGER.**If *info* = 0, the execution is successful.If *info* = -*i*, the *i*th parameter had an illegal value.If *info* = *i*, the *i*th diagonal element of *D* is zero, *D* is singular, and the inversion could not be completed.

## Application Notes

The computed inverse *X* satisfies the following error bounds:

$$|DU^T P^T X P U - I| \leq c(n) \varepsilon (|D| |U^T| |P^T| |X| |P| |U| + |D| |D^{-1}|)$$

for *uplo* = 'U', and

$$|DL^T P^T X P L - I| \leq c(n) \varepsilon (|D| |L^T| |P^T| |X| |P| |L| + |D| |D^{-1}|)$$

for *uplo* = 'L'. Here  $c(n)$  is a modest linear function of *n*, and  $\varepsilon$  is the machine precision; *I* denotes the identity matrix.The total number of floating-point operations is approximately  $(2/3)n^3$  for real flavors and  $(8/3)n^3$  for complex flavors.

## ?hetri

*Computes the inverse of a complex Hermitian matrix.*

```
call chetri (uplo, n, a, lda, ipiv, work, info)
call zhetri (uplo, n, a, lda, ipiv, work, info)
```

### Discussion

This routine computes the inverse ( $A^{-1}$ ) of a complex Hermitian matrix  $A$ . Before calling this routine, call [?hetrf](#) to factorize  $A$ .

### Input Parameters

<i>uplo</i>	CHARACTER*1. Must be ' <b>U</b> ' or ' <b>L</b> '.
	Indicates how the input matrix $A$ has been factored:
	If <i>uplo</i> = ' <b>U</b> ', the array <i>a</i> stores the Bunch-Kaufman factorization $A = PUDU^HPT$ .
	If <i>uplo</i> = ' <b>L</b> ', the array <i>a</i> stores the Bunch-Kaufman factorization $A = PLDL^HPT$ .
<i>n</i>	INTEGER. The order of the matrix $A$ ( <i>n</i> $\geq$ 0).
<i>a, work</i>	COMPLEX for <i>chetri</i> DOUBLE COMPLEX for <i>zhetri</i> . Arrays:  <i>a</i> ( <i>lda</i> ,*) contains the factorization of the matrix $A$ , as returned by <a href="#">?hetrf</a> . The second dimension of <i>a</i> must be at least max(1, <i>n</i> ). <i>work</i> (*) is a workspace array. The dimension of <i>work</i> must be at least max(1, <i>n</i> ).  <i>lda</i> INTEGER. The first dimension of <i>a</i> ; <i>lda</i> $\geq$ max(1, <i>n</i> ). <i>ipiv</i> INTEGER. Array, DIMENSION at least max(1, <i>n</i> ). The <i>ipiv</i> array, as returned by <a href="#">?hetrf</a> .

## Output Parameters

*a* Overwritten by the *n* by *n* matrix  $A^{-1}$ .

*info* INTEGER.

If *info* = 0, the execution is successful.

If *info* = *-i*, the *i*th parameter had an illegal value.

If *info* = *i*, the *i*th diagonal element of *D* is zero, *D* is singular, and the inversion could not be completed.

## Application Notes

The computed inverse *X* satisfies the following error bounds:

$$|DU^H P^T XPU - I| \leq c(n)\epsilon(|D||U^H|P^T|X|P|U| + |D||D^{-1}|)$$

for *uplo* = 'U', and

$$|DL^H P^T XPL - I| \leq c(n)\epsilon(|D||L^H|P^T|X|P|L| + |D||D^{-1}|)$$

for *uplo* = 'L'. Here  $c(n)$  is a modest linear function of *n*, and  $\epsilon$  is the machine precision; *I* denotes the identity matrix.

The total number of floating-point operations is approximately  $(2/3)n^3$  for real flavors and  $(8/3)n^3$  for complex flavors.

The real counterpart of this routine is [?sytri](#).

## ?sptri

*Computes the inverse of a symmetric matrix using packed storage.*

```
call ssptri (uplo, n, ap, ipiv, work, info)
call dsptri (uplo, n, ap, ipiv, work, info)
call csptri (uplo, n, ap, ipiv, work, info)
call zsptri (uplo, n, ap, ipiv, work, info)
```

### Discussion

This routine computes the inverse ( $A^{-1}$ ) of a packed symmetric matrix  $A$ . Before calling this routine, call [?sptrf](#) to factorize  $A$ .

### Input Parameters

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. Indicates how the input matrix $A$ has been factored: If <i>uplo</i> = 'U', the array <i>ap</i> stores the Bunch-Kaufman factorization $A = PUDU^TP^T$ . If <i>uplo</i> = 'L', the array <i>ap</i> stores the Bunch-Kaufman factorization $A = PLDL^TP^T$ .
<i>n</i>	INTEGER. The order of the matrix $A$ ( $n \geq 0$ ).
<i>ap</i> , <i>work</i>	REAL for <i>ssptri</i> DOUBLE PRECISION for <i>dsptri</i> COMPLEX for <i>csptri</i> DOUBLE COMPLEX for <i>zsptri</i> . Arrays: <i>ap</i> (*) contains the factorization of the matrix $A$ , as returned by <a href="#">?sptrf</a> . The dimension of <i>ap</i> must be at least $\max(1, n(n+1)/2)$ . <i>work</i> (*) is a workspace array. The dimension of <i>work</i> must be at least $\max(1, n)$ .

*ipiv***INTEGER.**Array, **DIMENSION** at least  $\max(1, n)$ .The *ipiv* array, as returned by [?sptrf](#).

## Output Parameters

*ap*Overwritten by the  $n$  by  $n$  matrix  $A^{-1}$  in packed form.*info***INTEGER.**If *info* = 0, the execution is successful.If *info* =  $-i$ , the *i*th parameter had an illegal value.If *info* = *i*, the *i*th diagonal element of  $D$  is zero,  $D$  is singular, and the inversion could not be completed.

## Application Notes

The computed inverse  $X$  satisfies the following error bounds:

$$|DU^T P^T X P U - I| \leq c(n) \varepsilon (|D| |U^T| |P^T| |X| |P| |U| + |D| |D^{-1}|)$$

for *uplo* = 'U', and

$$|DL^T P^T X P L - I| \leq c(n) \varepsilon (|D| |L^T| |P^T| |X| |P| |L| + |D| |D^{-1}|)$$

for *uplo* = 'L'. Here  $c(n)$  is a modest linear function of  $n$ , and  $\varepsilon$  is the machine precision;  $I$  denotes the identity matrix.The total number of floating-point operations is approximately  $(2/3)n^3$  for real flavors and  $(8/3)n^3$  for complex flavors.

## ?hptri

*Computes the inverse of a complex Hermitian matrix using packed storage.*

```
call chptri (uplo, n, ap, ipiv, work, info)
call zhptri (uplo, n, ap, ipiv, work, info)
```

### Discussion

This routine computes the inverse ( $A^{-1}$ ) of a complex Hermitian matrix  $A$  using packed storage.

Before calling this routine, call [?hptrf](#) to factorize  $A$ .

### Input Parameters

*uplo*            CHARACTER\*1. Must be '**U**' or '**L**'.

Indicates how the input matrix  $A$  has been factored:

If *uplo* = '**U**', the array *ap* stores the packed Bunch-Kaufman factorization  $A = PUDU^HPT$ .

If *uplo* = '**L**', the array *ap* stores the packed Bunch-Kaufman factorization  $A = PLDL^HPT$ .

*n*            INTEGER. The order of the matrix  $A$  (*n*  $\geq 0$ ).

*ap*            COMPLEX for *chptri*

DOUBLE COMPLEX for *zhptri*.

Arrays:

*ap*(\*) contains the factorization of the matrix  $A$ , as returned by [?hptrf](#).

The dimension of *ap* must be at least max(1,*n*(*n*+1)/2).

*work*(\*) is a workspace array.

The dimension of *work* must be at least max(1,*n*).

*ipiv*            INTEGER.

Array, DIMENSION at least max(1,*n*).

The *ipiv* array, as returned by [?hptrf](#).

## Output Parameters

<i>ap</i>	Overwritten by the <i>n</i> by <i>n</i> matrix $A^{-1}$ .
<i>info</i>	<b>INTEGER.</b> If <i>info</i> = 0, the execution is successful. If <i>info</i> = <i>-i</i> , the <i>i</i> th parameter had an illegal value. If <i>info</i> = <i>i</i> , the <i>i</i> th diagonal element of <i>D</i> is zero, <i>D</i> is singular, and the inversion could not be completed.

## Application Notes

The computed inverse *X* satisfies the following error bounds:

$$|DU^H P^T XPU - I| \leq c(n)\epsilon(|D||U^H|P^T|X|P|U| + |D||D^{-1}|)$$

for *uplo* = 'U', and

$$|DL^H P^T XPL - I| \leq c(n)\epsilon(|D||L^H|P^T|X|P|L| + |D||D^{-1}|)$$

for *uplo* = 'L'. Here  $c(n)$  is a modest linear function of *n*, and  $\epsilon$  is the machine precision; *I* denotes the identity matrix.

The total number of floating-point operations is approximately  $(2/3)n^3$  for real flavors and  $(8/3)n^3$  for complex flavors.

The real counterpart of this routine is [?sptri](#).

## ?trtri

*Computes the inverse of a triangular matrix.*

```
call strtri (uplo, diag, n, a, lda, info)
call dtrtri (uplo, diag, n, a, lda, info)
call ctrtri (uplo, diag, n, a, lda, info)
call ztrtri (uplo, diag, n, a, lda, info)
```

### Discussion

This routine computes the inverse ( $A^{-1}$ ) of a triangular matrix  $A$ .

### Input Parameters

<i>uplo</i>	CHARACTER*1. Must be ' <b>U</b> ' or ' <b>L</b> '. Indicates whether $A$ is upper or lower triangular: If <i>uplo</i> = ' <b>U</b> ', then $A$ is upper triangular. If <i>uplo</i> = ' <b>L</b> ', then $A$ is lower triangular.
<i>diag</i>	CHARACTER*1. Must be ' <b>N</b> ' or ' <b>U</b> '. If <i>diag</i> = ' <b>N</b> ', then $A$ is not a unit triangular matrix. If <i>diag</i> = ' <b>U</b> ', $A$ is unit triangular: diagonal elements of $A$ are assumed to be 1 and not referenced in the array <i>a</i> .
<i>n</i>	INTEGER. The order of the matrix $A$ ( <i>n</i> $\geq$ 0).
<i>a</i>	REAL for <b>strtri</b> DOUBLE PRECISION for <b>dtrtri</b> COMPLEX for <b>ctrtri</b> DOUBLE COMPLEX for <b>ztrtri</b> .  Array: DIMENSION ( <i>lda</i> , *). Contains the matrix $A$ . The second dimension of <i>a</i> must be at least $\max(1, n)$ .
<i>lda</i>	INTEGER. The first dimension of <i>a</i> ; <i>lda</i> $\geq \max(1, n)$ .

## Output Parameters

<i>a</i>	Overwritten by the <i>n</i> by <i>n</i> matrix $A^{-1}$ .
<i>info</i>	<i>INTEGER</i> . If <i>info</i> = 0, the execution is successful. If <i>info</i> = <i>-i</i> , the <i>i</i> th parameter had an illegal value. If <i>info</i> = <i>i</i> , the <i>i</i> th diagonal element of <i>A</i> is zero, <i>A</i> is singular, and the inversion could not be completed.

## Application Notes

The computed inverse *X* satisfies the following error bounds:

$$|XA - I| \leq c(n)\epsilon|X||A|$$

$$|X - A^{-1}| \leq c(n)\epsilon|A^{-1}||A||X|$$

where  $c(n)$  is a modest linear function of *n*;  $\epsilon$  is the machine precision; *I* denotes the identity matrix.

The total number of floating-point operations is approximately  $(1/3)n^3$  for real flavors and  $(4/3)n^3$  for complex flavors.

---

## ?tptri

*Computes the inverse of a triangular matrix using packed storage.*

---

```

call stptri (uplo, diag, n, ap, info)
call dtptri (uplo, diag, n, ap, info)
call ctptri (uplo, diag, n, ap, info)
call ztptri (uplo, diag, n, ap, info)

```

## Discussion

This routine computes the inverse ( $A^{-1}$ ) of a packed triangular matrix *A*.

## Input Parameters

<i>uplo</i>	CHARACTER*1. Must be ' <b>U</b> ' or ' <b>L</b> '. Indicates whether $A$ is upper or lower triangular: If <i>uplo</i> = ' <b>U</b> ', then $A$ is upper triangular. If <i>uplo</i> = ' <b>L</b> ', then $A$ is lower triangular.
<i>diag</i>	CHARACTER*1. Must be ' <b>N</b> ' or ' <b>U</b> '. If <i>diag</i> = ' <b>N</b> ', then $A$ is not a unit triangular matrix. If <i>diag</i> = ' <b>U</b> ', $A$ is unit triangular: diagonal elements of $A$ are assumed to be 1 and not referenced in the array <i>ap</i> .
<i>n</i>	INTEGER. The order of the matrix $A$ ( <i>n</i> $\geq$ 0).
<i>ap</i>	REAL for <b>stptri</b> DOUBLE PRECISION for <b>dtptri</b> COMPLEX for <b>ctptri</b> DOUBLE COMPLEX for <b>ztptri</b> .  Array: DIMENSION at least $\max(1, n(n+1)/2)$ . Contains the packed triangular matrix $A$ .

## Output Parameters

<i>ap</i>	Overwritten by the packed <i>n</i> by <i>n</i> matrix $A^{-1}$ .
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = <i>-i</i> , the <i>i</i> th parameter had an illegal value. If <i>info</i> = <i>i</i> , the <i>i</i> th diagonal element of $A$ is zero, $A$ is singular, and the inversion could not be completed.

## Application Notes

The computed inverse  $X$  satisfies the following error bounds:

$$|XA - I| \leq c(n)\epsilon|X||A|$$

$$|X - A^{-1}| \leq c(n)\epsilon|A^{-1}||A||X|$$

where  $c(n)$  is a modest linear function of *n*;  $\epsilon$  is the machine precision;  
 $I$  denotes the identity matrix.

The total number of floating-point operations is approximately  $(1/3)n^3$  for real flavors and  $(4/3)n^3$  for complex flavors.

## Routines for Matrix Equilibration

Routines described in this section are used to compute scaling factors needed to equilibrate a matrix. Note that these routines do not actually scale the matrices.

### ?geequ

*Computes row and column scaling factors intended to equilibrate a matrix and reduce its condition number.*

---

```
call sggequ (m, n, a, lda, r, c, rowcnd, colcnd, amax, info)
call dgeequ (m, n, a, lda, r, c, rowcnd, colcnd, amax, info)
call cggequ (m, n, a, lda, r, c, rowcnd, colcnd, amax, info)
call zggequ (m, n, a, lda, r, c, rowcnd, colcnd, amax, info)
```

#### Discussion

This routine computes row and column scalings intended to equilibrate an  $m$ -by- $n$  matrix A and reduce its condition number. The output array  $r$  returns the row scale factors and the array  $c$  the column scale factors. These factors are chosen to try to make the largest element in each row and column of the matrix B with elements  $b_{ij} = r(i) * a_{ij} * c(j)$  have absolute value 1.

#### Input Parameters

$m$	INTEGER. The number of rows of the matrix A, $m \geq 0$ .
$n$	INTEGER. The number of columns of the matrix A, $n \geq 0$ .
$a$	REAL for <b>sggequ</b> DOUBLE PRECISION for <b>dgeequ</b> COMPLEX for <b>cggequ</b> DOUBLE COMPLEX for <b>zggequ</b> .

Array: `DIMENSION ( lda, * ).`

Contains the  $m$ -by- $n$  matrix  $A$  whose equilibration factors are to be computed.

The second dimension of  $a$  must be at least  $\max(1, n)$ .

`lda`

`INTEGER.` The leading dimension of  $a$ ;  $lda \geq \max(1, m)$ .

## Output Parameters

`r, c`

`REAL` for single precision flavors;

`DOUBLE PRECISION` for double precision flavors.

Arrays:  $r(m)$ ,  $c(n)$ .

If  $info = 0$ , or  $info > m$ , the array  $r$  contains the row scale factors of the matrix  $A$ .

If  $info = 0$ , the array  $c$  contains the column scale factors of the matrix  $A$ .

`rowcnd`

`REAL` for single precision flavors;

`DOUBLE PRECISION` for double precision flavors.

If  $info = 0$  or  $info > m$ , `rowcnd` contains the ratio of the smallest  $r(i)$  to the largest  $r(i)$ .

`colcnd`

`REAL` for single precision flavors;

`DOUBLE PRECISION` for double precision flavors.

If  $info = 0$ , `colcnd` contains the ratio of the smallest  $c(i)$  to the largest  $c(i)$ .

`amax`

`REAL` for single precision flavors;

`DOUBLE PRECISION` for double precision flavors.

Absolute value of the largest element of the matrix  $A$ .

`info`

`INTEGER.`

If  $info = 0$ , the execution is successful.

If  $info = -i$ , the  $i$ th parameter had an illegal value.

If  $info = i$  and

$i \leq m$ , the  $i$ th row of  $A$  is exactly zero;

$i > m$ , the  $(i-m)$ th column of  $A$  is exactly zero.

## Application Notes

All the components of  $r$  and  $c$  are restricted to be between SMLNUM = smallest safe number and BIGNUM = largest safe number. Use of these scaling factors is not guaranteed to reduce the condition number of  $A$  but works well in practice.

If  $\text{rowcnd} \geq 0.1$  and  $\text{amax}$  is neither too large nor too small, it is not worth scaling by  $r$ . If  $\text{colcnd} \geq 0.1$ , it is not worth scaling by  $c$ .

If  $\text{amax}$  is very close to overflow or very close to underflow, the matrix  $A$  should be scaled.

## ?gbequ

*Computes row and column scaling factors intended to equilibrate a band matrix and reduce its condition number.*

```
call sgbequ (m, n, kl, ku, ab, ldab, r, c, rowcnd, colcnd,amax,info)
call dgbequ (m, n, kl, ku, ab, ldab, r, c, rowcnd, colcnd,amax,info)
call cgbequ (m, n, kl, ku, ab, ldab, r, c, rowcnd, colcnd,amax,info)
call zgbequ (m, n, kl, ku, ab, ldab, r, c, rowcnd, colcnd,amax,info)
```

### Discussion

This routine computes row and column scalings intended to equilibrate an  $m$ -by- $n$  band matrix  $A$  and reduce its condition number. The output array  $r$  returns the row scale factors and the array  $c$  the column scale factors. These factors are chosen to try to make the largest element in each row and column of the matrix  $B$  with elements  $b_{ij} = r(i) * a_{ij} * c(j)$  have absolute value 1.

### Input Parameters

$m$	INTEGER. The number of rows of the matrix $A$ , $m \geq 0$ .
$n$	INTEGER. The number of columns of the matrix $A$ , $n \geq 0$ .
$kl$	INTEGER. The number of sub-diagonals within the band of $A$ ( $kl \geq 0$ ).
$ku$	INTEGER. The number of super-diagonals within the band of $A$ ( $ku \geq 0$ ).
$ab$	REAL for sgbequ DOUBLE PRECISION for dgbequ COMPLEX for cgbequ DOUBLE COMPLEX for zgbequ. Array, DIMENSION ( $ldab, *$ ). Contains the original band matrix $A$ stored in rows from 1 to $kl + ku + 1$ .

The second dimension of  $\text{ab}$  must be at least  $\max(1, n)$ ;  $l_{dab}$  **INTEGER**. The leading dimension of  $\text{ab}$ ,  $l_{dab} \geq k_l + k_u + 1$ .

### Output Parameters

$r, c$	<b>REAL</b> for single precision flavors; <b>DOUBLE PRECISION</b> for double precision flavors. Arrays: $r(m), c(n)$ . If $info = 0$ , or $info > m$ , the array $r$ contains the row scale factors of the matrix $A$ . If $info = 0$ , the array $c$ contains the column scale factors of the matrix $A$ .
$rowcnd$	<b>REAL</b> for single precision flavors; <b>DOUBLE PRECISION</b> for double precision flavors. If $info = 0$ or $info > m$ , $rowcnd$ contains the ratio of the smallest $r(i)$ to the largest $r(i)$ .
$colcnd$	<b>REAL</b> for single precision flavors; <b>DOUBLE PRECISION</b> for double precision flavors. If $info = 0$ , $colcnd$ contains the ratio of the smallest $c(i)$ to the largest $c(i)$ .
$amax$	<b>REAL</b> for single precision flavors; <b>DOUBLE PRECISION</b> for double precision flavors. Absolute value of the largest element of the matrix $A$ .
$info$	<b>INTEGER</b> . If $info = 0$ , the execution is successful. If $info = -i$ , the $i$ th parameter had an illegal value. If $info = i$ and $i \leq m$ , the $i$ th row of $A$ is exactly zero; $i > m$ , the $(i-m)$ th column of $A$ is exactly zero.

### Application Notes

All the components of  $r$  and  $c$  are restricted to be between SMLNUM = smallest safe number and BIGNUM = largest safe number. Use of these scaling factors is not guaranteed to reduce the condition number of  $A$  but works well in practice.

If  $\text{rowcnd} \geq 0.1$  and  $\text{amax}$  is neither too large nor too small, it is not worth scaling by  $r$ . If  $\text{colcnd} \geq 0.1$ , it is not worth scaling by  $c$ .

If  $\text{amax}$  is very close to overflow or very close to underflow, the matrix  $A$  should be scaled.

## ?poequ

*Computes row and column scaling factors intended to equilibrate a symmetric (Hermitian) positive definite matrix and reduce its condition number.*

```
call spoequ (n, a, lda, s, scond, amax, info)
call dpoequ (n, a, lda, s, scond, amax, info)
call cpoequ (n, a, lda, s, scond, amax, info)
call zpoequ (n, a, lda, s, scond, amax, info)
```

### Discussion

This routine computes row and column scalings intended to equilibrate a symmetric (Hermitian) positive definite matrix  $A$  and reduce its condition number (with respect to the two-norm). The output array  $s$  returns scale factors computed as

$$s(i) = \frac{1}{\sqrt{a_{i,i}}}$$

These factors are chosen so that the scaled matrix  $B$  with elements  $b_{ij} = s(i)*a_{ij}*s(j)$  has diagonal elements equal to 1.

This choice of  $s$  puts the condition number of  $B$  within a factor  $n$  of the smallest possible condition number over all possible diagonal scalings.

### Input Parameters

**$n$**                     **INTEGER.** The order of the matrix  $A$ ,  $n \geq 0$ .

<i>a</i>	<code>REAL</code> for <code>speq</code> <code>DOUBLE PRECISION</code> for <code>dpeq</code> <code>COMPLEX</code> for <code>cpeq</code> <code>DOUBLE COMPLEX</code> for <code>zpeq</code> .
	Array: <code>DIMENSION ( lda, * )</code> . Contains the <i>n</i> -by- <i>n</i> symmetric or Hermitian positive definite matrix <i>A</i> whose scaling factors are to be computed. Only diagonal elements of <i>A</i> are referenced. The second dimension of <i>a</i> must be at least <code>max(1, n)</code> .
<i>lda</i>	<code>INTEGER</code> . The leading dimension of <i>a</i> ; <i>lda</i> $\geq \max(1, m)$ .

## Output Parameters

<i>s</i>	<code>REAL</code> for single precision flavors; <code>DOUBLE PRECISION</code> for double precision flavors. Array, <code>DIMENSION ( n )</code> . If <i>info</i> = 0, the array <i>s</i> contains the scale factors for <i>A</i> .
<i>scond</i>	<code>REAL</code> for single precision flavors; <code>DOUBLE PRECISION</code> for double precision flavors. If <i>info</i> = 0, <i>scond</i> contains the ratio of the smallest <i>s</i> (i) to the largest <i>s</i> (i).
<i>amax</i>	<code>REAL</code> for single precision flavors; <code>DOUBLE PRECISION</code> for double precision flavors. Absolute value of the largest element of the matrix <i>A</i> .
<i>info</i>	<code>INTEGER</code> . If <i>info</i> = 0, the execution is successful. If <i>info</i> = <i>-i</i> , the <i>i</i> th parameter had an illegal value. If <i>info</i> = <i>i</i> , the <i>i</i> th diagonal element of <i>A</i> is nonpositive.

## Application Notes

If *scond*  $\geq 0.1$  and *amax* is neither too large nor too small, it is not worth scaling by *s*.  
If *amax* is very close to overflow or very close to underflow, the matrix *A* should be scaled.

## ?ppequ

*Computes row and column scaling factors intended to equilibrate a symmetric (Hermitian) positive definite matrix in packed storage and reduce its condition number.*

```
call sppequ (uplo, n, ap, s, scond, amax, info)
call dppequ (uplo, n, ap, s, scond, amax, info)
call cppequ (uplo, n, ap, s, scond, amax, info)
call zppequ (uplo, n, ap, s, scond, amax, info)
```

### Discussion

This routine computes row and column scalings intended to equilibrate a symmetric (Hermitian) positive definite matrix  $A$  in packed storage and reduce its condition number (with respect to the two-norm). The output array  $s$  returns scale factors computed as

$$s(i) = \frac{1}{\sqrt{a_{i,i}}}$$

These factors are chosen so that the scaled matrix  $B$  with elements  $b_{ij} = s(i)*a_{ij}*s(j)$  has diagonal elements equal to 1.

This choice of  $s$  puts the condition number of  $B$  within a factor  $n$  of the smallest possible condition number over all possible diagonal scalings.

### Input Parameters

**uplo**                    CHARACTER\*1. Must be '**U**' or '**L**'.  
 Indicates whether the upper or lower triangular part of  $A$  is packed in the array **ap**:  
 If **uplo** = '**U**', the array **ap** stores the upper triangular part of the matrix  $A$ .  
 If **uplo** = '**L**', the array **ap** stores the lower triangular part of the matrix  $A$ .

*n*                    **INTEGER.** The order of matrix *A* (*n* ≥ 0).  
*ap*                  **REAL** for **sppequ**  
                        **DOUBLE PRECISION** for **dppequ**  
                        **COMPLEX** for **cppequ**  
                        **DOUBLE COMPLEX** for **zppequ**.  
Array, **DIMENSION** at least max(1,*n*(*n*+1)/2).  
The array *ap* contains either the upper or the lower  
triangular part of the matrix *A* (as specified by *uplo*) in  
*packed storage* (see [Matrix Storage Schemes](#)).

## Output Parameters

*s*                    **REAL** for single precision flavors;  
                        **DOUBLE PRECISION** for double precision flavors.  
Array, **DIMENSION** (*n*).  
If *info* = 0, the array *s* contains the scale factors for *A*.  
*scond*                **REAL** for single precision flavors;  
                        **DOUBLE PRECISION** for double precision flavors.  
If *info* = 0, *scond* contains the ratio of the smallest  
*s*(*i*) to the largest *s*(*i*).  
*amax*                **REAL** for single precision flavors;  
                        **DOUBLE PRECISION** for double precision flavors.  
Absolute value of the largest element of the matrix *A*.  
*info*                 **INTEGER.**  
If *info* = 0, the execution is successful.  
If *info* = -*i*, the *i*th parameter had an illegal value.  
If *info* = *i*, the *i*th diagonal element of *A* is  
nonpositive.

## Application Notes

If *scond* ≥ 0.1 and *amax* is neither too large nor too small, it is not worth scaling by *s*.

If *amax* is very close to overflow or very close to underflow, the matrix *A* should be scaled.

## ?pbequ

*Computes row and column scaling factors intended to equilibrate a symmetric (Hermitian) positive definite band matrix and reduce its condition number.*

```
call spbequ (uplo, n, kd, ab, ldab, s, scond, amax, info)
call dpbequ (uplo, n, kd, ab, ldab, s, scond, amax, info)
call cpbequ (uplo, n, kd, ab, ldab, s, scond, amax, info)
call zpbequ (uplo, n, kd, ab, ldab, s, scond, amax, info)
```

### Discussion

This routine computes row and column scalings intended to equilibrate a symmetric (Hermitian) positive definite matrix  $A$  in packed storage and reduce its condition number (with respect to the two-norm). The output array  $s$  returns scale factors computed as

$$s(i) = \frac{1}{\sqrt{a_{i,i}}}$$

These factors are chosen so that the scaled matrix  $B$  with elements  $b_{ij} = s(i)*a_{ij}*s(j)$  has diagonal elements equal to 1.

This choice of  $s$  puts the condition number of  $B$  within a factor  $n$  of the smallest possible condition number over all possible diagonal scalings.

### Input Parameters

<i>uplo</i>	CHARACTER*1. Must be ' <b>U</b> ' or ' <b>L</b> '.
	Indicates whether the upper or lower triangular part of $A$ is packed in the array <i>ab</i> :
	If <i>uplo</i> = ' <b>U</b> ', the array <i>ab</i> stores the upper triangular part of the matrix $A$ .
	If <i>uplo</i> = ' <b>L</b> ', the array <i>ab</i> stores the lower triangular part of the matrix $A$ .
<i>n</i>	INTEGER. The order of matrix $A$ ( $n \geq 0$ ).
<i>kd</i>	INTEGER. The number of super-diagonals or sub-diagonals in the matrix $A$ ( $kd \geq 0$ ).

<i>ab</i>	<b>REAL</b> for <i>spbequ</i> <b>DOUBLE PRECISION</b> for <i>dpbequ</i> <b>COMPLEX</b> for <i>cpbequ</i> <b>DOUBLE COMPLEX</b> for <i>zpbequ</i> . Array, <b>DIMENSION</b> ( <i>ldab</i> , *). The array <i>ap</i> contains either the upper or the lower triangular part of the matrix <i>A</i> (as specified by <i>uplo</i> ) in <i>band storage</i> (see <a href="#">Matrix Storage Schemes</a> ). The second dimension of <i>ab</i> must be at least $\max(1, n)$ .
<i>ldab</i>	<b>INTEGER</b> . The leading dimension of the array <i>ab</i> . ( <i>ldab</i> $\geq kd + 1$ ).

## Output Parameters

<i>s</i>	<b>REAL</b> for single precision flavors; <b>DOUBLE PRECISION</b> for double precision flavors. Array, <b>DIMENSION</b> ( <i>n</i> ). If <i>info</i> = 0, the array <i>s</i> contains the scale factors for <i>A</i> .
<i>scond</i>	<b>REAL</b> for single precision flavors; <b>DOUBLE PRECISION</b> for double precision flavors. If <i>info</i> = 0, <i>scond</i> contains the ratio of the smallest <i>s</i> (i) to the largest <i>s</i> (i).
<i>amax</i>	<b>REAL</b> for single precision flavors; <b>DOUBLE PRECISION</b> for double precision flavors. Absolute value of the largest element of the matrix <i>A</i> .
<i>info</i>	<b>INTEGER</b> . If <i>info</i> = 0, the execution is successful. If <i>info</i> = <i>-i</i> , the <i>i</i> th parameter had an illegal value. If <i>info</i> = <i>i</i> , the <i>i</i> th diagonal element of <i>A</i> is nonpositive.

## Application Notes

- If *scond*  $\geq 0.1$  and *amax* is neither too large nor too small, it is not worth scaling by *s*.  
If *amax* is very close to overflow or very close to underflow, the matrix *A* should be scaled.

## Driver Routines

[Table 4-3](#) lists the LAPACK driver routines for solving systems of linear equations with real or complex matrices.

**Table 4-3 Driver Routines for Solving Systems of Linear Equations**

Matrix type, storage scheme	Simple Driver	Expert Driver
general	<a href="#">?gesv</a>	<a href="#">?gesvx</a>
general band	<a href="#">?gbsv</a>	<a href="#">?gbsvx</a>
general tridiagonal	<a href="#">?gtsv</a>	<a href="#">?gtsvx</a>
symmetric/Hermitian positive-definite	<a href="#">?posv</a>	<a href="#">?posvx</a>
symmetric/Hermitian positive-definite, packed storage	<a href="#">?ppsv</a>	<a href="#">?ppsvx</a>
symmetric/Hermitian positive-definite, band	<a href="#">?pbsv</a>	<a href="#">?pbsvx</a>
symmetric/Hermitian positive-definite, tridiagonal	<a href="#">?ptsv</a>	<a href="#">?ptsvx</a>
symmetric/Hermitian indefinite	<a href="#">?sysv / ?hesv</a>	<a href="#">?sysvx / ?hesvx</a>
symmetric/Hermitian indefinite, packed storage	<a href="#">?spsv / ?hpsv</a>	<a href="#">?spsvx / ?hpsvx</a>
complex symmetric	<a href="#">?sysv</a>	<a href="#">?sysvx</a>
complex symmetric, packed storage	<a href="#">?spsv</a>	<a href="#">?spsvx</a>

In this table **?** stands for **s** (single precision real), **d** (double precision real), **c** (single precision complex), or **z** (double precision complex).

---

## ?gesv

*Computes the solution to the system of linear equations with a square matrix A and multiple right-hand sides.*

---

```
call sgesv (n, nrhs, a, lda, ipiv, b, ldb, info)
call dgesv (n, nrhs, a, lda, ipiv, b, ldb, info)
call cgesv (n, nrhs, a, lda, ipiv, b, ldb, info)
call zgesv (n, nrhs, a, lda, ipiv, b, ldb, info)
```

### Discussion

This routine solves for  $X$  the system of linear equations  $AX = B$ , where  $A$  is an  $n$ -by- $n$  matrix, the columns of matrix  $B$  are individual right-hand sides, and the columns of  $X$  are the corresponding solutions.

The  $LU$  decomposition with partial pivoting and row interchanges is used to factor  $A$  as  $A = P L U$ , where  $P$  is a permutation matrix,  $L$  is unit lower triangular, and  $U$  is upper triangular. The factored form of  $A$  is then used to solve the system of equations  $AX = B$ .

### Input Parameters

<i>n</i>	<b>INTEGER.</b> The order of $A$ ; the number of rows in $B$ ( $n \geq 0$ ).
<i>nrhs</i>	<b>INTEGER.</b> The number of right-hand sides; the number of columns in $B$ ( $nrhs \geq 0$ ).
<i>a, b</i>	<b>REAL</b> for <b>sgesv</b> <b>DOUBLE PRECISION</b> for <b>dgesv</b> <b>COMPLEX</b> for <b>cgesv</b> <b>DOUBLE COMPLEX</b> for <b>zgesv</b> . Arrays: <i>a</i> ( <i>lda,*</i> ), <i>b</i> ( <i>ldb,*</i> ). The array <i>a</i> contains the matrix $A$ . The array <i>b</i> contains the matrix $B$ whose columns are the right-hand sides for the systems of equations. The second dimension of <i>a</i> must be at least $\max(1, n)$ , the second dimension of <i>b</i> at least $\max(1, nrhs)$ .

---

*lda*                    INTEGER. The first dimension of *a*; *lda*  $\geq \max(1, n).  
*ldb*                    INTEGER. The first dimension of *b*; *ldb*  $\geq \max(1, n).$$

### Output Parameters

*a*                    Overwritten by the factors *L* and *U* from the factorization of  $A = P L U$ ; the unit diagonal elements of *L* are not stored .

*b*                    Overwritten by the solution matrix *X*.

*ipiv*                INTEGER.  
 Array, DIMENSION at least  $\max(1, n)$ .  
 The pivot indices that define the permutation matrix *P*;  
 row *i* of the matrix was interchanged with row *ipiv(i)*.

*info*                INTEGER. If *info*=0, the execution is successful.  
 If *info* = *-i*, the *i*th parameter had an illegal value.  
 If *info* = *i*,  $U(i, i)$  is exactly zero. The factorization has been completed, but the factor *U* is exactly singular, so the solution could not be computed.

---

## ?gesvx

Computes the solution to the system of linear equations with a square matrix *A* and multiple right-hand sides, and provides error bounds on the solution.

---

```
call sgesvx (fact, trans, n, nrhs, a, lda, af, ldaf, ipiv, equed, r,
             c, b, ldb, x, ldx, rcond, ferr, berr, work, iwork, info)
call dgesvx (fact, trans, n, nrhs, a, lda, af, ldaf, ipiv, equed, r,
             c, b, ldb, x, ldx, rcond, ferr, berr, work, iwork, info)
call cgesvx (fact, trans, n, nrhs, a, lda, af, ldaf, ipiv, equed, r,
             c, b, ldb, x, ldx, rcond, ferr, berr, work, rwork, info)
call zgesvx (fact, trans, n, nrhs, a, lda, af, ldaf, ipiv, equed, r,
             c, b, ldb, x, ldx, rcond, ferr, berr, work, rwork, info)
```

## Discussion

This routine uses the *LU* factorization to compute the solution to a real or complex system of linear equations  $AX = B$ , where  $A$  is an  $n$ -by- $n$  matrix, the columns of matrix  $B$  are individual right-hand sides, and the columns of  $X$  are the corresponding solutions.

Error bounds on the solution and a condition estimate are also provided.

The routine `?gesvx` performs the following steps:

1. If  $\text{fact} = \text{'E'}$ , real scaling factors  $r$  and  $c$  are computed to equilibrate the system:

$$\text{trans} = \text{'N'}: \quad \text{diag}(r) * A * \text{diag}(c) * \text{diag}(c)^{-1} * X = \text{diag}(r) * B$$

$$\text{trans} = \text{'T'}: \quad (\text{diag}(r) * A * \text{diag}(c))^T * \text{diag}(r)^{-1} * X = \text{diag}(c) * B$$

$$\text{trans} = \text{'C'}: \quad (\text{diag}(r) * A * \text{diag}(c))^H * \text{diag}(r)^{-1} * X = \text{diag}(c) * B$$

Whether or not the system will be equilibrated depends on the scaling of the matrix  $A$ , but if equilibration is used,  $A$  is overwritten by  $\text{diag}(r) * A * \text{diag}(c)$  and  $B$  by  $\text{diag}(r) * B$  (if  $\text{trans} = \text{'N'}$ ) or  $\text{diag}(c) * B$  (if  $\text{trans} = \text{'T'}$  or  $\text{'C'}$ ).

2. If  $\text{fact} = \text{'N'}$  or  $\text{'E'}$ , the *LU* decomposition is used to factor the matrix  $A$  (after equilibration if  $\text{fact} = \text{'E'}$ ) as  $A = P L U$ , where  $P$  is a permutation matrix,  $L$  is a unit lower triangular matrix, and  $U$  is upper triangular.
3. If some  $U_{i,i} = 0$ , so that  $U$  is exactly singular, then the routine returns with  $\text{info} = i$ . Otherwise, the factored form of  $A$  is used to estimate the condition number of the matrix  $A$ . If the reciprocal of the condition number is less than machine precision,  $\text{info} = n + 1$  is returned as a warning, but the routine still goes on to solve for  $X$  and compute error bounds as described below.
4. The system of equations is solved for  $X$  using the factored form of  $A$ .
5. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.
6. If equilibration was used, the matrix  $X$  is premultiplied by  $\text{diag}(c)$  (if  $\text{trans} = \text{'N'}$ ) or  $\text{diag}(r)$  (if  $\text{trans} = \text{'T'}$  or  $\text{'C'}$ ) so that it solves the original system before equilibration.

## Input Parameters

*fact* **CHARACTER\*1**. Must be '*F*', '*N*', or '*E*'.

Specifies whether or not the factored form of the matrix *A* is supplied on entry, and if not, whether the matrix *A* should be equilibrated before it is factored.

If *fact* = '*F*': on entry, *af* and *ipiv* contain the factored form of *A*. If *equed* is not '*N*', the matrix *A* has been equilibrated with scaling factors given by *r* and *c*. *a*, *af*, and *ipiv* are not modified.

If *fact* = '*N*', the matrix *A* will be copied to *af* and factored.

If *fact* = '*E*', the matrix *A* will be equilibrated if necessary, then copied to *af* and factored.

*trans* **CHARACTER\*1**. Must be '*N*', '*T*', or '*C*'.

Specifies the form of the system of equations:

If *trans* = '*N*', the system has the form  $A X = B$  (No transpose);

If *trans* = '*T*', the system has the form  $A^T X = B$  (Transpose);

If *trans* = '*C*', the system has the form  $A^H X = B$  (Conjugate transpose);

*n* **INTEGER**. The number of linear equations; the order of the matrix *A* (*n*  $\geq 0$ ).

*nrhs* **INTEGER**. The number of right hand sides; the number of columns of the matrices *B* and *X* (*nrhs*  $\geq 0$ ).

*a, af, b, work* **REAL** for *s gesvx*

**DOUBLE PRECISION** for *d gesvx*

**COMPLEX** for *c gesvx*

**DOUBLE COMPLEX** for *z gesvx*.

Arrays: *a*(*lda*,\*), *af*(*ldaf*,\*), *b*(*ldb*,\*), *work*(\*).

The array *a* contains the matrix *A*. If *fact* = '*F*' and *equed* is not '*N*', then *A* must have been equilibrated by the scaling factors in *r* and/or *c*. The second dimension

of  $a$  must be at least  $\max(1, n)$ .

The array  $af$  is an input argument if  $fact = 'F'$ . It contains the factored form of the matrix  $A$ , i.e., the factors  $L$  and  $U$  from the factorization  $A = P L U$  as computed by [?getrf](#). If  $equed$  is not ' $N$ ', then  $af$  is the factored form of the equilibrated matrix  $A$ . The second dimension of  $af$  must be at least  $\max(1, n)$ .

The array  $b$  contains the matrix  $B$  whose columns are the right-hand sides for the systems of equations. The second dimension of  $b$  must be at least  $\max(1, nrhs)$ .

$work(*)$  is a workspace array.

The dimension of  $work$  must be at least  $\max(1, 4 * n)$  for real flavors, and at least  $\max(1, 2 * n)$  for complex flavors.

$lda$	<code>INTEGER</code> . The first dimension of $a$ ; $lda \geq \max(1, n)$ .
$ldaf$	<code>INTEGER</code> . The first dimension of $af$ ; $ldaf \geq \max(1, n)$ .
$ldb$	<code>INTEGER</code> . The first dimension of $b$ ; $ldb \geq \max(1, n)$ .
$ipiv$	<code>INTEGER</code> .

$ipiv$	Array, <code>DIMENSION</code> at least $\max(1, n)$ . The array $ipiv$ is an input argument if $fact = 'F'$ . It contains the pivot indices from the factorization $A = P L U$ as computed by <a href="#">?getrf</a> ; row $i$ of the matrix was interchanged with row $ipiv(i)$ .
--------	---

$equed$	<code>CHARACTER*1</code> . Must be ' $N$ ', ' $R$ ', ' $C$ ', or ' $B$ '. $equed$ is an input argument if $fact = 'F'$ . It specifies the form of equilibration that was done: If $equed = 'N'$ , no equilibration was done (always true if $fact = 'N'$ ); If $equed = 'R'$ , row equilibration was done and $A$ has been premultiplied by $\text{diag}(r)$ ; If $equed = 'C'$ , column equilibration was done and $A$ has been postmultiplied by $\text{diag}(c)$ ; If $equed = 'B'$ , both row and column equilibration was done; $A$ has been replaced by $\text{diag}(r) * A * \text{diag}(c)$ .
---------	--

---

<i>r, c</i>	<b>REAL</b> for single precision flavors; <b>DOUBLE PRECISION</b> for double precision flavors. Arrays: <i>r(n)</i> , <i>c(n)</i> . The array <i>r</i> contains the row scale factors for <i>A</i> , and the array <i>c</i> contains the column scale factors for <i>A</i> . These arrays are input arguments if <i>fact</i> = 'F' only; otherwise they are output arguments. If <i>equed</i> = 'R' or 'B', <i>A</i> is multiplied on the left by $\text{diag}(r)$ ; if <i>equed</i> = 'N' or 'C', <i>r</i> is not accessed. If <i>fact</i> = 'F' and <i>equed</i> = 'R' or 'B', each element of <i>r</i> must be positive. If <i>equed</i> = 'C' or 'B', <i>A</i> is multiplied on the right by $\text{diag}(c)$ ; if <i>equed</i> = 'N' or 'R', <i>c</i> is not accessed. If <i>fact</i> = 'F' and <i>equed</i> = 'C' or 'B', each element of <i>c</i> must be positive.
<i>ldx</i>	<b>INTEGER</b> . The first dimension of the output array <i>x</i> ; $l dx \geq \max(1, n)$ .
<i>iwork</i>	<b>INTEGER</b> . Workspace array, <b>DIMENSION</b> at least $\max(1, n)$ ; used in real flavors only.
<i>rwork</i>	<b>REAL</b> for single precision flavors; <b>DOUBLE PRECISION</b> for double precision flavors. Workspace array, <b>DIMENSION</b> at least $\max(1, 2 * n)$ ; used in complex flavors only.

## Output Parameters

<i>x</i>	<b>REAL</b> for <i>s gesvx</i> <b>DOUBLE PRECISION</b> for <i>d gesvx</i> <b>COMPLEX</b> for <i>c gesvx</i> <b>DOUBLE COMPLEX</b> for <i>z gesvx</i> . Array, <b>DIMENSION</b> ( <i>l dx, *</i> ). If <i>info</i> = 0 or <i>info</i> = <i>n</i> +1, the array <i>x</i> contains the solution matrix <i>X</i> to the <i>original</i> system of equations. Note that <i>A</i> and <i>B</i> are modified on exit if <i>equed</i> ≠ 'N', and the solution to the <i>equilibrated</i> system is: $\text{diag}(c)^{-1} * X$ , if <i>trans</i> = 'N' and <i>equed</i> = 'C' or 'B';
----------	---

$\text{diag}(\text{r})^{-1} * X$ , if  $\text{trans} = \text{'T'}$  or  $\text{'C'}$  and  $\text{equed} = \text{'R'}$  or  $\text{'B'}$ .

The second dimension of  $\text{x}$  must be at least  $\max(1, \text{nrhs})$ .

$a$

Array  $a$  is not modified on exit if  $\text{fact} = \text{'F'}$  or  $\text{'N'}$ , or if  $\text{fact} = \text{'E'}$  and  $\text{equed} = \text{'N'}$ .

If  $\text{equed} \neq \text{'N'}$ ,  $A$  is scaled on exit as follows:

$\text{equed} = \text{'R'}: A = \text{diag}(\text{r}) * A$

$\text{equed} = \text{'C'}: A = A * \text{diag}(\text{c})$

$\text{equed} = \text{'B'}: A = \text{diag}(\text{r}) * A * \text{diag}(\text{c})$

$af$

If  $\text{fact} = \text{'N'}$  or  $\text{'E'}$ , then  $af$  is an output argument and on exit returns the factors  $L$  and  $U$  from the factorization  $A = P L U$  of the original matrix  $A$  (if  $\text{fact} = \text{'N'}$ ) or of the equilibrated matrix  $A$  (if  $\text{fact} = \text{'E'}$ ). See the description of  $a$  for the form of the equilibrated matrix.

$b$

Overwritten by  $\text{diag}(\text{r}) * B$  if  $\text{trans} = \text{'N'}$  and  $\text{equed} = \text{'R'}$  or  $\text{'B'}$ ;  
overwritten by  $\text{diag}(\text{c}) * B$  if  $\text{trans} = \text{'T'}$  and  $\text{equed} = \text{'C'}$  or  $\text{'B'}$ ;  
not changed if  $\text{equed} = \text{'N'}$ .

$r, c$

These arrays are output arguments if  $\text{fact} \neq \text{'F'}$ .

See the description of  $r, c$  in *Input Arguments* section.

$rcond$

**REAL** for single precision flavors.

**DOUBLE PRECISION** for double precision flavors.

An estimate of the reciprocal condition number of the matrix  $A$  after equilibration (if done). The routine sets  $rcond = 0$  if the estimate underflows; in this case the matrix is singular (to working precision). However, anytime  $rcond$  is small compared to 1.0, for the working precision, the matrix may be poorly conditioned or even singular.

$ferr, berr$

**REAL** for single precision flavors.

**DOUBLE PRECISION** for double precision flavors.

Arrays, **DIMENSION** at least  $\max(1, \text{nrhs})$ . Contain the component-wise forward and relative backward errors, respectively, for each solution vector.

---

<i>ipiv</i>	If <i>fact</i> = 'N' or 'E', then <i>ipiv</i> is an output argument and on exit contains the pivot indices from the factorization $A = P L U$ of the original matrix $A$ (if <i>fact</i> = 'N') or of the equilibrated matrix $A$ (if <i>fact</i> = 'E').
<i>equed</i>	If <i>fact</i> ≠ 'F', then <i>equed</i> is an output argument. It specifies the form of equilibration that was done (see the description of <i>equed</i> in <i>Input Arguments</i> section).
<i>work, rwork</i>	On exit, <i>work</i> (1) for real flavors, or <i>rwork</i> (1) for complex flavors, contains the reciprocal pivot growth factor $\text{norm}(A)/\text{norm}(U)$ . The "max absolute element" norm is used. If <i>work</i> (1) for real flavors, or <i>rwork</i> (1) for complex flavors is much less than 1, then the stability of the $LU$ factorization of the (equilibrated) matrix $A$ could be poor. This also means that the solution <i>x</i> , condition estimator <i>rcond</i> , and forward error bound <i>ferr</i> could be unreliable. If factorization fails with $0 < \text{info} \leq n$ , then <i>work</i> (1) for real flavors, or <i>rwork</i> (1) for complex flavors contains the reciprocal pivot growth factor for the leading <i>info</i> columns of $A$ .
<i>info</i>	<b>INTEGER.</b> If <i>info</i> =0, the execution is successful. If <i>info</i> = <i>-i</i> , the <i>i</i> th parameter had an illegal value. If <i>info</i> = <i>i</i> , and <i>i</i> ≤ <i>n</i> , then $U(i,i)$ is exactly zero. The factorization has been completed, but the factor $U$ is exactly singular, so the solution and error bounds could not be computed; <i>rcond</i> = 0 is returned. If <i>info</i> = <i>i</i> , and <i>i</i> = <i>n</i> +1, then $U$ is nonsingular, but <i>rcond</i> is less than machine precision, meaning that the matrix is singular to working precision. Nevertheless, the solution and error bounds are computed because there are a number of situations where the computed solution can be more accurate than the value of <i>rcond</i> would suggest.

## ?gbsv

*Computes the solution to the system of linear equations with a band matrix A and multiple right-hand sides.*

---

```
call sgbsv (n, k1, ku, nrhs, ab, ldab, ipiv, b, ldb, info)
call dgbsv (n, k1, ku, nrhs, ab, ldab, ipiv, b, ldb, info)
call cgbsv (n, k1, ku, nrhs, ab, ldab, ipiv, b, ldb, info)
call zgbsv (n, k1, ku, nrhs, ab, ldab, ipiv, b, ldb, info)
```

### Discussion

This routine solves for  $X$  the real or complex system of linear equations  $AX = B$ , where  $A$  is an  $n$ -by- $n$  band matrix with  $k1$  subdiagonals and  $ku$  superdiagonals, the columns of matrix  $B$  are individual right-hand sides, and the columns of  $X$  are the corresponding solutions.

The  $LU$  decomposition with partial pivoting and row interchanges is used to factor  $A$  as  $A = L U$ , where  $L$  is a product of permutation and unit lower triangular matrices with  $k1$  subdiagonals, and  $U$  is upper triangular with  $k1+ku$  superdiagonals. The factored form of  $A$  is then used to solve the system of equations  $AX = B$ .

### Input Parameters

$n$	<b>INTEGER.</b> The order of $A$ ; the number of rows in $B$ ( $n \geq 0$ ).
$k1$	<b>INTEGER.</b> The number of sub-diagonals within the band of $A$ ( $k1 \geq 0$ ).
$ku$	<b>INTEGER.</b> The number of super-diagonals within the band of $A$ ( $ku \geq 0$ ).
$nrhs$	<b>INTEGER.</b> The number of right-hand sides; the number of columns in $B$ ( $nrhs \geq 0$ ).
$ab, b$	<b>REAL</b> for <b>sgbsv</b> <b>DOUBLE PRECISION</b> for <b>dgbsv</b> <b>COMPLEX</b> for <b>cgbsv</b>

**DOUBLE COMPLEX** for `zgbsv`.

Arrays: `ab(ldb, *)`, `b(ldb, *)`.

The array `ab` contains the matrix  $A$  in band storage  
(see [Matrix Storage Schemes](#)).

The second dimension of `ab` must be at least  $\max(1, n)$ .

The array `b` contains the matrix  $B$  whose columns are  
the right-hand sides for the systems of equations.

The second dimension of `b` must be at least  
 $\max(1, nrhs)$ .

`ldab`      **INTEGER**. The first dimension of the array `ab`.

( $ldab \geq 2k_1 + k_u + 1$ )

`ldb`      **INTEGER**. The first dimension of `b`;  $ldb \geq \max(1, n)$ .

## Output Parameters

`ab`      Overwritten by  $L$  and  $U$ . The diagonal and  $k_1 + k_u$   
super-diagonals of  $U$  are stored in the first  $1 + k_1 + k_u$   
rows of `ab`. The multipliers used to form  $L$  are stored in  
the next  $k_1$  rows.

`b`      Overwritten by the solution matrix  $X$ .

`ipiv`      **INTEGER**.

Array, **DIMENSION** at least  $\max(1, n)$ .

The pivot indices: row  $i$  was interchanged with row  
`ipiv(i)`.

`info`      **INTEGER**. If `info=0`, the execution is successful.

If `info = -i`, the  $i$ th parameter had an illegal value.

If `info = i`,  $U(i,i)$  is exactly zero. The factorization  
has been completed, but the factor  $U$  is exactly singular,  
so the solution could not be computed.

---

## ?gbsvx

*Computes the solution to the real or complex system of linear equations with a band matrix A and multiple right-hand sides, and provides error bounds on the solution.*

---

```
call sgbsvx (fact, trans, n, kl, ku, nrhs, ab, ldab, afb, ldafb,
             ipiv, equed, r, c, b, ldb, x, idx, rcond, ferr, berr,
             work, iwork, info)
call dgbsvx (fact, trans, n, kl, ku, nrhs, ab, ldab, afb, ldafb,
             ipiv, equed, r, c, b, ldb, x, idx, rcond, ferr, berr,
             work, iwork, info)
call cgbsvx (fact, trans, n, kl, ku, nrhs, ab, ldab, afb, ldafb,
             ipiv, equed, r, c, b, ldb, x, idx, rcond, ferr, berr,
             work, rwork, info)
call zgbsvx (fact, trans, n, kl, ku, nrhs, ab, ldab, afb, ldafb,
             ipiv, equed, r, c, b, ldb, x, idx, rcond, ferr, berr,
             work, rwork, info)
```

### Discussion

This routine uses the *LU* factorization to compute the solution to a real or complex system of linear equations  $AX = B$ ,  $A^T X = B$ , or  $A^H X = B$ , where  $A$  is a band matrix of order `n` with `kl` subdiagonals and `ku` superdiagonals, the columns of matrix  $B$  are individual right-hand sides, and the columns of  $X$  are the corresponding solutions.

Error bounds on the solution and a condition estimate are also provided.

The routine `?gbsvx` performs the following steps:

1. If `fact` = '`E`', real scaling factors `r` and `c` are computed to equilibrate the system:

```
trans = 'N': diag(r)*A*diag(c) *diag(c)-1*X = diag(r)*B
trans = 'T': (diag(r)*A*diag(c))T *diag(r)-1*X = diag(c)*B
trans = 'C': (diag(r)*A*diag(c))H *diag(r)-1*X = diag(c)*B
```

Whether or not the system will be equilibrated depends on the scaling of the matrix  $A$ , but if equilibration is used,  $A$  is overwritten by  $\text{diag}(r)*A*\text{diag}(c)$  and  $B$  by  $\text{diag}(r)*B$  (if  $\text{trans}='N'$ ) or  $\text{diag}(c)*B$  (if  $\text{trans} = 'T'$  or ' $C$ ').

2. If  $\text{fact} = 'N'$  or ' $E$ ', the  $LU$  decomposition is used to factor the matrix  $A$  (after equilibration if  $\text{fact} = 'E'$ ) as  $A = L U$ , where  $L$  is a product of permutation and unit lower triangular matrices with  $k_l$  subdiagonals, and  $U$  is upper triangular with  $k_l+k_u$  superdiagonals.
3. If some  $U_{i,i} = 0$ , so that  $U$  is exactly singular, then the routine returns with  $\text{info} = i$ . Otherwise, the factored form of  $A$  is used to estimate the condition number of the matrix  $A$ . If the reciprocal of the condition number is less than machine precision,  $\text{info} = n + 1$  is returned as a warning, but the routine still goes on to solve for  $X$  and compute error bounds as described below.
4. The system of equations is solved for  $X$  using the factored form of  $A$ .
5. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.
6. If equilibration was used, the matrix  $X$  is premultiplied by  $\text{diag}(c)$  (if  $\text{trans} = 'N'$ ) or  $\text{diag}(r)$  (if  $\text{trans} = 'T'$  or ' $C$ ') so that it solves the original system before equilibration.

## Input Parameters

*fact*                    CHARACTER\*1. Must be ' $F$ ', ' $N$ ', or ' $E$ '.

Specifies whether or not the factored form of the matrix  $A$  is supplied on entry, and if not, whether the matrix  $A$  should be equilibrated before it is factored.

If  $\text{fact} = 'F'$ : on entry, *afb* and *ipiv* contain the factored form of  $A$ . If *equed* is not ' $N$ ', the matrix  $A$  has been equilibrated with scaling factors given by *r* and *c*. *ab*, *afb*, and *ipiv* are not modified.

If  $\text{fact} = 'N'$ , the matrix  $A$  will be copied to *afb* and factored.

If  $\text{fact} = 'E'$ , the matrix  $A$  will be equilibrated if necessary, then copied to *afb* and factored.

*trans*                    CHARACTER\*1. Must be ' $N$ ', ' $T$ ', or ' $C$ '.

Specifies the form of the system of equations:

If  $\text{trans} = \text{'N'}$ , the system has the form  $A X = B$   
(No transpose);

If  $\text{trans} = \text{'T'}$ , the system has the form  $A^T X = B$   
(Transpose);

If  $\text{trans} = \text{'C'}$ , the system has the form  $A^H X = B$   
(Conjugate transpose);

$n$  **INTEGER.** The number of linear equations; the order of  
the matrix  $A$  ( $n \geq 0$ ).

$kl$  **INTEGER.** The number of sub-diagonals within the  
band of  $A$  ( $kl \geq 0$ ).

$ku$  **INTEGER.** The number of super-diagonals within the  
band of  $A$  ( $ku \geq 0$ ).

$nrhs$  **INTEGER.** The number of right hand sides; the number  
of columns of the matrices  $B$  and  $X$  ( $nrhs \geq 0$ ).

$ab, afb, b, work$  **REAL** for **s gesvx**

**DOUBLE PRECISION** for **d gesvx**

**COMPLEX** for **c gesvx**

**DOUBLE COMPLEX** for **z gesvx**.

Arrays:  $a(1..n, 1..n)$ ,  $af(1..n, 1..n)$ ,  $b(1..nrhs, 1..n)$ ,  
 $work(1..n * nrhs)$ .

The array  $ab$  contains the matrix  $A$  in band storage  
(see [Matrix Storage Schemes](#)).

The second dimension of  $ab$  must be at least  $\max(1, n)$ .

If  $\text{fact} = \text{'F'}$  and  $\text{equed}$  is not ' $N$ ', then  $A$  must have  
been equilibrated by the scaling factors in  $r$  and/or  $c$ .

The array  $afb$  is an input argument if  $\text{fact} = \text{'F'}$ .

The second dimension of  $afb$  must be at least  $\max(1, n)$ .

It contains the factored form of the matrix  $A$ , i.e., the  
factors  $L$  and  $U$  from the factorization  $A = L U$  as  
computed by [?gbtrf](#).  $U$  is stored as an upper triangular  
band matrix with  $kl + ku$  super-diagonals in the first  
 $1 + kl + ku$  rows of  $afb$ . The multipliers used during

the factorization are stored in the next  $k_1$  rows.

If  $\text{equed}$  is not 'N', then  $\text{afb}$  is the factored form of the equilibrated matrix  $A$ .

The array  $b$  contains the matrix  $B$  whose columns are the right-hand sides for the systems of equations. The second dimension of  $b$  must be at least  $\max(1, nrhs)$ .

$\text{work}(*)$  is a workspace array.

The dimension of  $\text{work}$  must be at least  $\max(1, 3 * n)$  for real flavors, and at least  $\max(1, 2 * n)$  for complex flavors.

$ldab$

INTEGER. The first dimension of  $ab$ ;  $ldab \geq k_1 + ku + 1$ .

$ldafb$

INTEGER. The first dimension of  $afb$ ;  
 $ldafb \geq 2 * k_1 + ku + 1$ .

$ldb$

INTEGER. The first dimension of  $b$ ;  $ldb \geq \max(1, n)$ .

$ipiv$

INTEGER.

Array, DIMENSION at least  $\max(1, n)$ .

The array  $ipiv$  is an input argument if  $\text{fact} = 'F'$ .

It contains the pivot indices from the factorization  $A = L U$  as computed by [?gbtrf](#); row  $i$  of the matrix was interchanged with row  $ipiv(i)$ .

$equed$

CHARACTER\*1. Must be 'N', 'R', 'C', or 'B'.

$equed$  is an input argument if  $\text{fact} = 'F'$ . It specifies the form of equilibration that was done:

If  $equed = 'N'$ , no equilibration was done (always true if  $\text{fact} = 'N'$ );

If  $equed = 'R'$ , row equilibration was done and  $A$  has been premultiplied by  $\text{diag}(r)$ ;

If  $equed = 'C'$ , column equilibration was done and  $A$  has been postmultiplied by  $\text{diag}(c)$ ;

If  $equed = 'B'$ , both row and column equilibration was done;  $A$  has been replaced by  $\text{diag}(r) * A * \text{diag}(c)$ .

$r, c$

REAL for single precision flavors;

DOUBLE PRECISION for double precision flavors.

Arrays:  $r(n)$ ,  $c(n)$ .

The array  $r$  contains the row scale factors for  $A$ , and the

array  $c$  contains the column scale factors for  $A$ . These arrays are input arguments if  $\text{fact} = \text{'F'}$  only; otherwise they are output arguments.

If  $\text{equed} = \text{'R'}$  or  $\text{'B'}$ ,  $A$  is multiplied on the left by  $\text{diag}(r)$ ; if  $\text{equed} = \text{'N'}$  or  $\text{'C'}$ ,  $r$  is not accessed.

If  $\text{fact} = \text{'F'}$  and  $\text{equed} = \text{'R'}$  or  $\text{'B'}$ , each element of  $r$  must be positive.

If  $\text{equed} = \text{'C'}$  or  $\text{'B'}$ ,  $A$  is multiplied on the right by  $\text{diag}(c)$ ; if  $\text{equed} = \text{'N'}$  or  $\text{'R'}$ ,  $c$  is not accessed.

If  $\text{fact} = \text{'F'}$  and  $\text{equed} = \text{'C'}$  or  $\text{'B'}$ , each element of  $c$  must be positive.

*lidx* INTEGER. The first dimension of the output array  $x$ ;  $\text{lidx} \geq \max(1, n)$ .

*iwork* INTEGER. Workspace array, DIMENSION at least  $\max(1, n)$ ; used in real flavors only.

*rwork* REAL for single precision flavors;  
DOUBLE PRECISION for double precision flavors.  
Workspace array, DIMENSION at least  $\max(1, n)$ ; used in complex flavors only.

## Output Parameters

*x* REAL for `sgbsvx`  
DOUBLE PRECISION for `dgbsvx`  
COMPLEX for `cgbsvx`  
DOUBLE COMPLEX for `zgbsvx`.  
Array, DIMENSION (*lidx*, \*).

If  $\text{info} = 0$  or  $\text{info} = n+1$ , the array  $x$  contains the solution matrix  $X$  to the *original* system of equations. Note that  $A$  and  $B$  are modified on exit if  $\text{equed} \neq \text{'N'}$ , and the solution to the *equilibrated* system is:

$\text{diag}(c)^{-1} * X$ , if  $\text{trans} = \text{'N'}$  and  $\text{equed} = \text{'C'}$  or  $\text{'B'}$ ;  
 $\text{diag}(r)^{-1} * X$ , if  $\text{trans} = \text{'T'}$  or  $\text{'C'}$  and  $\text{equed} = \text{'R'}$  or  $\text{'B'}$ .

The second dimension of  $x$  must be at least  $\max(1, nrhs)$ .

---

<i>ab</i>	Array <i>ab</i> is not modified on exit if <i>fact</i> = 'F' or 'N', or if <i>fact</i> = 'E' and <i>equed</i> = 'N'. If <i>equed</i> ≠ 'N', <i>A</i> is scaled on exit as follows: <i>equed</i> = 'R': $A = \text{diag}(r) * A$ <i>equed</i> = 'C': $A = A * \text{diag}(c)$ <i>equed</i> = 'B': $A = \text{diag}(r) * A * \text{diag}(c)$
<i>afb</i>	If <i>fact</i> = 'N' or 'E', then <i>afb</i> is an output argument and on exit returns details of the LU factorization of the original matrix <i>A</i> (if <i>fact</i> = 'N') or of the equilibrated matrix <i>A</i> (if <i>fact</i> = 'E'). See the description of <i>ab</i> for the form of the equilibrated matrix.
<i>b</i>	Overwritten by $\text{diag}(r) * b$ if <i>trans</i> = 'N' and <i>equed</i> = 'R' or 'B'; overwritten by $\text{diag}(c) * b$ if <i>trans</i> = 'T' and <i>equed</i> = 'C' or 'B'; not changed if <i>equed</i> = 'N'.
<i>r, c</i>	These arrays are output arguments if <i>fact</i> ≠ 'F'. See the description of <i>r, c</i> in <i>Input Arguments</i> section.
<i>rcond</i>	<b>REAL</b> for single precision flavors. <b>DOUBLE PRECISION</b> for double precision flavors. An estimate of the reciprocal condition number of the matrix <i>A</i> after equilibration (if done). If <i>rcond</i> is less than the machine precision (in particular, if <i>rcond</i> = 0), the matrix is singular to working precision. This condition is indicated by a return code of <i>info</i> > 0.
<i>ferr, berr</i>	<b>REAL</b> for single precision flavors. <b>DOUBLE PRECISION</b> for double precision flavors. Arrays, <b>DIMENSION</b> at least $\max(1, n_{rhs})$ . Contain the component-wise forward and relative backward errors, respectively, for each solution vector.
<i>ipiv</i>	If <i>fact</i> = 'N' or 'E', then <i>ipiv</i> is an output argument and on exit contains the pivot indices from the factorization $A = L U$ of the original matrix <i>A</i> (if <i>fact</i> = 'N') or of the equilibrated matrix <i>A</i> (if <i>fact</i> = 'E').

*equed*

If  $\text{fact} \neq \text{'F'}$ , then *equed* is an output argument. It specifies the form of equilibration that was done (see the description of *equed* in *Input Arguments* section).

*work, rwork*

On exit, *work*(1) for real flavors, or *rwork*(1) for complex flavors, contains the reciprocal pivot growth factor  $\text{norm}(A)/\text{norm}(U)$ . The "max absolute element" norm is used. If *work*(1) for real flavors, or *rwork*(1) for complex flavors is much less than 1, then the stability of the *LU* factorization of the (equilibrated) matrix *A* could be poor. This also means that the solution *x*, condition estimator *rcond*, and forward error bound *ferr* could be unreliable. If factorization fails with  $0 < \text{info} \leq n$ , then *work*(1) for real flavors, or *rwork*(1) for complex flavors contains the reciprocal pivot growth factor for the leading *info* columns of *A*.

*info*

**INTEGER.** If *info*=0, the execution is successful.

If *info* = *-i*, the *i*th parameter had an illegal value.

If *info* = *i*, and *i* ≤ *n*, then  $U(i,i)$  is exactly zero. The factorization has been completed, but the factor *U* is exactly singular, so the solution and error bounds could not be computed; *rcond* = 0 is returned.

If *info* = *i*, and *i* = *n* +1, then *U* is nonsingular, but *rcond* is less than machine precision, meaning that the matrix is singular to working precision. Nevertheless, the solution and error bounds are computed because there are a number of situations where the computed solution can be more accurate than the value of *rcond* would suggest.

## ?gtsv

*Computes the solution to the system of linear equations with a tridiagonal matrix A and multiple right-hand sides.*

```
call sgtsv (n, nrhs, dl, d, du, b, ldb, info)
call dgtsv (n, nrhs, dl, d, du, b, ldb, info)
call cgtsv (n, nrhs, dl, d, du, b, ldb, info)
call zgtsv (n, nrhs, dl, d, du, b, ldb, info)
```

### Discussion

This routine solves for  $X$  the system of linear equations  $AX = B$ , where  $A$  is an  $n$ -by- $n$  tridiagonal matrix, the columns of matrix  $B$  are individual right-hand sides, and the columns of  $X$  are the corresponding solutions. The routine uses Gaussian elimination with partial pivoting.

Note that the equation  $A^T X = B$  may be solved by interchanging the order of the arguments  $du$  and  $dl$ .

### Input Parameters

$n$	<b>INTEGER.</b> The order of $A$ ; the number of rows in $B$ ( $n \geq 0$ ).
$nrhs$	<b>INTEGER.</b> The number of right-hand sides; the number of columns in $B$ ( $nrhs \geq 0$ ).
$dl, d, du, b$	<b>REAL</b> for <b>sgtsv</b> <b>DOUBLE PRECISION</b> for <b>dgtsv</b> <b>COMPLEX</b> for <b>cgtsv</b> <b>DOUBLE COMPLEX</b> for <b>zgtsv</b> . Arrays: $dl(n - 1), d(n), du(n - 1), b(ldb, *)$ . The array $dl$ contains the ( $n - 1$ ) subdiagonal elements of $A$ . The array $d$ contains the diagonal elements of $A$ . The array $du$ contains the ( $n - 1$ ) superdiagonal elements of $A$ .

The array  $b$  contains the matrix  $B$  whose columns are the right-hand sides for the systems of equations.

The second dimension of  $b$  must be at least  $\max(1, nrhs)$ .

$ldb$  **INTEGER.** The first dimension of  $b$ ;  $ldb \geq \max(1, n)$ .

### Output Parameters

$d1$	Overwritten by the $(n-2)$ elements of the second superdiagonal of the upper triangular matrix $U$ from the $LU$ factorization of $A$ . These elements are stored in $d1(1), \dots, d1(n-2)$ .
$d$	Overwritten by the $n$ diagonal elements of $U$ .
$du$	Overwritten by the $(n-1)$ elements of the first superdiagonal of $U$ .
$b$	Overwritten by the solution matrix $X$ .
$info$	<b>INTEGER.</b> If $info=0$ , the execution is successful. If $info = -i$ , the $i$ th parameter had an illegal value. If $info = i$ , $U(i,i)$ is exactly zero, and the solution has not been computed. The factorization has not been completed unless $i = n$ .

## ?gtsvx

*Computes the solution to the real or complex system of linear equations with a tridiagonal matrix A and multiple right-hand sides, and provides error bounds on the solution.*

```
call sgtsvx (fact, trans, n, nrhs, dl, d, du, dlf, df, duf, du2,
             ipiv, b, ldb, x, idx, rcond, ferr, berr, work,
             iwork, info)
call dgtsvx (fact, trans, n, nrhs, dl, d, du, dlf, df, duf, du2,
             ipiv, b, ldb, x, idx, rcond, ferr, berr, work,
             iwork, info)
call cgtsvx (fact, trans, n, nrhs, dl, d, du, dlf, df, duf, du2,
             ipiv, b, ldb, x, idx, rcond, ferr, berr, work,
             rwork, info)
call zgtsvx (fact, trans, n, nrhs, dl, d, du, dlf, df, duf, du2,
             ipiv, b, ldb, x, idx, rcond, ferr, berr, work,
             rwork, info)
```

### Discussion

This routine uses the *LU* factorization to compute the solution to a real or complex system of linear equations  $AX = B$ ,  $A^T X = B$ , or  $A^H X = B$ , where  $A$  is a tridiagonal matrix of order  $n$ , the columns of matrix  $B$  are individual right-hand sides, and the columns of  $X$  are the corresponding solutions.

Error bounds on the solution and a condition estimate are also provided.

The routine `?gtsvx` performs the following steps:

1. If  $\text{fact} = \text{'N'}$ , the *LU* decomposition is used to factor the matrix  $A$  as  $A = LU$ , where  $L$  is a product of permutation and unit lower bidiagonal matrices and  $U$  is an upper triangular matrix with nonzeros in only the main diagonal and first two superdiagonals.
2. If some  $U_{i,i} = 0$ , so that  $U$  is exactly singular, then the routine returns with  $\text{info} = i$ . Otherwise, the factored form of  $A$  is used to estimate the condition number of the matrix  $A$ . If the reciprocal of the condition number

is less than machine precision,  $\text{info} = n + 1$  is returned as a warning, but the routine still goes on to solve for  $X$  and compute error bounds as described below.

3. The system of equations is solved for  $X$  using the factored form of  $A$ .
4. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.

### Input Parameters

<i>fact</i>	<b>CHARACTER*1</b> . Must be ' <b>F</b> ' or ' <b>N</b> '.
	Specifies whether or not the factored form of the matrix $A$ has been supplied on entry.  If <i>fact</i> = ' <b>Fd1f, <i>df</i>, <i>duf</i>, <i>du2</i>, and <i>ipiv</i> contain the factored form of <math>A</math>; arrays <i>d1</i>, <i>d</i>, <i>du</i>, <i>d1f</i>, <i>df</i>, <i>duf</i>, <i>du2</i>, and <i>ipiv</i> will not be modified.</b>
	If <i>fact</i> = ' <b>N</b> ', the matrix $A$ will be copied to <i>d1f</i> , <i>df</i> , and <i>duf</i> and factored.
<i>trans</i>	<b>CHARACTER*1</b> . Must be ' <b>N</b> ', ' <b>T</b> ', or ' <b>C</b> '.
	Specifies the form of the system of equations:  If <i>trans</i> = ' <b>N</b> ', the system has the form $A X = B$ (No transpose); If <i>trans</i> = ' <b>T</b> ', the system has the form $A^T X = B$ (Transpose); If <i>trans</i> = ' <b>C</b> ', the system has the form $A^H X = B$ (Conjugate transpose);
<i>n</i>	<b>INTEGER</b> . The number of linear equations; the order of the matrix $A$ ( $n \geq 0$ ).
<i>nrhs</i>	<b>INTEGER</b> . The number of right hand sides; the number of columns of the matrices $B$ and $X$ ( $nrhs \geq 0$ ).
<i>d1, d, du, d1f, df,</i> <i>duf, du2, b, x, work</i>	REAL for <b>sgtsvx</b> DOUBLE PRECISION for <b>dgtsvx</b> COMPLEX for <b>cgtsvx</b> DOUBLE COMPLEX for <b>zgtsvx</b> .
Arrays:	

*dl*, dimension (*n* - 1), contains the subdiagonal elements of *A*.

*d*, dimension (*n*), contains the diagonal elements of *A*.

*du*, dimension (*n* - 1), contains the superdiagonal elements of *A*.

*d1f*, dimension (*n* - 1). If *fact* = 'F', then *d1f* is an input argument and on entry contains the (*n* - 1) multipliers that define the matrix *L* from the *LU* factorization of *A* as computed by [?gttrf](#).

*df*, dimension (*n*). If *fact* = 'F', then *df* is an input argument and on entry contains the *n* diagonal elements of the upper triangular matrix *U* from the *LU* factorization of *A*.

*duf*, dimension (*n* - 1). If *fact* = 'F', then *duf* is an input argument and on entry contains the (*n* - 1) elements of the first super-diagonal of *U*.

*du2*, dimension (*n* - 2). If *fact* = 'F', then *du2* is an input argument and on entry contains the (*n* - 2) elements of the second super-diagonal of *U*.

*b*(*ldb*, \*) contains the right-hand side matrix *B*. The second dimension of *b* must be at least max(1,*nrhs*).

*x*(*idx*, \*) contains the solution matrix *X*. The second dimension of *x* must be at least max(1,*nrhs*).

*work* (\*) is a workspace array;

the dimension of *work* must be at least max(1, 3 \* *n*) for real flavors and max(1, 2 \* *n*) for complex flavors.

*ldb* INTEGER. The first dimension of *b*; *ldb*  $\geq \max(1, n)$ .

*idx* INTEGER. The first dimension of *x*; *idx*  $\geq \max(1, n)$ .

*ipiv* INTEGER.

Array, DIMENSION at least max(1,*n*). If *fact* = 'F', then *ipiv* is an input argument and on entry contains the pivot indices, as returned by [?gttrf](#).

*iwork* INTEGER.

Workspace array, DIMENSION (*n*). Used for real flavors only.

<i>rwork</i>	<small>REAL for <code>cgtsvx</code> DOUBLE PRECISION for <code>zgtsvx</code>. Workspace array, DIMENSION (<i>n</i>). Used for complex flavors only.</small>
<b>Output Parameters</b>	
<i>x</i>	<small>REAL for <code>sgtsvx</code> DOUBLE PRECISION for <code>dgtsvx</code> COMPLEX for <code>cgtsvx</code> DOUBLE COMPLEX for <code>zgtsvx</code>. Array, DIMENSION (<i>lidx</i>, *). If <i>info</i> = 0 or <i>info</i> = <i>n</i>+1, the array <i>x</i> contains the solution matrix <i>X</i>. The second dimension of <i>x</i> must be at least max(1,<i>nrhs</i>). If <i>fact</i> = 'N', then <i>dlf</i> is an output argument and on exit contains the (<i>n</i> - 1) multipliers that define the matrix <i>L</i> from the <i>LU</i> factorization of <i>A</i>.</small>
<i>dlf</i>	<small>If <i>fact</i> = 'N', then <i>dlf</i> is an output argument and on exit contains the <i>n</i> diagonal elements of the upper triangular matrix <i>U</i> from the <i>LU</i> factorization of <i>A</i>.</small>
<i>df</i>	<small>If <i>fact</i> = 'N', then <i>df</i> is an output argument and on exit contains the (<i>n</i> - 1) elements of the first super-diagonal of <i>U</i>.</small>
<i>duf</i>	<small>If <i>fact</i> = 'N', then <i>duf</i> is an output argument and on exit contains the (<i>n</i> - 2) elements of the second super-diagonal of <i>U</i>.</small>
<i>du2</i>	<small>If <i>fact</i> = 'N', then <i>du2</i> is an output argument and on exit contains the pivot indices from the factorization <i>A</i> = <i>L</i> <i>U</i>; row <i>i</i> of the matrix was interchanged with row <i>ipiv(i)</i>. The value of <i>ipiv(i)</i> will always be either <i>i</i> or <i>i</i>+1; <i>ipiv(i)=i</i> indicates a row interchange was not required.</small>
<i>ipiv</i>	<small>The array <i>ipiv</i> is an output argument if <i>fact</i> = 'N' and, on exit, contains the pivot indices from the factorization <i>A</i> = <i>L</i> <i>U</i>; row <i>i</i> of the matrix was interchanged with row <i>ipiv(i)</i>. The value of <i>ipiv(i)</i> will always be either <i>i</i> or <i>i</i>+1; <i>ipiv(i)=i</i> indicates a row interchange was not required.</small>
<i>rcond</i>	<small>REAL for single precision flavors. DOUBLE PRECISION for double precision flavors. An estimate of the reciprocal condition number of the</small>

matrix  $A$ .

If  $rcond$  is less than the machine precision (in particular, if  $rcond = 0$ ), the matrix is singular to working precision. This condition is indicated by a return code of  $info > 0$ .

*ferr, berr*

**REAL** for single precision flavors.

**DOUBLE PRECISION** for double precision flavors.

Arrays, **DIMENSION** at least  $\max(1, nrhs)$ . Contain the component-wise forward and backward errors, respectively, for each solution vector.

*info*

**INTEGER**. If  $info=0$ , the execution is successful.

If  $info = -i$ , the  $i$ th parameter had an illegal value.

If  $info = i$ , and  $i \leq n$ , then  $U(i,i)$  is exactly zero. The factorization has not been completed unless  $i = n$ , but the factor  $U$  is exactly singular, so the solution and error bounds could not be computed;  $rcond = 0$  is returned.

If  $info = i$ , and  $i = n + 1$ , then  $U$  is nonsingular, but  $rcond$  is less than machine precision, meaning that the matrix is singular to working precision. Nevertheless, the solution and error bounds are computed because there are a number of situations where the computed solution can be more accurate than the value of  $rcond$  would suggest.

---

## ?posv

*Computes the solution to the system of linear equations with a symmetric or Hermitian positive definite matrix A and multiple right-hand sides.*

---

```
call sposv (uplo, n, nrhs, a, lda, b, ldb, info)
call dposv (uplo, n, nrhs, a, lda, b, ldb, info)
call cposv (uplo, n, nrhs, a, lda, b, ldb, info)
call zposv (uplo, n, nrhs, a, lda, b, ldb, info)
```

### Discussion

This routine solves for  $X$  the real or complex system of linear equations  $AX = B$ , where  $A$  is an  $n$ -by- $n$  symmetric/Hermitian positive definite matrix, the columns of matrix  $B$  are individual right-hand sides, and the columns of  $X$  are the corresponding solutions.

The Cholesky decomposition is used to factor  $A$  as  $A = U^H U$  if  $\text{uplo} = \text{'U'}$  or  $A = LL^H$  if  $\text{uplo} = \text{'L'}$ , where  $U$  is an upper triangular matrix and  $L$  is a lower triangular matrix. The factored form of  $A$  is then used to solve the system of equations  $AX = B$ .

### Input Parameters

<i>uplo</i>	CHARACTER*1. Must be ' $U$ ' or ' $L$ '. Indicates whether the upper or lower triangular part of $A$ is stored and how $A$ is factored: If $\text{uplo} = \text{'U'}$ , the array $a$ stores the upper triangular part of the matrix $A$ , and $A$ is factored as $U^H U$ . If $\text{uplo} = \text{'L'}$ , the array $a$ stores the lower triangular part of the matrix $A$ ; $A$ is factored as $LL^H$ .
<i>n</i>	INTEGER. The order of matrix $A$ ( $n \geq 0$ ).
<i>nrhs</i>	INTEGER. The number of right-hand sides; the number of columns in $B$ ( $nrhs \geq 0$ ).

<i>a</i> , <i>b</i>	<small>REAL for sposv DOUBLE PRECISION for dposv COMPLEX for cposv DOUBLE COMPLEX for zposv.</small> Arrays: <i>a</i> ( <i>lda</i> , *), <i>b</i> ( <i>ldb</i> , *).
	The array <i>a</i> contains either the upper or the lower triangular part of the matrix <i>A</i> (see <i>uplo</i> ). The second dimension of <i>a</i> must be at least $\max(1, n)$ .
	The array <i>b</i> contains the matrix <i>B</i> whose columns are the right-hand sides for the systems of equations. The second dimension of <i>b</i> must be at least $\max(1, nrhs)$ .
<i>lda</i>	<small>INTEGER.</small> The first dimension of <i>a</i> ; <i>lda</i> $\geq \max(1, n)$ .
<i>ldb</i>	<small>INTEGER.</small> The first dimension of <i>b</i> ; <i>ldb</i> $\geq \max(1, n)$ .

### Output Parameters

<i>a</i>	If <i>info</i> =0, the upper or lower triangular part of <i>a</i> is overwritten by the Cholesky factor <i>U</i> or <i>L</i> , as specified by <i>uplo</i> .
<i>b</i>	Overwritten by the solution matrix <i>X</i> .
<i>info</i>	<small>INTEGER.</small> If <i>info</i> =0, the execution is successful. If <i>info</i> = <i>-i</i> , the <i>i</i> th parameter had an illegal value. If <i>info</i> = <i>i</i> , the leading minor of order <i>i</i> (and hence the matrix <i>A</i> itself) is not positive definite, so the factorization could not be completed, and the solution has not been computed.

## ?posvx

*Uses the Cholesky factorization to compute the solution to the system of linear equations with a symmetric or Hermitian positive definite matrix A, and provides error bounds on the solution.*

---

```
call sposvx (fact, uplo, n, nrhs, a, lda, af, ldaf, equed, s, b,
            ldb, x, ldx, rcond, ferr, berr, work, iwork, info)
call dposvx (fact, uplo, n, nrhs, a, lda, af, ldaf, equed, s, b,
            ldb, x, ldx, rcond, ferr, berr, work, iwork, info)
call cposvx (fact, uplo, n, nrhs, a, lda, af, ldaf, equed, s, b,
            ldb, x, ldx, rcond, ferr, berr, work, rwork, info)
call zposvx (fact, uplo, n, nrhs, a, lda, af, ldaf, equed, s, b,
            ldb, x, ldx, rcond, ferr, berr, work, rwork, info)
```

### Discussion

This routine uses the Cholesky factorization  $A=U^H U$  or  $A=LL^H$  to compute the solution to a real or complex system of linear equations  $AX = B$ , where  $A$  is a *n*-by-*n* real symmetric/Hermitian positive definite matrix, the columns of matrix  $B$  are individual right-hand sides, and the columns of  $X$  are the corresponding solutions.

Error bounds on the solution and a condition estimate are also provided.

The routine ?posvx performs the following steps:

1. If *fact* = 'E', real scaling factors *s* are computed to equilibrate the system:

$$\text{diag}(\mathbf{s}) * \mathbf{A} * \text{diag}(\mathbf{s}) * \text{diag}(\mathbf{s})^{-1} * \mathbf{X} = \text{diag}(\mathbf{s}) * \mathbf{B}$$

Whether or not the system will be equilibrated depends on the scaling of the matrix  $A$ , but if equilibration is used,  $A$  is overwritten by  $\text{diag}(\mathbf{s}) * \mathbf{A} * \text{diag}(\mathbf{s})$  and  $B$  by  $\text{diag}(\mathbf{s}) * \mathbf{B}$ .

2. If *fact* = 'N' or 'E', the Cholesky decomposition is used to factor the matrix  $A$  (after equilibration if *fact* = 'E') as

$A = U^H U$ , if  $\text{uplo} = \text{'U'}$ , or

$A = L L^H$ , if  $\text{uplo} = \text{'L'}$ ,

where  $U$  is an upper triangular matrix and  $L$  is a lower triangular matrix.

3. If the leading  $i$ -by- $i$  principal minor is not positive definite, then the routine returns with  $\text{info} = i$ . Otherwise, the factored form of  $A$  is used to estimate the condition number of the matrix  $A$ . If the reciprocal of the condition number is less than machine precision,  $\text{info} = n + 1$  is returned as a warning, but the routine still goes on to solve for  $X$  and compute error bounds as described below.
4. The system of equations is solved for  $X$  using the factored form of  $A$ .
5. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.
6. If equilibration was used, the matrix  $X$  is premultiplied by  $\text{diag}(s)$  so that it solves the original system before equilibration.

### Input Parameters

***fact***      CHARACTER\*1. Must be '**F**', '**N**', or '**E**'.

Specifies whether or not the factored form of the matrix  $A$  is supplied on entry, and if not, whether the matrix  $A$  should be equilibrated before it is factored.

If ***fact*** = '**F*af*** contains the factored form of  $A$ . If ***equed*** = '**Y**', the matrix  $A$  has been equilibrated with scaling factors given by ***s***.

***a*** and ***af*** will not be modified.

If ***fact*** = '**N**', the matrix  $A$  will be copied to ***af*** and factored.

If ***fact*** = '**E**', the matrix  $A$  will be equilibrated if necessary, then copied to ***af*** and factored.

***uplo***      CHARACTER\*1. Must be '**U**' or '**L**'.

Indicates whether the upper or lower triangular part of  $A$  is stored and how  $A$  is factored:

If ***uplo*** = '**U**', the array ***a*** stores the upper triangular part of the matrix  $A$ , and  $A$  is factored as  $U^H U$ .

If ***uplo*** = '**L**', the array ***a*** stores the lower triangular part of the matrix  $A$ ;  $A$  is factored as  $LL^H$ .

<i>n</i>	INTEGER. The order of matrix <i>A</i> ( <i>n</i> ≥ 0).
<i>nrhs</i>	INTEGER. The number of right-hand sides; the number of columns in <i>B</i> ( <i>nrhs</i> ≥ 0).
<i>a, af, b, work</i>	<p>REAL for <b>sposvx</b>            DOUBLE PRECISION for <b>dposvx</b>            COMPLEX for <b>cposvx</b>            DOUBLE COMPLEX for <b>zposvx</b>.</p> <p>Arrays: <i>a( lda, * )</i>, <i>af( ldaf, * )</i>, <i>b( ldb, * )</i>, <i>work( * )</i>.</p>
	The array <i>a</i> contains the matrix <i>A</i> as specified by <i>uplo</i> . If <i>fact</i> = 'F' and <i>equed</i> = 'Y', then <i>A</i> must have been equilibrated by the scaling factors in <i>s</i> , and <i>a</i> must contain the equilibrated matrix $\text{diag}(s) * A * \text{diag}(s)$ . The second dimension of <i>a</i> must be at least $\max(1, n)$ .
	The array <i>af</i> is an input argument if <i>fact</i> = 'F'. It contains the triangular factor <i>U</i> or <i>L</i> from the Cholesky factorization of <i>A</i> in the same storage format as <i>A</i> . If <i>equed</i> is not 'N', then <i>af</i> is the factored form of the equilibrated matrix $\text{diag}(s) * A * \text{diag}(s)$ . The second dimension of <i>af</i> must be at least $\max(1, n)$ .
	The array <i>b</i> contains the matrix <i>B</i> whose columns are the right-hand sides for the systems of equations. The second dimension of <i>b</i> must be at least $\max(1, nrhs)$ .
	<i>work( * )</i> is a workspace array. The dimension of <i>work</i> must be at least $\max(1, 3 * n)$ for real flavors, and at least $\max(1, 2 * n)$ for complex flavors.
<i>lda</i>	INTEGER. The first dimension of <i>a</i> ; <i>lda</i> ≥ $\max(1, n)$ .
<i>ldaf</i>	INTEGER. The first dimension of <i>af</i> ; <i>ldaf</i> ≥ $\max(1, n)$ .
<i>ldb</i>	INTEGER. The first dimension of <i>b</i> ; <i>ldb</i> ≥ $\max(1, n)$ .
<i>equed</i>	CHARACTER*1. Must be 'N' or 'Y'. <i>equed</i> is an input argument if <i>fact</i> = 'F'. It specifies the form of equilibration that was done: If <i>equed</i> = 'N', no equilibration was done (always

true if  $\text{fact} = \text{'N'}$ ;  
 If  $\text{equed} = \text{'Y'}$ , equilibration was done and  $A$  has been replaced by  $\text{diag}(s) * A * \text{diag}(s)$ .

 $s$ 

**REAL** for single precision flavors;

**DOUBLE PRECISION** for double precision flavors.

Array, **DIMENSION (n)**.

The array  $s$  contains the scale factors for  $A$ . This array is an input argument if  $\text{fact} = \text{'F'}$  only; otherwise it is an output argument.

If  $\text{equed} = \text{'N'}$ ,  $s$  is not accessed.

If  $\text{fact} = \text{'F'}$  and  $\text{equed} = \text{'Y'}$ , each element of  $s$  must be positive.

 $l\alpha$ 

**INTEGER**. The first dimension of the output array  $x$ ;

$l\alpha \geq \max(1, n)$ .

 $i\alpha$ 

**INTEGER**.

Workspace array, **DIMENSION** at least  $\max(1, n)$ ; used in real flavors only.

 $r\alpha$ 

**REAL** for **cposvx**;

**DOUBLE PRECISION** for **zposvx**.

Workspace array, **DIMENSION** at least  $\max(1, n)$ ; used in complex flavors only.

## Output Parameters

 $x$ 

**REAL** for **sposvx**

**DOUBLE PRECISION** for **dposvx**

**COMPLEX** for **cposvx**

**DOUBLE COMPLEX** for **zposvx**.

Array, **DIMENSION (l $\alpha$ , \*)**.

If  $\text{info} = 0$  or  $\text{info} = n+1$ , the array  $x$  contains the solution matrix  $X$  to the *original* system of equations.

Note that if  $\text{equed} = \text{'Y'}$ ,  $A$  and  $B$  are modified on exit, and the solution to the equilibrated system is  $\text{diag}(s)^{-1} * X$ .

The second dimension of  $x$  must be at least  $\max(1, nrhs)$ .

<i>a</i>	Array <i>a</i> is not modified on exit if <i>fact</i> = 'F' or 'N', or if <i>fact</i> = 'E' and <i>equed</i> = 'N'. If <i>fact</i> = 'E' and <i>equed</i> = 'Y', <i>A</i> is overwritten by $\text{diag}(s)^*A^*\text{diag}(s)$
<i>af</i>	If <i>fact</i> = 'N' or 'E', then <i>af</i> is an output argument and on exit returns the triangular factor <i>U</i> or <i>L</i> from the Cholesky factorization $A=U^H U$ or $A=LL^H$ of the original matrix <i>A</i> (if <i>fact</i> = 'N'), or of the equilibrated matrix <i>A</i> (if <i>fact</i> = 'E'). See the description of <i>a</i> for the form of the equilibrated matrix.
<i>b</i>	Overwritten by $\text{diag}(s)^*B$ , if <i>equed</i> = 'Y'; not changed if <i>equed</i> = 'N'.
<i>s</i>	This array is an output argument if <i>fact</i> ≠ 'F'. See the description of <i>s</i> in <i>Input Arguments</i> section.
<i>rcond</i>	<b>REAL</b> for single precision flavors. <b>DOUBLE PRECISION</b> for double precision flavors. An estimate of the reciprocal condition number of the matrix <i>A</i> after equilibration (if done). If <i>rcond</i> is less than the machine precision (in particular, if <i>rcond</i> = 0), the matrix is singular to working precision. This condition is indicated by a return code of <i>info</i> > 0.
<i>ferr</i> , <i>berr</i>	<b>REAL</b> for single precision flavors. <b>DOUBLE PRECISION</b> for double precision flavors. Arrays, <b>DIMENSION</b> at least $\max(1, nrhs)$ . Contain the component-wise forward and relative backward errors, respectively, for each solution vector.
<i>equed</i>	If <i>fact</i> ≠ 'F', then <i>equed</i> is an output argument. It specifies the form of equilibration that was done (see the description of <i>equed</i> in <i>Input Arguments</i> section).
<i>info</i>	<b>INTEGER</b> . If <i>info</i> =0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> th parameter had an illegal value. If <i>info</i> = <i>i</i> , and <i>i</i> ≤ <i>n</i> , the leading minor of order <i>i</i> (and hence the matrix <i>A</i> itself) is not positive definite, so the factorization could not be completed, and the solution and error bounds could not be computed; <i>rcond</i> = 0 is

returned.

If `info` = `i`, and `i` = `n` + 1, then  $U$  is nonsingular, but `rcond` is less than machine precision, meaning that the matrix is singular to working precision. Nevertheless, the solution and error bounds are computed because there are a number of situations where the computed solution can be more accurate than the value of `rcond` would suggest.

## ?ppsv

*Computes the solution to the system of linear equations with a symmetric (Hermitian) positive definite packed matrix A and multiple right-hand sides.*

```
call spps v (uplo, n, nrhs, ap, b, ldb, info)
call dpps v (uplo, n, nrhs, ap, b, ldb, info)
call cpps v (uplo, n, nrhs, ap, b, ldb, info)
call zpps v (uplo, n, nrhs, ap, b, ldb, info)
```

### Discussion

This routine solves for  $X$  the real or complex system of linear equations  $AX = B$ , where  $A$  is an `n`-by-`n` real symmetric/Hermitian positive definite matrix stored in packed format, the columns of matrix  $B$  are individual right-hand sides, and the columns of  $X$  are the corresponding solutions.

The Cholesky decomposition is used to factor  $A$  as  $A = U^H U$  if `uplo` = 'U' or  $A = LL^H$  if `uplo` = 'L', where  $U$  is an upper triangular matrix and  $L$  is a lower triangular matrix. The factored form of  $A$  is then used to solve the system of equations  $AX = B$ .

### Input Parameters

`uplo`                    CHARACTER\*1. Must be 'U' or 'L'.

	Indicates whether the upper or lower triangular part of $A$ is stored and how $A$ is factored:
<i>uplo</i>	If <i>uplo</i> = 'U', the array <i>a</i> stores the upper triangular part of the matrix $A$ , and $A$ is factored as $U^H U$ . If <i>uplo</i> = 'L', the array <i>a</i> stores the lower triangular part of the matrix $A$ ; $A$ is factored as $LL^H$ .
<i>n</i>	<b>INTEGER.</b> The order of matrix $A$ ( <i>n</i> $\geq 0$ ).
<i>nrhs</i>	<b>INTEGER.</b> The number of right-hand sides; the number of columns in $B$ ( <i>nrhs</i> $\geq 0$ ).
<i>ap</i> , <i>b</i>	<b>REAL</b> for <b>sppsv</b> <b>DOUBLE PRECISION</b> for <b>dppsv</b> <b>COMPLEX</b> for <b>cppsv</b> <b>DOUBLE COMPLEX</b> for <b>zppsv</b> . Arrays: <i>ap</i> (*), <i>b</i> ( <i>ldb</i> , *). The array <i>ap</i> contains either the upper or the lower triangular part of the matrix $A$ (as specified by <i>uplo</i> ) in <i>packed storage</i> (see <a href="#">Matrix Storage Schemes</a> ). The dimension of <i>ap</i> must be at least $\max(1, n(n+1)/2)$ . The array <i>b</i> contains the matrix $B$ whose columns are the right-hand sides for the systems of equations. The second dimension of <i>b</i> must be at least $\max(1, \text{nrhs})$ .
<i>ldb</i>	<b>INTEGER.</b> The first dimension of <i>b</i> ; <i>ldb</i> $\geq \max(1, n)$ .

## Output Parameters

<i>ap</i>	If <i>info</i> =0, the upper or lower triangular part of $A$ in packed storage is overwritten by the Cholesky factor $U$ or $L$ , as specified by <i>uplo</i> .
<i>b</i>	Overwritten by the solution matrix $X$ .
<i>info</i>	<b>INTEGER.</b> If <i>info</i> =0, the execution is successful. If <i>info</i> = <i>-i</i> , the <i>i</i> th parameter had an illegal value. If <i>info</i> = <i>i</i> , the leading minor of order <i>i</i> (and hence the matrix $A$ itself) is not positive definite, so the factorization could not be completed, and the solution has not been computed.

## ?ppsvx

*Uses the Cholesky factorization to compute the solution to the system of linear equations with a symmetric (Hermitian) positive definite packed matrix A, and provides error bounds on the solution.*

```
call sppsvx (fact, uplo, n, nrhs, ap, afp, equed, s, b, ldb, x, ldx,
             rcond, ferr, berr, work, iwork, info)
call dppsvx (fact, uplo, n, nrhs, ap, afp, equed, s, b, ldb, x, ldx,
             rcond, ferr, berr, work, iwork, info)
call cppsvx (fact, uplo, n, nrhs, ap, afp, equed, s, b, ldb, x, ldx,
             rcond, ferr, berr, work, rwork, info)
call zppsvx (fact, uplo, n, nrhs, ap, afp, equed, s, b, ldb, x, ldx,
             rcond, ferr, berr, work, rwork, info)
```

### Discussion

This routine uses the Cholesky factorization  $A=U^H U$  or  $A=LL^H$  to compute the solution to a real or complex system of linear equations  $AX=B$ , where  $A$  is a  $n$ -by- $n$  symmetric or Hermitian positive definite matrix stored in packed format, the columns of matrix  $B$  are individual right-hand sides, and the columns of  $X$  are the corresponding solutions.

Error bounds on the solution and a condition estimate are also provided.

The routine `?ppsvx` performs the following steps:

1. If  $\text{fact} = \text{'E'}$ , real scaling factors  $s$  are computed to equilibrate the system:

$$\text{diag}(s)*A*\text{diag}(s) * \text{diag}(s)^{-1} * X = \text{diag}(s)*B$$

Whether or not the system will be equilibrated depends on the scaling of the matrix  $A$ , but if equilibration is used,  $A$  is overwritten by  $\text{diag}(s)*A*\text{diag}(s)$  and  $B$  by  $\text{diag}(s)*B$ .

2. If  $\text{fact} = \text{'N'}$  or  $\text{'E'}$ , the Cholesky decomposition is used to factor the matrix  $A$  (after equilibration if  $\text{fact} = \text{'E'}$ ) as

$$A = U^H U, \text{ if } \text{uplo} = \text{'U'}, \text{ or}$$

$$A = L L^H, \text{ if } \text{uplo} = \text{'L'},$$

where  $U$  is an upper triangular matrix and  $L$  is a lower triangular matrix.

3. If the leading  $i$ -by- $i$  principal minor is not positive definite, then the routine returns with  $\text{info} = i$ . Otherwise, the factored form of  $A$  is used to estimate the condition number of the matrix  $A$ . If the reciprocal of the condition number is less than machine precision,  $\text{info} = n + 1$  is returned as a warning, but the routine still goes on to solve for  $X$  and compute error bounds as described below.
4. The system of equations is solved for  $X$  using the factored form of  $A$ .
5. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.
6. If equilibration was used, the matrix  $X$  is premultiplied by  $\text{diag}(s)$  so that it solves the original system before equilibration.

## Input Parameters

*fact*

CHARACTER\*1. Must be '**F**', '**N**', or '**E**'.

Specifies whether or not the factored form of the matrix  $A$  is supplied on entry, and if not, whether the matrix  $A$  should be equilibrated before it is factored.

If *fact* = '**Fafp contains the factored form of  $A$ . If *equed* = '**Y**', the matrix  $A$  has been equilibrated with scaling factors given by *s*.**

*ap* and *afp* will not be modified.

If *fact* = '**N**', the matrix  $A$  will be copied to *afp* and factored.

If *fact* = '**E**', the matrix  $A$  will be equilibrated if necessary, then copied to *afp* and factored.

*uplo*

CHARACTER\*1. Must be '**U**' or '**L**'.

Indicates whether the upper or lower triangular part of  $A$  is stored and how  $A$  is factored:

If *uplo* = '**U**', the array *ap* stores the upper triangular part of the matrix  $A$ , and  $A$  is factored as  $U^H U$ .

If *uplo* = '**L**', the array *ap* stores the lower triangular part of the matrix  $A$ ;  $A$  is factored as  $LL^H$ .

*n*                    INTEGER. The order of matrix *A* (*n* ≥ 0).

*nrhs*                INTEGER. The number of right-hand sides; the number of columns in *B* (*nrhs* ≥ 0).

*ap, afp, b, work* REAL for *sppsvx*  
 DOUBLE PRECISION for *dppsvx*  
 COMPLEX for *cppsvx*  
 DOUBLE COMPLEX for *zppsvx*.

Arrays: *ap*( \* ), *afp*( \* ), *b*( *ldb*, \* ), *work* ( \* ).

The array *ap* contains the upper or lower triangle of the original symmetric/Hermitian matrix *A* in *packed storage* (see [Matrix Storage Schemes](#)). In case when *fact* = 'F' and *equed* = 'Y', *ap* must contain the equilibrated matrix  $\text{diag}(s) * A * \text{diag}(s)$ .

The array *afp* is an input argument if *fact* = 'F' and contains the triangular factor *U* or *L* from the Cholesky factorization of *A* in the same storage format as *A*. If *equed* is not 'N', then *afp* is the factored form of the equilibrated matrix *A*.

The array *b* contains the matrix *B* whose columns are the right-hand sides for the systems of equations.

*work* ( \* ) is a workspace array.

The dimension of arrays *ap* and *afp* must be at least  $\max(1, n(n+1)/2)$ ; the second dimension of *b* must be at least  $\max(1, nrhs)$ ; the dimension of *work* must be at least  $\max(1, 3*n)$  for real flavors and  $\max(1, 2*n)$  for complex flavors.

*ldb*                INTEGER. The first dimension of *b*; *ldb* ≥ max(1, *n*).

*equed*              CHARACTER\*1. Must be 'N' or 'Y'.

*equed* is an input argument if *fact* = 'F'. It specifies the form of equilibration that was done:

If *equed* = 'N', no equilibration was done (always true if *fact* = 'N');

If *equed* = 'Y', equilibration was done and *A* has been replaced by  $\text{diag}(s) * A * \text{diag}(s)$ .

<i>s</i>	<b>REAL</b> for single precision flavors; <b>DOUBLE PRECISION</b> for double precision flavors. Array, <b>DIMENSION (n)</b> . The array <i>s</i> contains the scale factors for <i>A</i> . This array is an input argument if <i>fact</i> = 'F' only; otherwise it is an output argument. If <i>equed</i> = 'N', <i>s</i> is not accessed. If <i>fact</i> = 'F' and <i>equed</i> = 'Y', each element of <i>s</i> must be positive.
<i>ldx</i>	<b>INTEGER</b> . The first dimension of the output array <i>x</i> ; <i>ldx</i> $\geq \max(1, n)$ .
<i>iwork</i>	<b>INTEGER</b> . Workspace array, <b>DIMENSION</b> at least $\max(1, n)$ ; used in real flavors only.
<i>rwork</i>	<b>REAL</b> for <i>cппsvx</i> ; <b>DOUBLE PRECISION</b> for <i>zппsvx</i> . Workspace array, <b>DIMENSION</b> at least $\max(1, n)$ ; used in complex flavors only.

## Output Parameters

<i>x</i>	<b>REAL</b> for <i>sппsvx</i> <b>DOUBLE PRECISION</b> for <i>dппsvx</i> <b>COMPLEX</b> for <i>cппsvx</i> <b>DOUBLE COMPLEX</b> for <i>zппsvx</i> . Array, <b>DIMENSION (ldx, *)</b> . If <i>info</i> = 0 or <i>info</i> = <i>n</i> +1, the array <i>x</i> contains the solution matrix <i>X</i> to the <i>original</i> system of equations. Note that if <i>equed</i> = 'Y', <i>A</i> and <i>B</i> are modified on exit, and the solution to the equilibrated system is $\text{diag}(s)^{-1} * X$ . The second dimension of <i>x</i> must be at least $\max(1, nrhs)$ .
<i>ap</i>	Array <i>ap</i> is not modified on exit if <i>fact</i> = 'F' or 'N', or if <i>fact</i> = 'E' and <i>equed</i> = 'N'. If <i>fact</i> = 'E' and <i>equed</i> = 'Y', <i>A</i> is overwritten by $\text{diag}(s) * A * \text{diag}(s)$

---

<i>afp</i>	If <i>fact</i> = 'N' or 'E', then <i>afp</i> is an output argument and on exit returns the triangular factor <i>U</i> or <i>L</i> from the Cholesky factorization $A=U^H U$ or $A=LL^H$ of the original matrix <i>A</i> (if <i>fact</i> = 'N'), or of the equilibrated matrix <i>A</i> (if <i>fact</i> = 'E'). See the description of <i>ap</i> for the form of the equilibrated matrix.
<i>b</i>	Overwritten by $\text{diag}(s)*B$ , if <i>equed</i> = 'Y'; not changed if <i>equed</i> = 'N'.
<i>s</i>	This array is an output argument if <i>fact</i> ≠ 'F'. See the description of <i>s</i> in <i>Input Arguments</i> section.
<i>rcond</i>	<i>REAL</i> for single precision flavors. <i>DOUBLE PRECISION</i> for double precision flavors. An estimate of the reciprocal condition number of the matrix <i>A</i> after equilibration (if done). If <i>rcond</i> is less than the machine precision (in particular, if <i>rcond</i> = 0), the matrix is singular to working precision. This condition is indicated by a return code of <i>info</i> > 0.
<i>ferr</i> , <i>berr</i>	<i>REAL</i> for single precision flavors. <i>DOUBLE PRECISION</i> for double precision flavors. Arrays, <i>DIMENSION</i> at least $\max(1, nrhs)$ . Contain the component-wise forward and relative backward errors, respectively, for each solution vector.
<i>equed</i>	If <i>fact</i> ≠ 'F', then <i>equed</i> is an output argument. It specifies the form of equilibration that was done (see the description of <i>equed</i> in <i>Input Arguments</i> section).
<i>info</i>	<i>INTEGER</i> . If <i>info</i> =0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> th parameter had an illegal value. If <i>info</i> = <i>i</i> , and <i>i</i> ≤ <i>n</i> , the leading minor of order <i>i</i> (and hence the matrix <i>A</i> itself) is not positive definite, so the factorization could not be completed, and the solution and error bounds could not be computed; <i>rcond</i> = 0 is returned. If <i>info</i> = <i>i</i> , and <i>i</i> = <i>n</i> + 1, then <i>U</i> is nonsingular, but <i>rcond</i> is less than machine precision, meaning that the matrix is singular to working precision. Nevertheless, the solution and error bounds are computed because there are a number of situations where the computed solution can be more accurate than the value of <i>rcond</i> would suggest.

---

## ?pbsv

*Computes the solution to the system of linear equations with a symmetric or Hermitian positive definite band matrix A and multiple right-hand sides.*

---

```
call spbsv (uplo, n, kd, nrhs, ab, ldbl, b, ldb, info)
call dpbsv (uplo, n, kd, nrhs, ab, ldbl, b, ldb, info)
call cpbsv (uplo, n, kd, nrhs, ab, ldbl, b, ldb, info)
call zpbsv (uplo, n, kd, nrhs, ab, ldbl, b, ldb, info)
```

### Discussion

This routine solves for  $X$  the real or complex system of linear equations  $AX = B$ , where  $A$  is an  $n$ -by- $n$  symmetric/Hermitian positive definite band matrix, the columns of matrix  $B$  are individual right-hand sides, and the columns of  $X$  are the corresponding solutions.

The Cholesky decomposition is used to factor  $A$  as  $A = U^H U$  if  $\text{uplo} = \text{'U'}$  or  $A = LL^H$  if  $\text{uplo} = \text{'L'}$ , where  $U$  is an upper triangular band matrix and  $L$  is a lower triangular band matrix, with the same number of superdiagonals or subdiagonals as  $A$ . The factored form of  $A$  is then used to solve the system of equations  $AX = B$ .

### Input Parameters

<i>uplo</i>	CHARACTER*1. Must be ' <b>U</b> ' or ' <b>L</b> '.
	Indicates whether the upper or lower triangular part of $A$ is stored in the array <i>ab</i> , and how $A$ is factored:
	If <i>uplo</i> = ' <b>U</b> ', the array <i>ab</i> stores the upper triangular part of the matrix $A$ , and $A$ is factored as $U^H U$ .
	If <i>uplo</i> = ' <b>L</b> ', the array <i>ab</i> stores the lower triangular part of the matrix $A$ ; $A$ is factored as $LL^H$ .

*n*

INTEGER. The order of matrix  $A$  ( $n \geq 0$ ).

---

<i>kd</i>	<b>INTEGER.</b> The number of superdiagonals of the matrix $A$ if $\text{uplo} = \text{'U'}$ , or the number of subdiagonals if $\text{uplo} = \text{'L'}$ ( $kd \geq 0$ ).
<i>nrhs</i>	<b>INTEGER.</b> The number of right-hand sides; the number of columns in $B$ ( $nrhs \geq 0$ ).
<i>ab, b</i>	<b>REAL</b> for <code>spbsv</code> <b>DOUBLE PRECISION</b> for <code>dpbsv</code> <b>COMPLEX</b> for <code>cpbsv</code> <b>DOUBLE COMPLEX</b> for <code>zpbsv</code> . Arrays: <i>ab</i> ( <i>ldab, *</i> ), <i>b</i> ( <i>ldb, *</i> ). The array <i>ab</i> contains either the upper or the lower triangular part of the matrix $A$ (as specified by <i>uplo</i> ) in <i>band storage</i> (see <a href="#">Matrix Storage Schemes</a> ). The second dimension of <i>ab</i> must be at least $\max(1, n)$ . The array <i>b</i> contains the matrix $B$ whose columns are the right-hand sides for the systems of equations. The second dimension of <i>b</i> must be at least $\max(1, nrhs)$ .
<i>ldab</i>	<b>INTEGER.</b> The first dimension of the array <i>ab</i> . ( $ldab \geq kd + 1$ )
<i>ldb</i>	<b>INTEGER.</b> The first dimension of <i>b</i> ; $ldb \geq \max(1, n)$ .

## Output Parameters

<i>ab</i>	The upper or lower triangular part of $A$ (in band storage) is overwritten by the Cholesky factor $U$ or $L$ , as specified by <i>uplo</i> , in the same storage format as $A$ .
<i>b</i>	Overwritten by the solution matrix $X$ .
<i>info</i>	<b>INTEGER.</b> If <i>info</i> =0, the execution is successful. If <i>info</i> = <i>-i</i> , the <i>i</i> th parameter had an illegal value. If <i>info</i> = <i>i</i> , the leading minor of order <i>i</i> (and hence the matrix $A$ itself) is not positive definite, so the factorization could not be completed, and the solution has not been computed.

## ?pbsvx

*Uses the Cholesky factorization to compute the solution to the system of linear equations with a symmetric (Hermitian) positive definite band matrix A, and provides error bounds on the solution.*

---

```
call spbsvx (fact, uplo, n, kd, nrhs, ab, ldab, afb, ldafb, equed,
             s, b, ldb, x, ldx, rcond, ferr, berr, work, iwork, info)
call dpbsvx (fact, uplo, n, kd, nrhs, ab, ldab, afb, ldafb, equed,
             s, b, ldb, x, ldx, rcond, ferr, berr, work, iwork, info)
call cpbsvx (fact, uplo, n, kd, nrhs, ab, ldab, afb, ldafb, equed,
             s, b, ldb, x, ldx, rcond, ferr, berr, work, iwork, info)
call zpbsvx (fact, uplo, n, kd, nrhs, ab, ldab, afb, ldafb, equed,
             s, b, ldb, x, ldx, rcond, ferr, berr, work, iwork, info)
```

### Discussion

This routine uses the Cholesky factorization  $A=U^H U$  or  $A=LL^H$  to compute the solution to a real or complex system of linear equations  $AX = B$ , where  $A$  is a *n*-by-*n* symmetric or Hermitian positive definite band matrix, the columns of matrix  $B$  are individual right-hand sides, and the columns of  $X$  are the corresponding solutions.

Error bounds on the solution and a condition estimate are also provided.

The routine ?pbsvx performs the following steps:

1. If *fact* = 'E', real scaling factors *s* are computed to equilibrate the system:

$$\text{diag}(\mathbf{s}) * A * \text{diag}(\mathbf{s}) * \text{diag}(\mathbf{s})^{-1} * X = \text{diag}(\mathbf{s}) * B$$

Whether or not the system will be equilibrated depends on the scaling of the matrix  $A$ , but if equilibration is used,  $A$  is overwritten by  $\text{diag}(\mathbf{s}) * A * \text{diag}(\mathbf{s})$  and  $B$  by  $\text{diag}(\mathbf{s}) * B$ .

2. If *fact* = 'N' or 'E', the Cholesky decomposition is used to factor the matrix  $A$  (after equilibration if *fact* = 'E') as

$A = U^H U$ , if  $\text{uplo} = \text{'U'}$ , or

$A = L L^H$ , if  $\text{uplo} = \text{'L'}$ ,

where  $U$  is an upper triangular band matrix and  $L$  is a lower triangular band matrix.

3. If the leading  $i$ -by- $i$  principal minor is not positive definite, then the routine returns with  $\text{info} = i$ . Otherwise, the factored form of  $A$  is used to estimate the condition number of the matrix  $A$ . If the reciprocal of the condition number is less than machine precision,  $\text{info} = n + 1$  is returned as a warning, but the routine still goes on to solve for  $X$  and compute error bounds as described below.
4. The system of equations is solved for  $X$  using the factored form of  $A$ .
5. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.
6. If equilibration was used, the matrix  $X$  is premultiplied by  $\text{diag}(s)$  so that it solves the original system before equilibration.

### Input Parameters

**fact**                    CHARACTER\*1. Must be '**F**', '**N**', or '**E**'.

Specifies whether or not the factored form of the matrix  $A$  is supplied on entry, and if not, whether the matrix  $A$  should be equilibrated before it is factored.

If **fact** = '**F**' : on entry, **afb** contains the factored form of  $A$ . If **equed** = '**Y**', the matrix  $A$  has been equilibrated with scaling factors given by **s**.

**ab** and **afb** will not be modified.

If **fact** = '**N**', the matrix  $A$  will be copied to **afb** and factored.

If **fact** = '**E**', the matrix  $A$  will be equilibrated if necessary, then copied to **afb** and factored.

**uplo**                    CHARACTER\*1. Must be '**U**' or '**L**'.

Indicates whether the upper or lower triangular part of  $A$  is stored and how  $A$  is factored:

<i>n</i>	If <i>uplo</i> = 'U', the array <i>ab</i> stores the upper triangular part of the matrix <i>A</i> , and <i>A</i> is factored as $U^H U$ . If <i>uplo</i> = 'L', the array <i>ab</i> stores the lower triangular part of the matrix <i>A</i> ; <i>A</i> is factored as $LL^H$ .
<i>kd</i>	<b>INTEGER.</b> The number of super-diagonals or sub-diagonals in the matrix <i>A</i> ( <i>kd</i> ≥ 0).
<i>nrhs</i>	<b>INTEGER.</b> The number of right-hand sides; the number of columns in <i>B</i> ( <i>nrhs</i> ≥ 0).
<i>ab,afb,b,work</i>	<b>REAL</b> for <i>spbsvx</i> <b>DOUBLE PRECISION</b> for <i>dpbsvx</i> <b>COMPLEX</b> for <i>cpbsvx</i> <b>DOUBLE COMPLEX</b> for <i>zpbsvx</i> . Arrays: <i>ab(1dab,*), afb(1dab,*), b(1db,*), work(*)</i> . The array <i>ab</i> contains the upper or lower triangle of the matrix <i>A</i> in band storage (see <a href="#">Matrix Storage Schemes</a> ). If <i>fact</i> = 'F' and <i>equed</i> = 'Y', then <i>ab</i> must contain the equilibrated matrix $\text{diag}(s) * A * \text{diag}(s)$ . The second dimension of <i>ab</i> must be at least $\max(1, n)$ . The array <i>afb</i> is an input argument if <i>fact</i> = 'F'. It contains the triangular factor <i>U</i> or <i>L</i> from the Cholesky factorization of the band matrix <i>A</i> in the same storage format as <i>A</i> . If <i>equed</i> = 'Y', then <i>afb</i> is the factored form of the equilibrated matrix <i>A</i> . The second dimension of <i>afb</i> must be at least $\max(1, n)$ . The array <i>b</i> contains the matrix <i>B</i> whose columns are the right-hand sides for the systems of equations. The second dimension of <i>b</i> must be at least $\max(1, nrhs)$ . <i>work(*)</i> is a workspace array. The dimension of <i>work</i> must be at least $\max(1, 3 * n)$ for real flavors, and at least $\max(1, 2 * n)$ for complex flavors.
<i>ldab</i>	<b>INTEGER.</b> The first dimension of <i>ab</i> ; <i>ldab</i> ≥ <i>kd</i> +1.
<i>ldafb</i>	<b>INTEGER.</b> The first dimension of <i>afb</i> ; <i>ldafb</i> ≥ <i>kd</i> +1.

---

<i>ldb</i>	<code>INTEGER</code> . The first dimension of <i>b</i> ; $ldb \geq \max(1, n)$ .
<i>equed</i>	<code>CHARACTER*1</code> . Must be ' <code>N</code> ' or ' <code>Y</code> '. <i>equed</i> is an input argument if <i>fact</i> = ' <code>F</code> '. It specifies the form of equilibration that was done: If <i>equed</i> = ' <code>N</code> ', no equilibration was done (always true if <i>fact</i> = ' <code>N</code> '); If <i>equed</i> = ' <code>Y</code> ', equilibration was done and <i>A</i> has been replaced by $\text{diag}(s) * A * \text{diag}(s)$ .
<i>s</i>	<code>REAL</code> for single precision flavors; <code>DOUBLE PRECISION</code> for double precision flavors. Array, <code>DIMENSION (n)</code> . The array <i>s</i> contains the scale factors for <i>A</i> . This array is an input argument if <i>fact</i> = ' <code>F</code> ' only; otherwise it is an output argument. If <i>equed</i> = ' <code>N</code> ', <i>s</i> is not accessed. If <i>fact</i> = ' <code>F</code> ' and <i>equed</i> = ' <code>Y</code> ', each element of <i>s</i> must be positive.
<i>ldx</i>	<code>INTEGER</code> . The first dimension of the output array <i>x</i> ; $ldx \geq \max(1, n)$ .
<i>iwork</i>	<code>INTEGER</code> . Workspace array, <code>DIMENSION</code> at least $\max(1, n)$ ; used in real flavors only.
<i>rwork</i>	<code>REAL</code> for <code>cpbsvx</code> ; <code>DOUBLE PRECISION</code> for <code>zpbsvx</code> . Workspace array, <code>DIMENSION</code> at least $\max(1, n)$ ; used in complex flavors only.

## Output Parameters

<i>x</i>	<code>REAL</code> for <code>spbsvx</code> <code>DOUBLE PRECISION</code> for <code>dpbsvx</code> <code>COMPLEX</code> for <code>cpbsvx</code> <code>DOUBLE COMPLEX</code> for <code>zpbsvx</code> . Array, <code>DIMENSION (lidx, *)</code> .
----------	--

If `info` = 0 or `info` = `n`+1, the array `x` contains the solution matrix  $X$  to the *original* system of equations. Note that if `equed` = 'Y',  $A$  and  $B$  are modified on exit, and the solution to the equilibrated system is  $\text{diag}(\mathbf{s})^{-1} * X$ .

The second dimension of `x` must be at least  $\max(1, \mathbf{nrhs})$ .

`ab`

On exit, if `fact` = 'E' and `equed` = 'Y',  $A$  is overwritten by  $\text{diag}(\mathbf{s}) * A * \text{diag}(\mathbf{s})$

`afb`

If `fact` = 'N' or 'E', then `afb` is an output argument and on exit returns the triangular factor  $U$  or  $L$  from the Cholesky factorization  $A = U^H U$  or  $A = LL^H$  of the original matrix  $A$  (if `fact` = 'N'), or of the equilibrated matrix  $A$  (if `fact` = 'E'). See the description of `ab` for the form of the equilibrated matrix.

`b`

Overwritten by  $\text{diag}(\mathbf{s}) * B$ , if `equed` = 'Y'; not changed if `equed` = 'N'.

`s`

This array is an output argument if `fact` ≠ 'F'. See the description of `s` in *Input Arguments* section.

`rcond`

`REAL` for single precision flavors.

`DOUBLE PRECISION` for double precision flavors.

An estimate of the reciprocal condition number of the matrix  $A$  after equilibration (if done). If `rcond` is less than the machine precision (in particular, if `rcond` = 0), the matrix is singular to working precision. This condition is indicated by a return code of `info` > 0.

`ferr, berr`

`REAL` for single precision flavors.

`DOUBLE PRECISION` for double precision flavors.

Arrays, `DIMENSION` at least  $\max(1, \mathbf{nrhs})$ . Contain the component-wise forward and relative backward errors, respectively, for each solution vector.

`equed`

If `fact` ≠ 'F', then `equed` is an output argument. It specifies the form of equilibration that was done (see the description of `equed` in *Input Arguments* section).

---

*info*            INTEGER. If *info*=0, the execution is successful.  
 If *info* = -*i*, the *i*th parameter had an illegal value.  
 If *info* = *i*, and *i* ≤ *n*, the leading minor of order *i* (and  
 hence the matrix *A* itself) is not positive definite, so the  
 factorization could not be completed, and the solution  
 and error bounds could not be computed; *rcond* = 0 is  
 returned.  
 If *info* = *i*, and *i* = *n* +1, then *U* is nonsingular, but  
*rcond* is less than machine precision, meaning that the  
 matrix is singular to working precision. Nevertheless,  
 the solution and error bounds are computed because  
 there are a number of situations where the computed  
 solution can be more accurate than the value of *rcond*  
 would suggest.

---

## ?ptsv

*Computes the solution to the system of  
 linear equations with a symmetric or  
 Hermitian positive definite tridiagonal  
 matrix A and multiple right-hand sides.*

---

```
call sptsv (n, nrhs, d, e, b, ldb, info)
call dptsv (n, nrhs, d, e, b, ldb, info)
call cptsv (n, nrhs, d, e, b, ldb, info)
call zptsv (n, nrhs, d, e, b, ldb, info)
```

### Discussion

This routine solves for *X* the real or complex system of linear equations  $AX = B$ , where *A* is an *n*-by-*n* symmetric/Hermitian positive definite tridiagonal matrix, the columns of matrix *B* are individual right-hand sides, and the columns of *X* are the corresponding solutions.

*A* is factored as  $A = L D L^H$ , and the factored form of *A* is then used to solve the system of equations  $AX = B$ .

## Input Parameters

<i>n</i>	<b>INTEGER.</b> The order of matrix <i>A</i> ( <i>n</i> ≥ 0).
<i>nrhs</i>	<b>INTEGER.</b> The number of right-hand sides; the number of columns in <i>B</i> ( <i>nrhs</i> ≥ 0).
<i>d</i>	<b>REAL</b> for single precision flavors. <b>DOUBLE PRECISION</b> for double precision flavors. Array, dimension at least $\max(1, n)$ . Contains the diagonal elements of the tridiagonal matrix <i>A</i> .
<i>e, b</i>	<b>REAL</b> for <i>sptsv</i> <b>DOUBLE PRECISION</b> for <i>dptsv</i> <b>COMPLEX</b> for <i>cptsv</i> <b>DOUBLE COMPLEX</b> for <i>zptsv</i> . Arrays: <i>e</i> ( <i>n</i> - 1), <i>b</i> ( <i>ldb</i> , *). The array <i>e</i> contains the ( <i>n</i> - 1) subdiagonal elements of <i>A</i> . The array <i>b</i> contains the matrix <i>B</i> whose columns are the right-hand sides for the systems of equations. The second dimension of <i>b</i> must be at least $\max(1, nrhs)$ .
<i>ldb</i>	<b>INTEGER.</b> The first dimension of <i>b</i> ; <i>ldb</i> ≥ $\max(1, n)$ .

## Output Parameters

<i>d</i>	Overwritten by the <i>n</i> diagonal elements of the diagonal matrix <i>D</i> from the $LDL^H$ factorization of <i>A</i> .
<i>e</i>	Overwritten by the ( <i>n</i> - 1) subdiagonal elements of the unit bidiagonal factor <i>L</i> from the factorization of <i>A</i> .
<i>b</i>	Overwritten by the solution matrix <i>X</i> .
<i>info</i>	<b>INTEGER.</b> If <i>info</i> =0, the execution is successful. If <i>info</i> = <i>-i</i> , the <i>i</i> th parameter had an illegal value. If <i>info</i> = <i>i</i> , the leading minor of order <i>i</i> (and hence the matrix <i>A</i> itself) is not positive definite, and the solution has not been computed. The factorization has not been completed unless <i>i</i> = <i>n</i> .

## ?ptsvx

Uses the factorization  $A=LDL^H$  to compute the solution to the system of linear equations with a symmetric (Hermitian) positive definite tridiagonal matrix A, and provides error bounds on the solution.

```
call sptsvx (fact, n, nrhs, d, e, df, ef, b, ldb, x, idx, rcond,
             ferr, berr, work, info)
call dptsvx (fact, n, nrhs, d, e, df, ef, b, ldb, x, idx, rcond,
             ferr, berr, work, info)
call cptsvx (fact, n, nrhs, d, e, df, ef, b, ldb, x, idx, rcond,
             ferr, berr, work, rwork, info)
call zptsvx (fact, n, nrhs, d, e, df, ef, b, ldb, x, idx, rcond,
             ferr, berr, work, rwork, info)
```

### Discussion

This routine uses the Cholesky factorization  $A=L D L^H$  to compute the solution to a real or complex system of linear equations  $AX=B$ , where A is a *n*-by-*n* symmetric or Hermitian positive definite tridiagonal matrix, the columns of matrix B are individual right-hand sides, and the columns of X are the corresponding solutions.

Error bounds on the solution and a condition estimate are also provided.

The routine ?ptsvx performs the following steps:

1. If *fact* = 'N', the matrix A is factored as  $A = L D L^H$ , where L is a unit lower bidiagonal matrix and D is diagonal. The factorization can also be regarded as having the form  $A = U^H D U$ .
2. If the leading *i*-by-*i* principal minor is not positive definite, then the routine returns with *info* = *i*. Otherwise, the factored form of A is used to estimate the condition number of the matrix A. If the reciprocal of the condition number is less than machine precision, *info* = *n* + 1 is returned as a warning, but the routine still goes on to solve for X and compute error bounds as described below.

3. The system of equations is solved for  $X$  using the factored form of  $A$ .
4. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.

### Input Parameters

<i>fact</i>	CHARACTER*1. Must be ' <b>F</b> ' or ' <b>N</b> '.
	Specifies whether or not the factored form of the matrix $A$ is supplied on entry.
	If <i>fact</i> = ' <b>Fdf and <i>ef</i> contain the factored form of <math>A</math>. Arrays <i>d</i>, <i>e</i>, <i>df</i>, and <i>ef</i> will not be modified.</b>
	If <i>fact</i> = ' <b>N</b> ', the matrix $A$ will be copied to <i>df</i> and <i>ef</i> and factored.
<i>n</i>	INTEGER. The order of matrix $A$ ( <i>n</i> $\geq$ 0).
<i>nrhs</i>	INTEGER. The number of right-hand sides; the number of columns in $B$ ( <i>nrhs</i> $\geq$ 0).
<i>d, df, rwork</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors Arrays: <i>d(n)</i> , <i>df(n)</i> , <i>rwork(n)</i> . The array <i>d</i> contains the <i>n</i> diagonal elements of the tridiagonal matrix $A$ . The array <i>df</i> is an input argument if <i>fact</i> = ' <b>F</b> ' and on entry contains the <i>n</i> diagonal elements of the diagonal matrix $D$ from the $L D L^H$ factorization of $A$ . The array <i>rwork</i> is a workspace array used for complex flavors only.
<i>e, ef, b, work</i>	REAL for <i>sptsvx</i> DOUBLE PRECISION for <i>dptsvx</i> COMPLEX for <i>cptsvx</i> DOUBLE COMPLEX for <i>zptsvx</i> . Arrays: <i>e(n - 1)</i> , <i>ef(n - 1)</i> , <i>b(1:nb, :)</i> , <i>work(*)</i> . The array <i>e</i> contains the ( <i>n</i> - 1) subdiagonal elements of the tridiagonal matrix $A$ .

The array `ef` is an input argument if `fact = 'F'` and on entry contains the ( $n - 1$ ) subdiagonal elements of the unit bidiagonal factor  $L$  from the  $L D L^H$  factorization of  $A$ .

The array `b` contains the matrix  $B$  whose columns are the right-hand sides for the systems of equations.

The array `work` is a workspace array. The dimension of `work` must be at least  $2 * n$  for real flavors, and at least  $n$  for complex flavors.

`ldb`

INTEGER. The leading dimension of `b`;  $ldb \geq \max(1, n)$ .

`ldx`

INTEGER. The leading dimension of `x`;  $ldx \geq \max(1, n)$ .

## Output Parameters

`x`

REAL for `sptsvx`

DOUBLE PRECISION for `dptsvx`

COMPLEX for `cptsvx`

DOUBLE COMPLEX for `zptsvx`.

Array, DIMENSION ( $ldx, *$ ).

If `info = 0` or `info = n+1`, the array `x` contains the solution matrix  $X$  to the system of equations. The second dimension of `x` must be at least  $\max(1, nrhs)$ .

`df, ef`

These arrays are output arguments if `fact = 'N'`.

See the description of `df, ef` in *Input Arguments* section.

`rcond`

REAL for single precision flavors.

DOUBLE PRECISION for double precision flavors.

An estimate of the reciprocal condition number of the matrix  $A$  after equilibration (if done). If `rcond` is less than the machine precision (in particular, if `rcond = 0`), the matrix is singular to working precision. This condition is indicated by a return code of `info > 0`.

`ferr, berr`

REAL for single precision flavors.

DOUBLE PRECISION for double precision flavors.

Arrays, DIMENSION at least  $\max(1, nrhs)$ . Contain the component-wise forward and relative backward errors, respectively, for each solution vector.

*info*                    **INTEGER.** If *info*=0, the execution is successful.  
 If *info* = -*i*, the *i*th parameter had an illegal value.  
 If *info* = *i*, and *i* ≤ *n*, the leading minor of order *i* (and hence the matrix *A* itself) is not positive definite, so the factorization could not be completed, and the solution and error bounds could not be computed; *rcond* = 0 is returned.  
 If *info* = *i*, and *i* = *n* +1, then *U* is nonsingular, but *rcond* is less than machine precision, meaning that the matrix is singular to working precision. Nevertheless, the solution and error bounds are computed because there are a number of situations where the computed solution can be more accurate than the value of *rcond* would suggest.

---

## ?sysv

*Computes the solution to the system of linear equations with a real or complex symmetric matrix A and multiple right-hand sides.*

---

```
call ssysv (uplo, n, nrhs, a, lda, ipiv, b, ldb, work, lwork, info)
call dsysv (uplo, n, nrhs, a, lda, ipiv, b, ldb, work, lwork, info)
call csysv (uplo, n, nrhs, a, lda, ipiv, b, ldb, work, lwork, info)
call zsysv (uplo, n, nrhs, a, lda, ipiv, b, ldb, work, lwork, info)
```

### Discussion

This routine solves for *X* the real or complex system of linear equations  $AX = B$ , where *A* is an *n*-by-*n* symmetric matrix, the columns of matrix *B* are individual right-hand sides, and the columns of *X* are the corresponding solutions.

The diagonal pivoting method is used to factor  $A$  as  $A = U D U^T$  or  $A = L D L^T$ , where  $U$  (or  $L$ ) is a product of permutation and unit upper (lower) triangular matrices, and  $D$  is symmetric and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.

The factored form of  $A$  is then used to solve the system of equations  $AX = B$ .

### Input Parameters

<i>uplo</i>	CHARACTER*1. Must be ' <b>U</b> ' or ' <b>L</b> '.
	Indicates whether the upper or lower triangular part of $A$ is stored and how $A$ is factored:
	If <i>uplo</i> = ' <b>U</b> ', the array <i>a</i> stores the upper triangular part of the matrix $A$ , and $A$ is factored as $UDU^T$ .
	If <i>uplo</i> = ' <b>L</b> ', the array <i>a</i> stores the lower triangular part of the matrix $A$ ; $A$ is factored as $LDL^T$ .
<i>n</i>	INTEGER. The order of matrix $A$ ( <i>n</i> $\geq 0$ ).
<i>nrhs</i>	INTEGER. The number of right-hand sides; the number of columns in $B$ ( <i>nrhs</i> $\geq 0$ ).
<i>a, b, work</i>	REAL for <b>ssysv</b> DOUBLE PRECISION for <b>dsysv</b> COMPLEX for <b>csysv</b> DOUBLE COMPLEX for <b>zsysv</b> . Arrays: <i>a</i> ( <i>lda</i> , *), <i>b</i> ( <i>ldb</i> , *), <i>work</i> ( <i>lwork</i> ). The array <i>a</i> contains either the upper or the lower triangular part of the symmetric matrix $A$ (see <i>uplo</i> ). The second dimension of <i>a</i> must be at least $\max(1, n)$ . The array <i>b</i> contains the matrix $B$ whose columns are the right-hand sides for the systems of equations. The second dimension of <i>b</i> must be at least $\max(1, nrhs)$ . <i>work</i> ( <i>lwork</i> ) is a workspace array.
<i>lda</i>	INTEGER. The first dimension of <i>a</i> ; <i>lda</i> $\geq \max(1, n)$ .
<i>ldb</i>	INTEGER. The first dimension of <i>b</i> ; <i>ldb</i> $\geq \max(1, n)$ .
<i>lwork</i>	INTEGER. The size of the <i>work</i> array ( <i>lwork</i> $\geq 1$ ) See <i>Application notes</i> for the suggested value of <i>lwork</i> .

## Output Parameters

<i>a</i>	If <i>info</i> = 0, <i>a</i> is overwritten by the block-diagonal matrix <i>D</i> and the multipliers used to obtain the factor <i>U</i> (or <i>L</i> ) from the factorization of <i>A</i> as computed by <a href="#">?sytrf</a> .
<i>b</i>	If <i>info</i> = 0, <i>b</i> is overwritten by the solution matrix <i>X</i> .
<i>ipiv</i>	<b>INTEGER.</b> Array, <b>DIMENSION</b> at least $\max(1, n)$ . Contains details of the interchanges and the block structure of <i>D</i> , as determined by <a href="#">?sytrf</a> . If <i>ipiv(i)</i> = <i>k</i> > 0, then $d_{ii}$ is a 1-by-1 diagonal block, and the <i>i</i> th row and column of <i>A</i> was interchanged with the <i>k</i> th row and column. If <i>uplo</i> = 'U' and <i>ipiv(i)</i> = <i>ipiv(i-1)</i> = <i>-m</i> < 0, then <i>D</i> has a 2-by-2 block in rows/columns <i>i</i> and <i>i-1</i> , and ( <i>i-1</i> )th row and column of <i>A</i> was interchanged with the <i>m</i> th row and column. If <i>uplo</i> = 'L' and <i>ipiv(i)</i> = <i>ipiv(i+1)</i> = <i>-m</i> < 0, then <i>D</i> has a 2-by-2 block in rows/columns <i>i</i> and <i>i+1</i> , and ( <i>i+1</i> )th row and column of <i>A</i> was interchanged with the <i>m</i> th row and column.
<i>work(1)</i>	If <i>info</i> =0, on exit <i>work(1)</i> contains the minimum value of <i>lwork</i> required for optimum performance. Use this <i>lwork</i> for subsequent runs.
<i>info</i>	<b>INTEGER.</b> If <i>info</i> =0, the execution is successful. If <i>info</i> = <i>-i</i> , the <i>i</i> th parameter had an illegal value. If <i>info</i> = <i>i</i> , $d_{ii}$ is 0. The factorization has been completed, but <i>D</i> is exactly singular, so the solution could not be computed.

## Application Notes

For better performance, try using *lwork* = *n*\**blocksize*, where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use *lwork* = -1 for the first run. In this case, a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first

---

entry `work(1)` of the `work` array , and no error message related to `lwork` is issued by XERBLA. On exit, examine `work(1)` and use this value for subsequent runs.

---

## ?sysvx

*Uses the diagonal pivoting factorization to compute the solution to the system of linear equations with a real or complex symmetric matrix A, and provides error bounds on the solution.*

---

```
call ssysvx (fact, uplo, n, nrhs, a, lda, af, ldaf, ipiv, b, ldb,
             x, idx, rcond, ferr, berr, work, lwork, iwork, info)
call dsysvx (fact, uplo, n, nrhs, a, lda, af, ldaf, ipiv, b, ldb,
             x, idx, rcond, ferr, berr, work, lwork, iwork, info)
call csysvx (fact, uplo, n, nrhs, a, lda, af, ldaf, ipiv, b, ldb,
             x, idx, rcond, ferr, berr, work, lwork, rwork, info)
call zsysvx (fact, uplo, n, nrhs, a, lda, af, ldaf, ipiv, b, ldb,
             x, idx, rcond, ferr, berr, work, lwork, rwork, info)
```

### Discussion

This routine uses the diagonal pivoting factorization to compute the solution to a real or complex system of linear equations  $AX = B$ , where  $A$  is a  $n$ -by- $n$  symmetric matrix, the columns of matrix  $B$  are individual right-hand sides, and the columns of  $X$  are the corresponding solutions.

Error bounds on the solution and a condition estimate are also provided.

The routine `?sysvx` performs the following steps:

1. If `fact` = '`N`', the diagonal pivoting method is used to factor the matrix  $A$ . The form of the factorization is  $A = U D U^T$  or  $A = L D L^T$ , where  $U$  (or  $L$ ) is a product of permutation and unit upper (lower) triangular matrices, and  $D$  is symmetric and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.

2. If some  $d_{i,i} = 0$ , so that  $D$  is exactly singular, then the routine returns with  $\text{info} = i$ . Otherwise, the factored form of  $A$  is used to estimate the condition number of the matrix  $A$ . If the reciprocal of the condition number is less than machine precision,  $\text{info} = n + 1$  is returned as a warning, but the routine still goes on to solve for  $X$  and compute error bounds as described below.
3. The system of equations is solved for  $X$  using the factored form of  $A$ .
4. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.

### Input Parameters

<i>fact</i>	CHARACTER*1. Must be ' <b>F</b> ' or ' <b>N</b> '.
	Specifies whether or not the factored form of the matrix $A$ has been supplied on entry. If <i>fact</i> = ' <b>Faf and <i>ipiv</i> contain the factored form of <math>A</math>. Arrays <i>a</i>, <i>af</i>, and <i>ipiv</i> will not be modified. If <i>fact</i> = '<b>N</b>', the matrix <math>A</math> will be copied to <i>af</i> and factored.</b>
<i>uplo</i>	CHARACTER*1. Must be ' <b>U</b> ' or ' <b>L</b> '. Indicates whether the upper or lower triangular part of $A$ is stored and how $A$ is factored: If <i>uplo</i> = ' <b>U</b> ', the array <i>a</i> stores the upper triangular part of the symmetric matrix $A$ , and $A$ is factored as $UDU^T$ . If <i>uplo</i> = ' <b>L</b> ', the array <i>a</i> stores the lower triangular part of the symmetric matrix $A$ ; $A$ is factored as $LDL^T$ .
<i>n</i>	INTEGER. The order of matrix $A$ ( $n \geq 0$ ).
<i>nrhs</i>	INTEGER. The number of right-hand sides; the number of columns in $B$ ( $nrhs \geq 0$ ).
<i>a, af, b, work</i>	REAL for <b>ssysvx</b> DOUBLE PRECISION for <b>dsysvx</b> COMPLEX for <b>csysvx</b>

**DOUBLE COMPLEX** for `zsysvx`.

Arrays: `a( lda, * )`, `af( ldaf, * )`, `b( ldb, * )`,  
`work( * )`.

The array `a` contains either the upper or the lower triangular part of the symmetric matrix  $A$  (see `uplo`). The second dimension of `a` must be at least  $\max(1, n)$ .

The array `af` is an input argument if `fact = 'F'`. It contains the block diagonal matrix  $D$  and the multipliers used to obtain the factor  $U$  or  $L$  from the factorization  $A = U D U^T$  or  $A = L D L^T$  as computed by `?sytrf`.

The second dimension of `af` must be at least  $\max(1, n)$ .

The array `b` contains the matrix  $B$  whose columns are the right-hand sides for the systems of equations. The second dimension of `b` must be at least  $\max(1, nrhs)$ .

`work( * )` is a workspace array of dimension (`lwork`).

`lda` **INTEGER.** The first dimension of `a`; `lda`  $\geq \max(1, n)$ .

`ldaf` **INTEGER.** The first dimension of `af`; `ldaf`  $\geq \max(1, n)$ .

`ldb` **INTEGER.** The first dimension of `b`; `ldb`  $\geq \max(1, n)$ .

`ipiv` **INTEGER.**

Array, **DIMENSION** at least  $\max(1, n)$ .

The array `ipiv` is an input argument if `fact = 'F'`.

It contains details of the interchanges and the block structure of  $D$ , as determined by `?sytrf`.

If `ipiv(i) = k > 0`, then  $d_{ii}$  is a 1-by-1 diagonal block, and the  $i$ th row and column of  $A$  was interchanged with the  $k$ th row and column.

If `uplo = 'U'` and `ipiv(i) = ipiv(i-1) = -m < 0`, then  $D$  has a 2-by-2 block in rows/columns  $i$  and  $i-1$ , and  $(i-1)$ th row and column of  $A$  was interchanged with the  $m$ th row and column.

If `uplo = 'L'` and `ipiv(i) = ipiv(i+1) = -m < 0`, then  $D$  has a 2-by-2 block in rows/columns  $i$  and  $i+1$ , and  $(i+1)$ th row and column of  $A$  was interchanged with the  $m$ th row and column.

<i>lidx</i>	INTEGER. The leading dimension of the output array <i>x</i> ; <i>lidx</i> ≥ max(1, <i>n</i> ).
<i>lwork</i>	INTEGER. The size of the <i>work</i> array . See <i>Application notes</i> for the suggested value of <i>lwork</i> .
<i>iwork</i>	INTEGER. Workspace array, DIMENSION at least max(1, <i>n</i> ); used in real flavors only.
<i>rwork</i>	REAL for <i>csysvx</i> ; DOUBLE PRECISION for <i>zsysvx</i> . Workspace array, DIMENSION at least max(1, <i>n</i> ); used in complex flavors only.

### Output Parameters

<i>x</i>	REAL for <i>ssysvx</i> DOUBLE PRECISION for <i>dsysvx</i> COMPLEX for <i>csysvx</i> DOUBLE COMPLEX for <i>zsysvx</i> . Array, DIMENSION ( <i>lidx</i> , *). If <i>info</i> = 0 or <i>info</i> = <i>n</i> +1, the array <i>x</i> contains the solution matrix <i>X</i> to the system of equations. The second dimension of <i>x</i> must be at least max(1, <i>nrhs</i> ). If <i>info</i> < 0, <i>x</i> contains the input matrix <i>A</i> .
<i>af</i> , <i>ipiv</i>	These arrays are output arguments if <i>fact</i> = 'N' . See the description of <i>af</i> , <i>ipiv</i> in <i>Input Arguments</i> section.
<i>rcond</i>	REAL for single precision flavors. DOUBLE PRECISION for double precision flavors. An estimate of the reciprocal condition number of the matrix <i>A</i> . If <i>rcond</i> is less than the machine precision (in particular, if <i>rcond</i> = 0), the matrix is singular to working precision. This condition is indicated by a return code of <i>info</i> > 0.
<i>ferr</i> , <i>berr</i>	REAL for single precision flavors. DOUBLE PRECISION for double precision flavors. Arrays, DIMENSION at least max(1, <i>nrhs</i> ). Contain the component-wise forward and relative backward errors, respectively, for each solution vector.

---

<i>work(1)</i>	If <i>info</i> =0, on exit <i>work(1)</i> contains the minimum value of <i>lwork</i> required for optimum performance. Use this <i>lwork</i> for subsequent runs.
<i>info</i>	<b>INTEGER.</b> If <i>info</i> =0, the execution is successful. If <i>info</i> = <i>-i</i> , the <i>i</i> th parameter had an illegal value. If <i>info</i> = <i>i</i> , and <i>i</i> ≤ <i>n</i> , then <i>d<sub>ii</sub></i> is exactly zero. The factorization has been completed, but the block diagonal matrix <i>D</i> is exactly singular, so the solution and error bounds could not be computed; <i>rcond</i> = 0 is returned. If <i>info</i> = <i>i</i> , and <i>i</i> = <i>n</i> +1, then <i>D</i> is nonsingular, but <i>rcond</i> is less than machine precision, meaning that the matrix is singular to working precision. Nevertheless, the solution and error bounds are computed because there are a number of situations where the computed solution can be more accurate than the value of <i>rcond</i> would suggest.

### Application Notes

For real flavors, *lwork* must be at least  $3*n$ , and for complex flavors at least  $2*n$ . For better performance, try using *lwork* = *n*\**blocksize*, where *blocksize* is the optimal block size for *?sytrf*.

If you are in doubt how much workspace to supply, use *lwork* = -1 for the first run. In this case, a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry *work(1)* of the *work* array, and no error message related to *lwork* is issued by XERBLA. On exit, examine *work(1)* and use this value for subsequent runs.

---

## ?hesvx

*Uses the diagonal pivoting factorization to compute the solution to the complex system of linear equations with a Hermitian matrix A, and provides error bounds on the solution.*

---

```
call chesvx (fact, uplo, n, nrhs, a, lda, af, ldaf, ipiv, b, ldb,
             x, ldx, rcond, ferr, berr, work, lwork, rwork, info)
call zhesvx (fact, uplo, n, nrhs, a, lda, af, ldaf, ipiv, b, ldb,
             x, ldx, rcond, ferr, berr, work, lwork, rwork, info)
```

### Discussion

This routine uses the diagonal pivoting factorization to compute the solution to a complex system of linear equations  $AX = B$ , where  $A$  is a  $n$ -by- $n$  Hermitian matrix, the columns of matrix  $B$  are individual right-hand sides, and the columns of  $X$  are the corresponding solutions.

Error bounds on the solution and a condition estimate are also provided.

The routine `?hesvx` performs the following steps:

1. If `fact` = '`N`', the diagonal pivoting method is used to factor the matrix  $A$ . The form of the factorization is  $A = U D U^H$  or  $A = L D L^H$ , where  $U$  (or  $L$ ) is a product of permutation and unit upper (lower) triangular matrices, and  $D$  is Hermitian and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.
2. If some  $d_{i,i} = 0$ , so that  $D$  is exactly singular, then the routine returns with `info` =  $i$ . Otherwise, the factored form of  $A$  is used to estimate the condition number of the matrix  $A$ . If the reciprocal of the condition number is less than machine precision, `info` =  $n + 1$  is returned as a warning, but the routine still goes on to solve for  $X$  and compute error bounds as described below.
3. The system of equations is solved for  $X$  using the factored form of  $A$ .
4. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.

## Input Parameters

<i>fact</i>	CHARACTER*1. Must be 'F' or 'N'. Specifies whether or not the factored form of the matrix A has been supplied on entry. If <i>fact</i> = 'F': on entry, <i>af</i> and <i>ipiv</i> contain the factored form of A. Arrays <i>a</i> , <i>af</i> , and <i>ipiv</i> will not be modified. If <i>fact</i> = 'N', the matrix A will be copied to <i>af</i> and factored.
<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. Indicates whether the upper or lower triangular part of A is stored and how A is factored: If <i>uplo</i> = 'U', the array <i>a</i> stores the upper triangular part of the Hermitian matrix A, and A is factored as $UDU^H$ . If <i>uplo</i> = 'L', the array <i>a</i> stores the lower triangular part of the Hermitian matrix A; A is factored as $LDL^H$ .
<i>n</i>	INTEGER. The order of matrix A ( <i>n</i> $\geq$ 0).
<i>nrhs</i>	INTEGER. The number of right-hand sides; the number of columns in B ( <i>nrhs</i> $\geq$ 0).
<i>a, af, b, work</i>	COMPLEX for <i>chesvx</i> DOUBLE COMPLEX for <i>zhesvx</i> . Arrays: <i>a</i> ( <i>lda</i> ,*), <i>af</i> ( <i>ldaf</i> ,*), <i>b</i> ( <i>ldb</i> ,*), <i>work</i> (*). The array <i>a</i> contains either the upper or the lower triangular part of the Hermitian matrix A (see <i>uplo</i> ). The second dimension of <i>a</i> must be at least $\max(1, n)$ . The array <i>af</i> is an input argument if <i>fact</i> = 'F'. It contains the block diagonal matrix D and the multipliers used to obtain the factor U or L from the factorization $A = U D U^H$ or $A = L D L^H$ as computed by <a href="#">?hetrf</a> . The second dimension of <i>af</i> must be at least $\max(1, n)$ .

The array  $b$  contains the matrix  $B$  whose columns are the right-hand sides for the systems of equations. The second dimension of  $b$  must be at least  $\max(1, nrhs)$ .

$work(*)$  is a workspace array of dimension ( $lwork$ ).

$lda$  INTEGER. The first dimension of  $a$ ;  $lda \geq \max(1, n)$ .

$ldaf$  INTEGER. The first dimension of  $af$ ;  $ldaf \geq \max(1, n)$ .

$ldb$  INTEGER. The first dimension of  $b$ ;  $ldb \geq \max(1, n)$ .

$ipiv$  INTEGER.

Array, DIMENSION at least  $\max(1, n)$ .

The array  $ipiv$  is an input argument if  $fact = 'F'$ .

It contains details of the interchanges and the block structure of  $D$ , as determined by [?hetrf](#).

If  $ipiv(i) = k > 0$ , then  $d_{ii}$  is a 1-by-1 diagonal block, and the  $i$ th row and column of  $A$  was interchanged with the  $k$ th row and column.

If  $uplo = 'U'$  and  $ipiv(i) = ipiv(i-1) = -m < 0$ , then  $D$  has a 2-by-2 block in rows/columns  $i$  and  $i-1$ , and  $(i-1)$ th row and column of  $A$  was interchanged with the  $m$ th row and column.

If  $uplo = 'L'$  and  $ipiv(i) = ipiv(i+1) = -m < 0$ , then  $D$  has a 2-by-2 block in rows/columns  $i$  and  $i+1$ , and  $(i+1)$ th row and column of  $A$  was interchanged with the  $m$ th row and column.

$lidx$  INTEGER. The leading dimension of the output array  $x$ ;  $lidx \geq \max(1, n)$ .

$lwork$  INTEGER. The size of the  $work$  array.

See *Application notes* for the suggested value of  $lwork$ .

$rwork$  REAL for [chesvx](#);

DOUBLE PRECISION for [zhesvx](#).

Workspace array, DIMENSION at least  $\max(1, n)$ .

## Output Parameters

$x$  COMPLEX for [chesvx](#)

DOUBLE COMPLEX for [zhesvx](#).

Array, DIMENSION ( $lidx, *$ ).

If  $\text{info} = 0$  or  $\text{info} = n+1$ , the array  $x$  contains the solution matrix  $X$  to the system of equations. The second dimension of  $x$  must be at least  $\max(1, \text{nrhs})$ .

$\text{af}, \text{ipiv}$  These arrays are output arguments if  $\text{fact} = 'N'$ . See the description of  $\text{af}, \text{ipiv}$  in *Input Arguments* section.

$rcond$  **REAL** for `chesvx`;  
**DOUBLE PRECISION** for `zhesvx`.  
An estimate of the reciprocal condition number of the matrix  $A$ . If  $rcond$  is less than the machine precision (in particular, if  $rcond = 0$ ), the matrix is singular to working precision. This condition is indicated by a return code of  $\text{info} > 0$ .

$ferr, berr$  **REAL** for `chesvx`;  
**DOUBLE PRECISION** for `zhesvx`.  
Arrays, **DIMENSION** at least  $\max(1, \text{nrhs})$ . Contain the component-wise forward and relative backward errors, respectively, for each solution vector.

$\text{work}(1)$  If  $\text{info}=0$ , on exit  $\text{work}(1)$  contains the minimum value of  $\text{lwork}$  required for optimum performance. Use this  $\text{lwork}$  for subsequent runs.

$\text{info}$  **INTEGER**. If  $\text{info}=0$ , the execution is successful.  
If  $\text{info} = -i$ , the  $i$ th parameter had an illegal value.  
If  $\text{info} = i$ , and  $i \leq n$ , then  $d_{ii}$  is exactly zero. The factorization has been completed, but the block diagonal matrix  $D$  is exactly singular, so the solution and error bounds could not be computed;  $rcond = 0$  is returned.  
If  $\text{info} = i$ , and  $i = n + 1$ , then  $D$  is nonsingular, but  $rcond$  is less than machine precision, meaning that the matrix is singular to working precision. Nevertheless, the solution and error bounds are computed because there are a number of situations where the computed solution can be more accurate than the value of  $rcond$  would suggest.

## Application Notes

The value of `lwork` must be at least  $2*n$ . For better performance, try using  $lwork = n * blocksize$ , where `blocksize` is the optimal block size for `?hetrf`.

If you are in doubt how much workspace to supply, use `lwork = -1` for the first run. In this case, a workspace query is assumed; the routine only calculates the optimal size of the `work` array, returns this value as the first entry `work(1)` of the `work` array, and no error message related to `lwork` is issued by XERBLA. On exit, examine `work(1)` and use this value for subsequent runs.

---

## ?hesv

*Computes the solution to the system of linear equations with a Hermitian matrix A and multiple right-hand sides.*

---

```
call chesv (uplo, n, nrhs, a, lda, ipiv, b, ldb, work, lwork, info)
call zhesv (uplo, n, nrhs, a, lda, ipiv, b, ldb, work, lwork, info)
```

### Discussion

This routine solves for  $X$  the real or complex system of linear equations  $AX = B$ , where  $A$  is an  $n$ -by- $n$  symmetric matrix, the columns of matrix  $B$  are individual right-hand sides, and the columns of  $X$  are the corresponding solutions.

The diagonal pivoting method is used to factor  $A$  as  $A = U D U^H$  or  $A = L D L^H$ , where  $U$  (or  $L$ ) is a product of permutation and unit upper (lower) triangular matrices, and  $D$  is Hermitian and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.

The factored form of  $A$  is then used to solve the system of equations  $AX = B$ .

### Input Parameters

<code>uplo</code>	CHARACTER*1. Must be ' <code>U</code> ' or ' <code>L</code> '.
-------------------	--

	Indicates whether the upper or lower triangular part of $A$ is stored and how $A$ is factored: If $\text{uplo} = \text{'U'}$ , the array $a$ stores the upper triangular part of the matrix $A$ , and $A$ is factored as $UDU^H$ . If $\text{uplo} = \text{'L'}$ , the array $a$ stores the lower triangular part of the matrix $A$ ; $A$ is factored as $LDL^H$ .
$n$	<b>INTEGER.</b> The order of matrix $A$ ( $n \geq 0$ ).
$nrhs$	<b>INTEGER.</b> The number of right-hand sides; the number of columns in $B$ ( $nrhs \geq 0$ ).
$a, b, work$	<b>COMPLEX</b> for <code>chesv</code> <b>DOUBLE COMPLEX</b> for <code>zhesv</code> . Arrays: $a(\text{lda}, *)$ , $b(\text{ldb}, *)$ , $\text{work}(\text{lwork})$ . The array $a$ contains either the upper or the lower triangular part of the Hermitian matrix $A$ (see <code>uplo</code> ). The second dimension of $a$ must be at least $\max(1, n)$ . The array $b$ contains the matrix $B$ whose columns are the right-hand sides for the systems of equations. The second dimension of $b$ must be at least $\max(1, nrhs)$ . $\text{work}(\text{lwork})$ is a workspace array.
$\text{lda}$	<b>INTEGER.</b> The first dimension of $a$ ; $\text{lda} \geq \max(1, n)$ .
$\text{ldb}$	<b>INTEGER.</b> The first dimension of $b$ ; $\text{ldb} \geq \max(1, n)$ .
$\text{lwork}$	<b>INTEGER.</b> The size of the <code>work</code> array ( $\text{lwork} \geq 1$ ) See <i>Application notes</i> for the suggested value of <code>lwork</code> .

## Output Parameters

$a$	If $\text{info} = 0$ , $a$ is overwritten by the block-diagonal matrix $D$ and the multipliers used to obtain the factor $U$ (or $L$ ) from the factorization of $A$ as computed by <a href="#">?hetrf</a> .
$b$	If $\text{info} = 0$ , $b$ is overwritten by the solution matrix $X$ .
$ipiv$	<b>INTEGER.</b> Array, <b>DIMENSION</b> at least $\max(1, n)$ . Contains details of the interchanges and the block structure of $D$ , as determined by <a href="#">?hetrf</a> .

If  $\text{ipiv}(i) = k > 0$ , then  $d_{ii}$  is a 1-by-1 diagonal block, and the  $i$ th row and column of  $A$  was interchanged with the  $k$ th row and column.

If  $\text{uplo} = \text{'U'}$  and  $\text{ipiv}(i) = \text{ipiv}(i-1) = -m < 0$ , then  $D$  has a 2-by-2 block in rows/columns  $i$  and  $i-1$ , and  $(i-1)$ th row and column of  $A$  was interchanged with the  $m$ th row and column.

If  $\text{uplo} = \text{'L'}$  and  $\text{ipiv}(i) = \text{ipiv}(i+1) = -m < 0$ , then  $D$  has a 2-by-2 block in rows/columns  $i$  and  $i+1$ , and  $(i+1)$ th row and column of  $A$  was interchanged with the  $m$ th row and column.

*work(1)*

If  $\text{info}=0$ , on exit *work(1)* contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.

*info*

*INTEGER*. If  $\text{info}=0$ , the execution is successful.

If  $\text{info} = -i$ , the  $i$ th parameter had an illegal value.

If  $\text{info} = i$ ,  $d_{ii}$  is 0. The factorization has been completed, but  $D$  is exactly singular, so the solution could not be computed.

## Application Notes

For better performance, try using  $\text{lwork} = n * \text{blocksize}$ , where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use  $\text{lwork} = -1$  for the first run. In this case, a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry *work(1)* of the *work* array, and no error message related to *lwork* is issued by XERBLA. On exit, examine *work(1)* and use this value for subsequent runs.

## ?spsv

*Computes the solution to the system of linear equations with a real or complex symmetric matrix A stored in packed format, and multiple right-hand sides.*

```
call sspsv (uplo, n, nrhs, ap, ipiv, b, ldb, info)
call dsspv (uplo, n, nrhs, ap, ipiv, b, ldb, info)
call cspsv (uplo, n, nrhs, ap, ipiv, b, ldb, info)
call zspsv (uplo, n, nrhs, ap, ipiv, b, ldb, info)
```

### Discussion

This routine solves for  $X$  the real or complex system of linear equations  $AX = B$ , where  $A$  is an  $n$ -by- $n$  symmetric matrix stored in packed format, the columns of matrix  $B$  are individual right-hand sides, and the columns of  $X$  are the corresponding solutions.

The diagonal pivoting method is used to factor  $A$  as  $A = U D U^T$  or  $A = L D L^T$ , where  $U$  (or  $L$ ) is a product of permutation and unit upper (lower) triangular matrices, and  $D$  is symmetric and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.

The factored form of  $A$  is then used to solve the system of equations  $AX = B$ .

### Input Parameters

<i>uplo</i>	CHARACTER*1. Must be ' <b>U</b> ' or ' <b>L</b> '.
	Indicates whether the upper or lower triangular part of $A$ is stored and how $A$ is factored:
	If <i>uplo</i> = ' <b>U</b> ', the array <i>ap</i> stores the upper triangular part of the matrix $A$ , and $A$ is factored as $UDU^T$ .
	If <i>uplo</i> = ' <b>L</b> ', the array <i>ap</i> stores the lower triangular part of the matrix $A$ ; $A$ is factored as $LDL^T$ .
<i>n</i>	INTEGER. The order of matrix $A$ ( $n \geq 0$ ).
<i>nrhs</i>	INTEGER. The number of right-hand sides; the number of columns in $B$ ( $nrhs \geq 0$ ).

<i>ap</i> , <i>b</i>	<small>REAL for <b>sspsv</b> DOUBLE PRECISION for <b>dspsv</b> COMPLEX for <b>cspsv</b> DOUBLE COMPLEX for <b>zspsv</b>.</small> Arrays: <i>ap</i> ( * ), <i>b</i> ( <i>ldb</i> , * ) The dimension of <i>ap</i> must be at least max(1, <i>n</i> ( <i>n</i> +1)/2). The array <i>ap</i> contains the factor <i>U</i> or <i>L</i> , as specified by <i>uplo</i> , in <i>packed storage</i> (see <a href="#">Matrix Storage Schemes</a> ). The array <i>b</i> contains the matrix <i>B</i> whose columns are the right-hand sides for the systems of equations. The second dimension of <i>b</i> must be at least max(1, <i>nrhs</i> ). <i>ldb</i> <small>INTEGER. The first dimension of <i>b</i>; <i>ldb</i> ≥ max(1, <i>n</i></small>
----------------------	---

## Output Parameters

<i>ap</i>	The block-diagonal matrix <i>D</i> and the multipliers used to obtain the factor <i>U</i> (or <i>L</i> ) from the factorization of <i>A</i> as computed by <a href="#">?sptrf</a> , stored as a packed triangular matrix in the same storage format as <i>A</i> .
<i>b</i>	If <i>info</i> = 0, <i>b</i> is overwritten by the solution matrix <i>X</i> .
<i>ipiv</i>	<small>INTEGER.</small> Array, <small>DIMENSION</small> at least max(1, <i>n</i> ). Contains details of the interchanges and the block structure of <i>D</i> , as determined by <a href="#">?sptrf</a> . If <i>ipiv</i> ( <i>i</i> ) = <i>k</i> > 0, then <i>d</i> <sub><i>ii</i></sub> is a 1-by-1 block, and the <i>i</i> th row and column of <i>A</i> was interchanged with the <i>k</i> th row and column. If <i>uplo</i> = 'U' and <i>ipiv</i> ( <i>i</i> ) = <i>ipiv</i> ( <i>i</i> -1) = - <i>m</i> < 0, then <i>D</i> has a 2-by-2 block in rows/columns <i>i</i> and <i>i</i> -1, and ( <i>i</i> -1)th row and column of <i>A</i> was interchanged with the <i>m</i> th row and column. If <i>uplo</i> = 'L' and <i>ipiv</i> ( <i>i</i> ) = <i>ipiv</i> ( <i>i</i> +1) = - <i>m</i> < 0, then <i>D</i> has a 2-by-2 block in rows/columns <i>i</i> and <i>i</i> +1, and ( <i>i</i> +1)th row and column of <i>A</i> was interchanged with the <i>m</i> th row and column.

*info*

**INTEGER.** If *info*=0, the execution is successful.  
If *info* = *-i*, the *i*th parameter had an illegal value.  
If *info* = *i*,  $d_{ii}$  is 0. The factorization has been completed, but  $D$  is exactly singular, so the solution could not be computed.

---

## ?spsvx

*Uses the diagonal pivoting factorization to compute the solution to the system of linear equations with a real or complex symmetric matrix A stored in packed format, and provides error bounds on the solution.*

---

```
call sspsvx (fact, uplo, n, nrhs, ap, afp, ipiv, b, ldb, x, ldx,
             rcond, ferr, berr, work, iwork, info)
call dspsvx (fact, uplo, n, nrhs, ap, afp, ipiv, b, ldb, x, ldx,
             rcond, ferr, berr, work, iwork, info)
call cspsvx (fact, uplo, n, nrhs, ap, afp, ipiv, b, ldb, x, ldx,
             rcond, ferr, berr, work, rwork, info)
call zspsvx (fact, uplo, n, nrhs, ap, afp, ipiv, b, ldb, x, ldx,
             rcond, ferr, berr, work, rwork, info)
```

### Discussion

This routine uses the diagonal pivoting factorization to compute the solution to a real or complex system of linear equations  $AX = B$ , where  $A$  is a  $n$ -by- $n$  symmetric matrix stored in packed format, the columns of matrix  $B$  are individual right-hand sides, and the columns of  $X$  are the corresponding solutions.

Error bounds on the solution and a condition estimate are also provided.

The routine `?spsvx` performs the following steps:

1. If `fact` = '`N`', the diagonal pivoting method is used to factor the matrix  $A$ . The form of the factorization is  $A = U D U^T$  or  $A = L D L^T$ , where  $U$  (or  $L$ ) is a product of permutation and unit upper (lower) triangular matrices, and  $D$  is symmetric and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.
2. If some  $d_{i,i} = 0$ , so that  $D$  is exactly singular, then the routine returns with `info` = `i`. Otherwise, the factored form of  $A$  is used to estimate the condition number of the matrix  $A$ . If the reciprocal of the condition number

is less than machine precision, `info = n + 1` is returned as a warning, but the routine still goes on to solve for  $X$  and compute error bounds as described below.

3. The system of equations is solved for  $X$  using the factored form of  $A$ .
4. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.

### Input Parameters

<code>fact</code>	<code>CHARACTER*1</code> . Must be ' <code>F</code> ' or ' <code>N</code> '.
	Specifies whether or not the factored form of the matrix $A$ has been supplied on entry. If <code>fact = 'F'</code> : on entry, <code>afp</code> and <code>ipiv</code> contain the factored form of $A$ . Arrays <code>ap</code> , <code>afp</code> , and <code>ipiv</code> will not be modified. If <code>fact = 'N'</code> , the matrix $A$ will be copied to <code>afp</code> and factored.
<code>uplo</code>	<code>CHARACTER*1</code> . Must be ' <code>U</code> ' or ' <code>L</code> '.
	Indicates whether the upper or lower triangular part of $A$ is stored and how $A$ is factored: If <code>uplo = 'U'</code> , the array <code>ap</code> stores the upper triangular part of the symmetric matrix $A$ , and $A$ is factored as $UDU^T$ . If <code>uplo = 'L'</code> , the array <code>ap</code> stores the lower triangular part of the symmetric matrix $A$ ; $A$ is factored as $LDL^T$ .
<code>n</code>	<code>INTEGER</code> . The order of matrix $A$ ( $n \geq 0$ ).
<code>nrhs</code>	<code>INTEGER</code> . The number of right-hand sides; the number of columns in $B$ ( $nrhs \geq 0$ ).
<code>ap, afp, b, work</code>	<code>REAL</code> for <code>sspsvx</code> <code>DOUBLE PRECISION</code> for <code>dspsvx</code> <code>COMPLEX</code> for <code>cspsvx</code> <code>DOUBLE COMPLEX</code> for <code>zspsvx</code> . Arrays: <code>ap(*)</code> , <code>afp(*)</code> , <code>b(ldb,*)</code> , <code>work(*)</code> .

The array  $\text{ap}$  contains the upper or lower triangle of the symmetric matrix  $A$  in *packed storage* (see [Matrix Storage Schemes](#)).

The array  $\text{afp}$  is an input argument if  $\text{fact} = \text{'F'}$ . It contains the block diagonal matrix  $D$  and the multipliers used to obtain the factor  $U$  or  $L$  from the factorization  $A = U D U^T$  or  $A = L D L^T$  as computed by [?sptrf](#), in the same storage format as  $A$ .

The array  $b$  contains the matrix  $B$  whose columns are the right-hand sides for the systems of equations.

$\text{work} (*)$  is a workspace array.

The dimension of arrays  $\text{ap}$  and  $\text{afp}$  must be at least  $\max(1, n(n+1)/2)$ ; the second dimension of  $b$  must be at least  $\max(1, nrhs)$ ; the dimension of  $\text{work}$  must be at least  $\max(1, 3*n)$  for real flavors and  $\max(1, 2*n)$  for complex flavors.

$ldb$

**INTEGER.** The first dimension of  $b$ ;  $ldb \geq \max(1, n)$ .

$ipiv$

**INTEGER.**

Array, **DIMENSION** at least  $\max(1, n)$ .

The array  $ipiv$  is an input argument if  $\text{fact} = \text{'F'}$ . It contains details of the interchanges and the block structure of  $D$ , as determined by [?sptrf](#).

If  $ipiv(i) = k > 0$ , then  $d_{ii}$  is a 1-by-1 diagonal block, and the  $i$ th row and column of  $A$  was interchanged with the  $k$ th row and column.

If  $uplo = \text{'U'}$  and  $ipiv(i) = ipiv(i-1) = -m < 0$ , then  $D$  has a 2-by-2 block in rows/columns  $i$  and  $i-1$ , and ( $i-1$ )th row and column of  $A$  was interchanged with the  $m$ th row and column.

If  $uplo = \text{'L'}$  and  $ipiv(i) = ipiv(i+1) = -m < 0$ , then  $D$  has a 2-by-2 block in rows/columns  $i$  and  $i+1$ , and ( $i+1$ )th row and column of  $A$  was interchanged with the  $m$ th row and column.

$ldx$

**INTEGER.** The leading dimension of the output array  $x$ ;  $ldx \geq \max(1, n)$ .

---

<i>iwork</i>	<b>INTEGER.</b> Workspace array, <b>DIMENSION</b> at least $\max(1, n)$ ; used in real flavors only.
<i>rwork</i>	<b>REAL</b> for <b>cspsvx</b> ; <b>DOUBLE PRECISION</b> for <b>zspsvx</b> . Workspace array, <b>DIMENSION</b> at least $\max(1, n)$ ; used in complex flavors only.

### Output Parameters

<i>x</i>	<b>REAL</b> for <b>sspsvx</b> <b>DOUBLE PRECISION</b> for <b>dspsvx</b> <b>COMPLEX</b> for <b>cspsvx</b> <b>DOUBLE COMPLEX</b> for <b>zspsvx</b> . Array, <b>DIMENSION</b> ( <i>lidx</i> , *). If <i>info</i> = 0 or <i>info</i> = <i>n</i> +1, the array <i>x</i> contains the solution matrix <i>X</i> to the system of equations. The second dimension of <i>x</i> must be at least $\max(1, nrhs)$ .
<i>afp</i> , <i>ipiv</i>	These arrays are output arguments if <i>fact</i> = 'N'. See the description of <i>afp</i> , <i>ipiv</i> in <i>Input Arguments</i> section.
<i>rcond</i>	<b>REAL</b> for single precision flavors. <b>DOUBLE PRECISION</b> for double precision flavors. An estimate of the reciprocal condition number of the matrix <i>A</i> . If <i>rcond</i> is less than the machine precision (in particular, if <i>rcond</i> = 0), the matrix is singular to working precision. This condition is indicated by a return code of <i>info</i> > 0.
<i>ferr</i> , <i>berr</i>	<b>REAL</b> for single precision flavors. <b>DOUBLE PRECISION</b> for double precision flavors. Arrays, <b>DIMENSION</b> at least $\max(1, nrhs)$ . Contain the component-wise forward and relative backward errors, respectively, for each solution vector.
<i>info</i>	<b>INTEGER</b> . If <i>info</i> =0, the execution is successful. If <i>info</i> = <i>-i</i> , the <i>i</i> th parameter had an illegal value. If <i>info</i> = <i>i</i> , and <i>i</i> ≤ <i>n</i> , then $d_{ii}$ is exactly zero. The factorization has been completed, but the block diagonal

matrix  $D$  is exactly singular, so the solution and error bounds could not be computed;  $rcond = 0$  is returned. If  $info = i$ , and  $i = n + 1$ , then  $D$  is nonsingular, but  $rcond$  is less than machine precision, meaning that the matrix is singular to working precision. Nevertheless, the solution and error bounds are computed because there are a number of situations where the computed solution can be more accurate than the value of  $rcond$  would suggest.

## ?hpsvx

*Uses the diagonal pivoting factorization to compute the solution to the system of linear equations with a Hermitian matrix A stored in packed format, and provides error bounds on the solution.*

```
call chpsvx (fact, uplo, n, nrhs, ap, afp, ipiv, b, ldb, x, ldx,
             rcond, ferr, berr, work, rwork, info)
call zhpsvx (fact, uplo, n, nrhs, ap, afp, ipiv, b, ldb, x, ldx,
             rcond, ferr, berr, work, rwork, info)
```

### Discussion

This routine uses the diagonal pivoting factorization to compute the solution to a complex system of linear equations  $AX = B$ , where  $A$  is a  $n$ -by- $n$  Hermitian matrix stored in packed format, the columns of matrix  $B$  are individual right-hand sides, and the columns of  $X$  are the corresponding solutions.

Error bounds on the solution and a condition estimate are also provided.

The routine `?hpsvx` performs the following steps:

1. If `fact` = '`N`', the diagonal pivoting method is used to factor the matrix  $A$ . The form of the factorization is  $A = U D U^H$  or  $A = L D L^H$ , where  $U$  (or  $L$ ) is a product of permutation and unit upper (lower) triangular matrices, and  $D$  is Hermitian and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.
2. If some  $d_{i,i} = 0$ , so that  $D$  is exactly singular, then the routine returns with `info` =  $i$ . Otherwise, the factored form of  $A$  is used to estimate the condition number of the matrix  $A$ . If the reciprocal of the condition number is less than machine precision, `info` =  $n + 1$  is returned as a warning, but the routine still goes on to solve for  $X$  and compute error bounds as described below.
3. The system of equations is solved for  $X$  using the factored form of  $A$ .
4. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.

## Input Parameters

<i>fact</i>	CHARACTER*1. Must be 'F' or 'N'. Specifies whether or not the factored form of the matrix A has been supplied on entry. If <i>fact</i> = 'F': on entry, <i>afp</i> and <i>ipiv</i> contain the factored form of A. Arrays <i>ap</i> , <i>afp</i> , and <i>ipiv</i> will not be modified. If <i>fact</i> = 'N', the matrix A will be copied to <i>afp</i> and factored.
<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. Indicates whether the upper or lower triangular part of A is stored and how A is factored: If <i>uplo</i> = 'U', the array <i>ap</i> stores the upper triangular part of the Hermitian matrix A, and A is factored as $UDU^H$ . If <i>uplo</i> = 'L', the array <i>ap</i> stores the lower triangular part of the Hermitian matrix A; A is factored as $LDL^H$ .
<i>n</i>	INTEGER. The order of matrix A ( $n \geq 0$ ).
<i>nrhs</i>	INTEGER. The number of right-hand sides; the number of columns in B ( $nrhs \geq 0$ ).
<i>ap</i> , <i>afp</i> , <i>b</i> , <i>work</i>	COMPLEX for <i>chpsvx</i> DOUBLE COMPLEX for <i>zhpsvx</i> . Arrays: <i>ap</i> (*), <i>afp</i> (*), <i>b</i> ( <i>ldb</i> , *), <i>work</i> (*). The array <i>ap</i> contains the upper or lower triangle of the Hermitian matrix A in <i>packed storage</i> (see <a href="#">Matrix Storage Schemes</a> ). The array <i>afp</i> is an input argument if <i>fact</i> = 'F'. It contains the block diagonal matrix D and the multipliers used to obtain the factor U or L from the factorization $A = U D U^H$ or $A = L D L^H$ as computed by <a href="#">?hptrf</a> , in the same storage format as A. The array <i>b</i> contains the matrix B whose columns are the right-hand sides for the systems of equations. <i>work</i> (*) is a workspace array.

The dimension of arrays  $\text{ap}$  and  $\text{afp}$  must be at least  $\max(1, \text{n}(\text{n}+1)/2)$ ; the second dimension of  $\text{b}$  must be at least  $\max(1, \text{nrhs})$ ; the dimension of  $\text{work}$  must be at least  $\max(1, 2*\text{n})$ .

$\text{ldb}$  INTEGER. The first dimension of  $\text{b}$ ;  $\text{ldb} \geq \max(1, \text{n})$ .

$\text{ipiv}$  INTEGER.

Array, DIMENSION at least  $\max(1, \text{n})$ .

The array  $\text{ipiv}$  is an input argument if  $\text{fact} = \text{'F'}$ .

It contains details of the interchanges and the block structure of  $D$ , as determined by [?hptrf](#).

If  $\text{ipiv}(\text{i}) = \text{k} > 0$ , then  $d_{\text{ii}}$  is a 1-by-1 diagonal block, and the  $\text{i}$ th row and column of  $A$  was interchanged with the  $\text{k}$ th row and column.

If  $\text{uplo} = \text{'U'}$  and  $\text{ipiv}(\text{i}) = \text{ipiv}(\text{i}-1) = -\text{m} < 0$ , then  $D$  has a 2-by-2 block in rows/columns  $\text{i}$  and  $\text{i}-1$ , and  $(\text{i}-1)$ th row and column of  $A$  was interchanged with the  $\text{m}$ th row and column.

If  $\text{uplo} = \text{'L'}$  and  $\text{ipiv}(\text{i}) = \text{ipiv}(\text{i}+1) = -\text{m} < 0$ , then  $D$  has a 2-by-2 block in rows/columns  $\text{i}$  and  $\text{i}+1$ , and  $(\text{i}+1)$ th row and column of  $A$  was interchanged with the  $\text{m}$ th row and column.

$\text{ldx}$  INTEGER. The leading dimension of the output array  $\text{x}$ ;  $\text{ldx} \geq \max(1, \text{n})$ .

$\text{rwork}$  REAL for [chpsvx](#);

DOUBLE PRECISION for [zhpsvx](#).

Workspace array, DIMENSION at least  $\max(1, \text{n})$ .

## Output Parameters

$\text{x}$  COMPLEX for [chpsvx](#)

DOUBLE COMPLEX for [zhpsvx](#).

Array, DIMENSION ( $\text{ldx}, *$ ).

If  $\text{info} = 0$  or  $\text{info} = \text{n}+1$ , the array  $\text{x}$  contains the solution matrix  $X$  to the system of equations. The second dimension of  $\text{x}$  must be at least  $\max(1, \text{nrhs})$ .

<i>afp</i> , <i>ipiv</i>	These arrays are output arguments if <i>fact</i> = 'N'. See the description of <i>afp</i> , <i>ipiv</i> in <i>Input Arguments</i> section.
<i>rcond</i>	<b>REAL</b> for <i>chpsvx</i> ; <b>DOUBLE PRECISION</b> for <i>zhpsvx</i> . An estimate of the reciprocal condition number of the matrix <i>A</i> . If <i>rcond</i> is less than the machine precision (in particular, if <i>rcond</i> = 0), the matrix is singular to working precision. This condition is indicated by a return code of <i>info</i> > 0.
<i>ferr</i> , <i>berr</i>	<b>REAL</b> for <i>chpsvx</i> ; <b>DOUBLE PRECISION</b> for <i>zhpsvx</i> . Arrays, <b>DIMENSION</b> at least max(1, <i>nrhs</i> ). Contain the component-wise forward and relative backward errors, respectively, for each solution vector.
<i>info</i>	<b>INTEGER</b> . If <i>info</i> =0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> th parameter had an illegal value. If <i>info</i> = <i>i</i> , and <i>i</i> ≤ <i>n</i> , then <i>d<sub>ii</sub></i> is exactly zero. The factorization has been completed, but the block diagonal matrix <i>D</i> is exactly singular, so the solution and error bounds could not be computed; <i>rcond</i> = 0 is returned. If <i>info</i> = <i>i</i> , and <i>i</i> = <i>n</i> +1, then <i>D</i> is nonsingular, but <i>rcond</i> is less than machine precision, meaning that the matrix is singular to working precision. Nevertheless, the solution and error bounds are computed because there are a number of situations where the computed solution can be more accurate than the value of <i>rcond</i> would suggest.

## ?hpsv

*Computes the solution to the system of linear equations with a Hermitian matrix A stored in packed format, and multiple right-hand sides.*

```
call chpsv (uplo, n, nrhs, ap, ipiv, b, ldb, info)
call zhpsv (uplo, n, nrhs, ap, ipiv, b, ldb, info)
```

### Discussion

This routine solves for  $X$  the system of linear equations  $AX = B$ , where  $A$  is an  $n$ -by- $n$  Hermitian matrix stored in packed format, the columns of matrix  $B$  are individual right-hand sides, and the columns of  $X$  are the corresponding solutions.

The diagonal pivoting method is used to factor  $A$  as  $A = U D U^H$  or  $A = L D L^H$ , where  $U$  (or  $L$ ) is a product of permutation and unit upper (lower) triangular matrices, and  $D$  is Hermitian and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.

The factored form of  $A$  is then used to solve the system of equations  $AX = B$ .

### Input Parameters

<i>uplo</i>	CHARACTER*1. Must be ' <b>U</b> ' or ' <b>L</b> '.
	Indicates whether the upper or lower triangular part of $A$ is stored and how $A$ is factored:
	If <i>uplo</i> = ' <b>U</b> ', the array <i>ap</i> stores the upper triangular part of the matrix $A$ , and $A$ is factored as $UDU^H$ .
	If <i>uplo</i> = ' <b>L</b> ', the array <i>ap</i> stores the lower triangular part of the matrix $A$ ; $A$ is factored as $LDL^H$ .
<i>n</i>	INTEGER. The order of matrix $A$ ( <i>n</i> $\geq 0$ ).
<i>nrhs</i>	INTEGER. The number of right-hand sides; the number of columns in $B$ ( <i>nrhs</i> $\geq 0$ ).

<i>ap</i> , <i>b</i>	<small>COMPLEX for <code>chpsv</code> DOUBLE COMPLEX for <code>zhpsv</code>.</small>
	Arrays: <i>ap</i> (*), <i>b</i> ( <i>ldb</i> , *)
	The dimension of <i>ap</i> must be at least $\max(1, \textcolor{red}{n}(\textcolor{red}{n}+1)/2)$ . The array <i>ap</i> contains the factor <i>U</i> or <i>L</i> , as specified by <i>uplo</i> , in <i>packed storage</i> (see <a href="#">Matrix Storage Schemes</a> ). The array <i>b</i> contains the matrix <i>B</i> whose columns are the right-hand sides for the systems of equations. The second dimension of <i>b</i> must be at least $\max(1, \textcolor{red}{nrhs})$ .
<i>ldb</i>	<small>INTEGER. The first dimension of <i>b</i>; <i>ldb</i> <math>\geq \max(1, \textcolor{red}{n})</math>.</small>

## Output Parameters

<i>ap</i>	The block-diagonal matrix <i>D</i> and the multipliers used to obtain the factor <i>U</i> (or <i>L</i> ) from the factorization of <i>A</i> as computed by <a href="#">?hptrf</a> , stored as a packed triangular matrix in the same storage format as <i>A</i> .
<i>b</i>	If <i>info</i> = 0, <i>b</i> is overwritten by the solution matrix <i>X</i> .
<i>ipiv</i>	<small>INTEGER.</small> Array, <small>DIMENSION</small> at least $\max(1, \textcolor{red}{n})$ . Contains details of the interchanges and the block structure of <i>D</i> , as determined by <a href="#">?hptrf</a> . If <i>ipiv</i> ( <i>i</i> ) = <i>k</i> > 0, then <i>d<sub>ii</sub></i> is a 1-by-1 block, and the <i>i</i> th row and column of <i>A</i> was interchanged with the <i>k</i> th row and column. If <i>uplo</i> = 'U' and <i>ipiv</i> ( <i>i</i> ) = <i>ipiv</i> ( <i>i</i> -1) = - <i>m</i> < 0, then <i>D</i> has a 2-by-2 block in rows/columns <i>i</i> and <i>i</i> -1, and ( <i>i</i> -1)th row and column of <i>A</i> was interchanged with the <i>m</i> th row and column. If <i>uplo</i> = 'L' and <i>ipiv</i> ( <i>i</i> ) = <i>ipiv</i> ( <i>i</i> +1) = - <i>m</i> < 0, then <i>D</i> has a 2-by-2 block in rows/columns <i>i</i> and <i>i</i> +1, and ( <i>i</i> +1)th row and column of <i>A</i> was interchanged with the <i>m</i> th row and column.

*info*

**INTEGER.** If *info*=0, the execution is successful.  
If *info* = *-i*, the *i*th parameter had an illegal value.  
If *info* = *i*,  $d_{ii}$  is 0. The factorization has been completed, but  $D$  is exactly singular, so the solution could not be computed.

# LAPACK Routines: Least Squares and Eigenvalue Problems

5

This chapter describes the Math Kernel Library implementation of routines from the LAPACK package that are used for solving linear least-squares problems, eigenvalue and singular value problems, as well as performing a number of related computational tasks.

Sections in this chapter include descriptions of LAPACK [computational routines](#) and [driver routines](#).

For full reference on LAPACK routines and related information see [\[LUG\]](#).

**Least-Squares Problems.** A typical *least-squares problem* is as follows: given a matrix  $A$  and a vector  $b$ , find the vector  $x$  that minimizes the sum of squares  $\sum_i ((Ax)_i - b_i)^2$  or, equivalently, find the vector  $x$  that minimizes the 2-norm  $\|Ax - b\|_2$ .

In the most usual case,  $A$  is an  $m$  by  $n$  matrix with  $m \geq n$  and  $\text{rank}(A) = n$ . This problem is also referred to as finding the *least-squares solution* to an *overdetermined* system of linear equations (here we have more equations than unknowns). To solve this problem, you can use the *QR* factorization of the matrix  $A$  (see *QR Factorization* on [page 5-6](#)).

If  $m < n$  and  $\text{rank}(A) = m$ , there exist an infinite number of solutions  $x$  which exactly satisfy  $Ax = b$ , and thus minimize the norm  $\|Ax - b\|_2$ . In this case it is often useful to find the unique solution that minimizes  $\|x\|_2$ . This problem is referred to as finding the *minimum-norm solution* to an *underdetermined* system of linear equations (here we have more unknowns than equations). To solve this problem, you can use the *LQ* factorization of the matrix  $A$  (see *LQ Factorization* on [page 5-7](#)).

In the general case you may have a *rank-deficient least-squares problem*, with  $\text{rank}(A) < \min(m, n)$ : find the *minimum-norm least-squares solution* that minimizes both  $\|x\|_2$  and  $\|Ax - b\|_2$ . In this case (or when the rank of A is in doubt) you can use the *QR factorization with pivoting* or *singular value decomposition* (see [page 5-74](#)).

**Eigenvalue Problems** (from German *eigen* “own”) are stated as follows: given a matrix  $A$ , find the *eigenvalues*  $\lambda$  and the corresponding *eigenvectors*  $z$  that satisfy the equation

$$Az = \lambda z \quad (\text{right eigenvectors } z)$$

or the equation

$$z^H A = \lambda z^H \quad (\text{left eigenvectors } z).$$

If  $A$  is a real symmetric or complex Hermitian matrix, the above two equations are equivalent, and the problem is called a *symmetric eigenvalue problem*. Routines for solving this type of problems are described in the section *Symmetric Eigenvalue Problems* (see [page 5-101](#)).

Routines for solving eigenvalue problems with nonsymmetric or non-Hermitian matrices are described in the section *Nonsymmetric Eigenvalue Problems* (see [page 5-174](#)).

The library also includes routines that handle *generalized symmetric-definite eigenvalue problems*: find the eigenvalues  $\lambda$  and the corresponding eigenvectors  $x$  that satisfy one of the following equations:

$$Az = \lambda Bz, \quad ABz = \lambda z, \quad \text{or} \quad BAz = \lambda z$$

where  $A$  is symmetric or Hermitian, and  $B$  is symmetric positive-definite or Hermitian positive-definite. Routines for reducing these problems to standard symmetric eigenvalue problems are described in the section *Generalized Symmetric-Definite Eigenvalue Problems* (see [page 5-157](#)).

\* \* \*

To solve a particular problem, you usually call several computational routines. Sometimes you need to combine the routines of this chapter with other LAPACK routines described in Chapter 4 as well as with BLAS routines (Chapter 2).

For example, to solve a set of least-squares problems minimizing  $\|Ax - b\|_2$  for all columns  $b$  of a given matrix  $B$  (where  $A$  and  $B$  are real matrices), you can call `?geqrf` to form the factorization  $A = QR$ , then call `?ormqr` to compute  $C = Q^H B$ , and finally call the BLAS routine `?trsm` to solve for  $X$  the system of equations  $RX = C$ .

Another way is to call an appropriate driver routine that performs several tasks in one call. For example, to solve the least-squares problem the driver routine `?gels` can be used.

## Routine Naming Conventions

For each routine in this chapter, you can use the LAPACK name.

**LAPACK names** have the structure `xyyzzz`, which is described below.

The initial letter `x` indicates the data type:

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

The second and third letters `yy` indicate the matrix type and storage scheme:

<code>bd</code>	bidiagonal matrix
<code>ge</code>	general matrix
<code>gb</code>	general band matrix
<code>hs</code>	upper Hessenberg matrix
<code>or</code>	(real) orthogonal matrix
<code>op</code>	(real) orthogonal matrix (packed storage)
<code>un</code>	(complex) unitary matrix
<code>up</code>	(complex) unitary matrix (packed storage)
<code>pt</code>	symmetric or Hermitian positive-definite tridiagonal matrix
<code>sy</code>	symmetric matrix
<code>sp</code>	symmetric matrix (packed storage)
<code>sb</code>	(real) symmetric band matrix
<code>st</code>	(real) symmetric tridiagonal matrix
<code>he</code>	Hermitian matrix
<code>hp</code>	Hermitian matrix (packed storage)
<code>hb</code>	(complex) Hermitian band matrix
<code>tr</code>	triangular or quasi-triangular matrix.

The last three letters `zzz` indicate the computation performed, for example:

<code>qrf</code>	form the $QR$ factorization
<code>lqf</code>	form the $LQ$ factorization.

Thus, the routine `sgeqrf` forms the  $QR$  factorization of general real matrices in single precision; the corresponding routine for complex matrices is `cgeqrf`.

## Matrix Storage Schemes

LAPACK routines use the following matrix storage schemes:

- *Full storage*: a matrix  $A$  is stored in a two-dimensional array  $a$ , with the matrix element  $a_{ij}$  stored in the array element  $a(i, j)$ .
- *Packed storage* scheme allows you to store symmetric, Hermitian, or triangular matrices more compactly: the upper or lower triangle of the matrix is packed by columns in a one-dimensional array.
- *Band storage*: an  $m$  by  $n$  band matrix with  $k_1$  sub-diagonals and  $k_u$  super-diagonals is stored compactly in a two-dimensional array  $ab$  with  $k_1+k_u+1$  rows and  $n$  columns. Columns of the matrix are stored in the corresponding columns of the array, and *diagonals* of the matrix are stored in rows of the array.

In Chapters 4 and 5, arrays that hold matrices in packed storage have names ending in  $p$ ; arrays with matrices in band storage have names ending in  $b$ .

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

## Mathematical Notation

In addition to the mathematical notation used in previous chapters, descriptions of routines in this chapter use the following notation:

$\lambda_i$	<i>Eigenvalues</i> of the matrix $A$ (for the definition of eigenvalues, see <i>Eigenvalue Problems</i> on <a href="#">page 5-2</a> ).
$\sigma_i$	<i>Singular values</i> of the matrix $A$ . They are equal to square roots of the eigenvalues of $A^H A$ . (For more information, see <a href="#">Singular Value Decomposition</a> ).
$\ x\ _2$	The <i>2-norm</i> of the vector $x$ : $\ x\ _2 = (\sum_i  x_i ^2)^{1/2} = \ x\ _E$ .
$\ A\ _2$	The <i>2-norm</i> (or <i>spectral norm</i> ) of the matrix $A$ . $\ A\ _2 = \max_i \sigma_i$ , $\ A\ _2^2 = \max_{\ x\ =1} (Ax \cdot Ax)$ .
$\ A\ _E$	The <i>Euclidean norm</i> of the matrix $A$ : $\ A\ _E^2 = \sum_i \sum_j  a_{ij} ^2$ (for vectors, the Euclidean norm and the 2-norm are equal: $\ x\ _E = \ x\ _2$ ).
$\theta(x, y)$	The <i>acute angle between vectors</i> $x$ and $y$ : $\cos \theta(x, y) =  x \cdot y  / (\ x\ _2 \ y\ _2)$ .

## Computational Routines

In the sections that follow, the descriptions of LAPACK computational routines are given. These routines perform distinct computational tasks that can be used for:

[Orthogonal Factorizations](#)  
[Singular Value Decomposition](#)  
[Symmetric Eigenvalue Problems](#)  
[Generalized Symmetric-Definite Eigenvalue Problems](#)  
[Nonsymmetric Eigenvalue Problems](#)  
[Generalized Nonsymmetric Eigenvalue Problems](#)  
[Generalized Singular Value Decomposition](#)

See also the respective [driver routines](#).

### Orthogonal Factorizations

This section describes the LAPACK routines for the  $QR$  ( $RQ$ ) and  $LQ$  ( $QL$ ) factorization of matrices. Routines for the  $RZ$  factorization as well as for generalized  $QR$  and  $RQ$  factorizations are also included.

**QR Factorization.** Assume that  $A$  is an  $m$  by  $n$  matrix to be factored.

If  $m \geq n$ , the  $QR$  factorization is given by

$$A = Q \begin{pmatrix} R \\ 0 \end{pmatrix} = (Q_1, Q_2) \begin{pmatrix} R \\ 0 \end{pmatrix}$$

where  $R$  is an  $n$  by  $n$  upper triangular matrix with real diagonal elements, and  $Q$  is an  $m$  by  $m$  orthogonal (or unitary) matrix.

You can use the  $QR$  factorization for solving the following least-squares problem: minimize  $\|Ax - b\|_2$  where  $A$  is a full-rank  $m$  by  $n$  matrix ( $m \geq n$ ). After factoring the matrix, compute the solution  $x$  by solving  $Rx = (Q_1)^T b$ .

If  $m < n$ , the  $QR$  factorization is given by

$$A = QR = Q(R_1 R_2)$$

where  $R$  is trapezoidal,  $R_1$  is upper triangular and  $R_2$  is rectangular.

The LAPACK routines do not form the matrix  $Q$  explicitly. Instead,  $Q$  is represented as a product of  $\min(m, n)$  elementary reflectors. Routines are provided to work with  $Q$  in this representation.

**LQ Factorization** of an  $m$  by  $n$  matrix  $A$  is as follows. If  $m \leq n$ ,

$$A = (L, 0)Q = (L, 0) \begin{pmatrix} Q_1 \\ Q_2 \end{pmatrix} = LQ_1$$

where  $L$  is an  $m$  by  $m$  lower triangular matrix with real diagonal elements, and  $Q$  is an  $n$  by  $n$  orthogonal (or unitary) matrix.

If  $m > n$ , the  $LQ$  factorization is

$$A = \begin{pmatrix} L_1 \\ L_2 \end{pmatrix}Q$$

where  $L_1$  is an  $n$  by  $n$  lower triangular matrix,  $L_2$  is rectangular, and  $Q$  is an  $n$  by  $n$  orthogonal (or unitary) matrix.

You can use the  $LQ$  factorization to find the minimum-norm solution of an underdetermined system of linear equations  $Ax = b$  where  $A$  is an  $m$  by  $n$  matrix of rank  $m$  ( $m < n$ ). After factoring the matrix, compute the solution vector  $x$  as follows: solve  $Ly = b$  for  $y$ , and then compute  $x = (Q_1)^H y$ .

Table 5-1 lists LAPACK routines that perform orthogonal factorization of matrices.

**Table 5-1 Computational Routines for Orthogonal Factorization**

Matrix type, factorization	Factorize without pivoting	Factorize with pivoting	Generate matrix Q	Apply matrix Q
general matrices, QR factorization	<a href="#">?geqrf</a>	<a href="#">?geqpf</a> <a href="#">?geqp3</a>	<a href="#">?orgqr</a> <a href="#">?ungqr</a>	<a href="#">?ormqr</a> <a href="#">?unmqr</a>
general matrices, RQ factorization	<a href="#">?gerqf</a>		<a href="#">?orgrq</a> <a href="#">?ungrq</a>	<a href="#">?ormrq</a> <a href="#">?unmrq</a>
general matrices, LQ factorization	<a href="#">?gelqf</a>		<a href="#">?orglq</a> <a href="#">?unglq</a>	<a href="#">?ormlq</a> <a href="#">?unmlq</a>
general matrices, QL factorization	<a href="#">?geqlf</a>		<a href="#">?orgql</a> <a href="#">?ungql</a>	<a href="#">?ormql</a> <a href="#">?unmql</a>
trapezoidal matrices, RZ factorization	<a href="#">?tzrzf</a>			<a href="#">?ormrz</a> <a href="#">?unmrz</a>
pair of matrices, generalized QR factorization	<a href="#">?ggqrf</a>			
pair of matrices, generalized RQ factorization	<a href="#">?ggrqf</a>			

---

## ?geqrf

*Computes the QR factorization of a general m by n matrix.*

---

```
call sgeqrf ( m, n, a, lda, tau, work, lwork, info )
call dgeqrf ( m, n, a, lda, tau, work, lwork, info )
call cgeqrf ( m, n, a, lda, tau, work, lwork, info )
call zgeqrf ( m, n, a, lda, tau, work, lwork, info )
```

### Discussion

The routine forms the *QR* factorization of a general  $m$  by  $n$  matrix  $A$  (see *Orthogonal Factorizations* on [page 5-6](#)). No pivoting is performed.

The routine does not form the matrix  $Q$  explicitly. Instead,  $Q$  is represented as a product of  $\min(m, n)$  elementary reflectors. Routines are provided to work with  $Q$  in this representation.

### Input Parameters

$m$	INTEGER. The number of rows in the matrix $A$ ( $m \geq 0$ ).
$n$	INTEGER. The number of columns in $A$ ( $n \geq 0$ ).
$a, \text{ work}$	REAL for <code>sgeqrf</code> DOUBLE PRECISION for <code>dgeqrf</code> COMPLEX for <code>cgeqrf</code> DOUBLE COMPLEX for <code>zgeqrf</code> . Arrays: $a(\text{lda}, *)$ contains the matrix $A$ .
	The second dimension of $a$ must be at least $\max(1, n)$ . $\text{work}(\text{lwork})$ is a workspace array.
$\text{lda}$	INTEGER. The first dimension of $a$ ; at least $\max(1, m)$ .
$\text{lwork}$	INTEGER. The size of the $\text{work}$ array ( $\text{lwork} \geq n$ ) See <a href="#">Application notes</a> for the suggested value of $\text{lwork}$ .

## Output Parameters

*a*

Overwritten by the factorization data as follows:

If  $m \geq n$ , the elements below the diagonal are overwritten by the details of the unitary matrix  $Q$ , and the upper triangle is overwritten by the corresponding elements of the upper triangular matrix  $R$ .

If  $m < n$ , the strictly lower triangular part is overwritten by the details of the unitary matrix  $Q$ , and the remaining elements are overwritten by the corresponding elements of the  $m$  by  $n$  upper trapezoidal matrix  $R$ .

*tau*

REAL for `sgeqrf`  
 DOUBLE PRECISION for `dgeqrf`  
 COMPLEX for `cgeqrf`  
 DOUBLE COMPLEX for `zgeqrf`.

Array, DIMENSION at least max (1, min( $m, n$ )).

Contains additional information on the matrix  $Q$ .

*work(1)*

If  $info = 0$ , on exit *work(1)* contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.

*info*

INTEGER.

If  $info = 0$ , the execution is successful.

If  $info = -i$ , the  $i$ th parameter had an illegal value.

## Application Notes

For better performance, try using  $lwork = n * blocksize$ , where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run. On exit, examine *work(1)* and use this value for subsequent runs.

The computed factorization is the exact factorization of a matrix  $A + E$ , where  $\|E\|_2 = O(\epsilon) \|A\|_2$ .

The approximate number of floating-point operations for real flavors is

$$(4/3)n^3 \quad \text{if } m = n,$$

$$(2/3)n^2(3m-n) \quad \text{if } m > n,$$

$$(2/3)m^2(3n-m) \quad \text{if } m < n.$$

The number of operations for complex flavors is 4 times greater.

To solve a set of least-squares problems minimizing  $\|Ax - b\|_2$  for all columns  $b$  of a given matrix  $B$ , you can call the following:

[?geqrf](#) (this routine) to factorize  $A = QR$ ;

[?ormqr](#) to compute  $C = Q^T B$  (for real matrices);

[?unmqr](#) to compute  $C = Q^H B$  (for complex matrices);

[?trsm](#) (a BLAS routine) to solve  $RX = C$ .

(The columns of the computed  $X$  are the least-squares solution vectors  $x$ .)

To compute the elements of  $Q$  explicitly, call

[?orgqr](#) (for real matrices)

[?ungqr](#) (for complex matrices).

## ?geqpf

Computes the  $QR$  factorization of a general  $m$  by  $n$  matrix with pivoting.

```
call sgeqpf ( m, n, a, lda, jpvt, tau, work, info )
call dgeqpf ( m, n, a, lda, jpvt, tau, work, info )
call cgeqpf ( m, n, a, lda, jpvt, tau, work, rwork, info )
call zgeqpf ( m, n, a, lda, jpvt, tau, work, rwork, info )
```

### Discussion

This routine is deprecated and has been replaced by routine [?geqp3](#).

The routine `?geqpf` forms the  $QR$  factorization of a general  $m$  by  $n$  matrix  $A$  with column pivoting:  $AP = QR$  (see *Orthogonal Factorizations* on [page 5-6](#)). Here  $P$  denotes an  $n$  by  $n$  permutation matrix.

The routine does not form the matrix  $Q$  explicitly. Instead,  $Q$  is represented as a product of  $\min(m, n)$  elementary reflectors. Routines are provided to work with  $Q$  in this representation.

### Input Parameters

$m$                     INTEGER. The number of rows in the matrix  $A$  ( $m \geq 0$ ).

$n$                     INTEGER. The number of columns in  $A$  ( $n \geq 0$ ).

$a$ ,  $work$           REAL for `sgeqpf`  
                          DOUBLE PRECISION for `dgeqpf`  
                          COMPLEX for `cgeqpf`  
                          DOUBLE COMPLEX for `zgeqpf`.

Arrays:

$a$  ( $lda, *$ ) contains the matrix  $A$ .

The second dimension of  $a$  must be at least  $\max(1, n)$ .

$work$  ( $lwork$ ) is a workspace array.

$lda$                 INTEGER. The first dimension of  $a$ ; at least  $\max(1, m)$ .

$lwork$                 INTEGER. The size of the  $work$  array; must be at least  $\max(1, 3*n)$ .

<i>jpvt</i>	<b>INTEGER.</b> Array, <b>DIMENSION</b> at least max(1, <i>n</i> ). On entry, if <i>jpvt(i)</i> > 0, the <i>i</i> th column of <i>A</i> is moved to the beginning of <i>AP</i> before the computation, and fixed in place during the computation. If <i>jpvt(i)</i> = 0, the <i>i</i> th column of <i>A</i> is a free column (that is, it may be interchanged during the computation with any other free column).
<i>rwork</i>	<b>REAL</b> for <b>cgeqpf</b> <b>DOUBLE PRECISION</b> for <b>zgeqpf</b> . A workspace array, <b>DIMENSION</b> at least max(1, 2 * <i>n</i> ).

## Output Parameters

<i>a</i>	Overwritten by the factorization data as follows: If <i>m</i> ≥ <i>n</i> , the elements below the diagonal are overwritten by the details of the unitary (orthogonal) matrix <i>Q</i> , and the upper triangle is overwritten by the corresponding elements of the upper triangular matrix <i>R</i> . If <i>m</i> < <i>n</i> , the strictly lower triangular part is overwritten by the details of the matrix <i>Q</i> , and the remaining elements are overwritten by the corresponding elements of the <i>m</i> by <i>n</i> upper trapezoidal matrix <i>R</i> .
<i>tau</i>	<b>REAL</b> for <b>sgeqpf</b> <b>DOUBLE PRECISION</b> for <b>dgeqpf</b> <b>COMPLEX</b> for <b>cgeqpf</b> <b>DOUBLE COMPLEX</b> for <b>zgeqpf</b> . Array, <b>DIMENSION</b> at least max (1, min( <i>m</i> , <i>n</i> )). Contains additional information on the matrix <i>Q</i> .
<i>jpvt</i>	Overwritten by details of the permutation matrix <i>P</i> in the factorization <i>AP</i> = <i>QR</i> . More precisely, the columns of <i>AP</i> are the columns of <i>A</i> in the following order: <i>jpvt(1)</i> , <i>jpvt(2)</i> , ..., <i>jpvt(n)</i> .
<i>info</i>	<b>INTEGER.</b> If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> th parameter had an illegal value.

## Application Notes

The computed factorization is the exact factorization of a matrix  $A + E$ , where  $\|E\|_2 = O(\epsilon) \|A\|_2$ .

The approximate number of floating-point operations for real flavors is

$$\begin{aligned} (4/3)n^3 &\quad \text{if } m = n, \\ (2/3)n^2(3m-n) &\quad \text{if } m > n, \\ (2/3)m^2(3n-m) &\quad \text{if } m < n. \end{aligned}$$

The number of operations for complex flavors is 4 times greater.

To solve a set of least-squares problems minimizing  $\|Ax - b\|_2$  for all columns  $b$  of a given matrix  $B$ , you can call the following:

- ?geqpf (this routine) to factorize  $AP = QR$ ;
- ?ormqr to compute  $C = Q^T B$  (for real matrices);
- ?unmqr to compute  $C = Q^H B$  (for complex matrices);
- ?trsm (a BLAS routine) to solve  $RX = C$ .

(The columns of the computed  $X$  are the permuted least-squares solution vectors  $x$ ; the output array *jpvt* specifies the permutation order.)

To compute the elements of  $Q$  explicitly, call

- ?orgqr (for real matrices)
- ?ungqr (for complex matrices).

---

## ?geqp3

*Computes the QR factorization of a general  $m$  by  $n$  matrix with column pivoting using Level 3 BLAS.*

---

```
call sgeqp3 ( m, n, a, lda, jpvt, tau, work, lwork, info )
call dgeqp3 ( m, n, a, lda, jpvt, tau, work, lwork, info )
call cgeqp3 ( m, n, a, lda, jpvt, tau, work, lwork, rwork, info )
call zgeqp3 ( m, n, a, lda, jpvt, tau, work, lwork, rwork, info )
```

### Discussion

The routine forms the  $QR$  factorization of a general  $m$  by  $n$  matrix  $A$  with column pivoting:  $AP = QR$  (see *Orthogonal Factorizations* on [page 5-6](#)) using Level 3 BLAS. Here  $P$  denotes an  $n$  by  $n$  permutation matrix. Use this routine instead of [?geqpf](#) for better performance.

The routine does not form the matrix  $Q$  explicitly. Instead,  $Q$  is represented as a product of  $\min(m, n)$  elementary reflectors. Routines are provided to work with  $Q$  in this representation.

### Input Parameters

<i>m</i>	INTEGER. The number of rows in the matrix $A$ ( $m \geq 0$ ).
<i>n</i>	INTEGER. The number of columns in $A$ ( $n \geq 0$ ).
<i>a, work</i>	REAL for <a href="#">sgeqp3</a> DOUBLE PRECISION for <a href="#">dgeqp3</a> COMPLEX for <a href="#">cgeqp3</a> DOUBLE COMPLEX for <a href="#">zgeqp3</a> .
Arrays:	
<i>a</i> ( <i>lda,*</i> )	contains the matrix $A$ .
The second dimension of <i>a</i> must be at least $\max(1, n)$ .	
<i>work</i> ( <i>lwork</i> )	is a workspace array.
<i>lda</i>	INTEGER. The first dimension of <i>a</i> ; at least $\max(1, m)$ .

<i>lwork</i>	<b>INTEGER.</b> The size of the <i>work</i> array; must be at least $\max(1, 3*n+1)$ for real flavors, and at least $\max(1, n+1)$ for complex flavors.
<i>jpvt</i>	<b>INTEGER.</b> Array, <b>DIMENSION</b> at least $\max(1, n)$ . On entry, if $jpvt(i) \neq 0$ , the <i>i</i> th column of <i>A</i> is moved to the beginning of <i>AP</i> before the computation, and fixed in place during the computation. If $jpvt(i) = 0$ , the <i>i</i> th column of <i>A</i> is a free column (that is, it may be interchanged during the computation with any other free column).
<i>rwork</i>	<b>REAL</b> for <i>cgeqp3</i> <b>DOUBLE PRECISION</b> for <i>zgeqp3</i> . A workspace array, <b>DIMENSION</b> at least $\max(1, 2*n)$ . Used in complex flavors only.

## Output Parameters

<i>a</i>	Overwritten by the factorization data as follows: If $m \geq n$ , the elements below the diagonal are overwritten by the details of the unitary (orthogonal) matrix <i>Q</i> , and the upper triangle is overwritten by the corresponding elements of the upper triangular matrix <i>R</i> . If $m < n$ , the strictly lower triangular part is overwritten by the details of the matrix <i>Q</i> , and the remaining elements are overwritten by the corresponding elements of the <i>m</i> by <i>n</i> upper trapezoidal matrix <i>R</i> .
<i>tau</i>	<b>REAL</b> for <i>sgeqp3</i> <b>DOUBLE PRECISION</b> for <i>dgeqp3</i> <b>COMPLEX</b> for <i>cgeqp3</i> <b>DOUBLE COMPLEX</b> for <i>zgeqp3</i> . Array, <b>DIMENSION</b> at least $\max(1, \min(m, n))$ . Contains scalar factors of the elementary reflectors for the matrix <i>Q</i> .

<i>jpvt</i>	Overwritten by details of the permutation matrix $P$ in the factorization $AP = QR$ . More precisely, the columns of $AP$ are the columns of $A$ in the following order: $jpvt(1), jpvt(2), \dots, jpvt(n)$ .
<i>info</i>	<i>INTEGER</i> . If $info = 0$ , the execution is successful. If $info = -i$ , the $i$ th parameter had an illegal value.

## Application Notes

To solve a set of least-squares problems minimizing  $\|Ax - b\|_2$  for all columns  $b$  of a given matrix  $B$ , you can call the following:

?geqp3 (this routine)	to factorize $AP = QR$ ;
?ormqr	to compute $C = Q^T B$ (for real matrices);
?unmqr	to compute $C = Q^H B$ (for complex matrices);
?trsm (a BLAS routine)	to solve $RX = C$ .

(The columns of the computed  $X$  are the permuted least-squares solution vectors  $x$ ; the output array *jpvt* specifies the permutation order.)

To compute the elements of  $Q$  explicitly, call

?orgqr	(for real matrices)
?ungqr	(for complex matrices).

## ?orgqr

Generates the real orthogonal matrix  $Q$  of the QR factorization formed by ?geqrf.

---

```
call sorgqr ( m, n, k, a, lda, tau, work, lwork, info )
call dorgqr ( m, n, k, a, lda, tau, work, lwork, info )
```

### Discussion

The routine generates the whole or part of  $m$  by  $m$  orthogonal matrix  $Q$  of the QR factorization formed by the routines sgeqrf/dgeqrf (see [page 5-8](#)) or sgeqpf/dgeqpf (see [page 5-11](#)). Use this routine after a call to sgeqrf/dgeqrf or sgeqpf/dgeqpf.

Usually  $Q$  is determined from the QR factorization of an  $m$  by  $p$  matrix  $A$  with  $m \geq p$ . To compute the whole matrix  $Q$ , use:

```
call ?orgqr ( m, m, p, a, lda, tau, work, lwork, info )
```

To compute the leading  $p$  columns of  $Q$  (which form an orthonormal basis in the space spanned by the columns of  $A$ ):

```
call ?orgqr ( m, p, p, a, lda, tau, work, lwork, info )
```

To compute the matrix  $Q^k$  of the QR factorization of  $A$ 's leading  $k$  columns:

```
call ?orgqr ( m, m, k, a, lda, tau, work, lwork, info )
```

To compute the leading  $k$  columns of  $Q^k$  (which form an orthonormal basis in the space spanned by  $A$ 's leading  $k$  columns):

```
call ?orgqr ( m, k, k, a, lda, tau, work, lwork, info )
```

### Input Parameters

$m$  INTEGER. The order of the orthogonal matrix  $Q$  ( $m \geq 0$ ).

$n$  INTEGER. The number of columns of  $Q$  to be computed ( $0 \leq n \leq m$ ).

$k$  INTEGER. The number of elementary reflectors whose product defines the matrix  $Q$  ( $0 \leq k \leq n$ ).

*a*, *tau*, *work*    **REAL** for **sorgqr**  
                       **DOUBLE PRECISION** for **dorgqr**

Arrays:  
 $a(1da, *)$  and  $\tauau(*)$  are the arrays returned by  
 $sgeqrf / dgeqrf$  or  $sgeqpf / dgeqpf$ .  
 The second dimension of *a* must be at least  $\max(1, n)$ .  
 The dimension of *tau* must be at least  $\max(1, k)$ .

*work(lwork)* is a workspace array.

*lda*                  **INTEGER**. The first dimension of *a*; at least  $\max(1, m)$ .

*lwork*                **INTEGER**. The size of the *work* array ( $lwork \geq n$ )  
 See *Application notes* for the suggested value of *lwork*.

## Output Parameters

*a*                   Overwritten by *n* leading columns of the *m* by *m* orthogonal matrix *Q*.

*work(1)*           If *info* = 0, on exit *work(1)* contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.

*info*                  **INTEGER**.  
 If *info* = 0, the execution is successful.  
 If *info* = *-i*, the *i*th parameter had an illegal value.

## Application Notes

For better performance, try using  $lwork = n * blocksize$ , where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*. If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run. On exit, examine *work(1)* and use this value for subsequent runs.

The computed *Q* differs from an exactly orthogonal matrix by a matrix *E* such that  $\|E\|_2 = O(\epsilon) \|A\|_2$  where  $\epsilon$  is the machine precision.

The total number of floating-point operations is approximately  $4 * m * n * k - 2 * (m + n) * k^2 + (4/3) * k^3$ .  
 If *n* = *k*, the number is approximately  $(2/3) * n^2 * (3m - n)$ .

The complex counterpart of this routine is [?ungqr](#).

## ?ormqr

*Multiplies a real matrix by the orthogonal matrix Q of the QR factorization formed by ?geqrf or ?geqpf.*

```
call sormqr ( side,trans,m,n,k,a,lda,tau,c,ldc,work,lwork,info )
call dormqr ( side,trans,m,n,k,a,lda,tau,c,ldc,work,lwork,info )
```

### Discussion

The routine multiplies a real matrix  $C$  by  $Q$  or  $Q^T$ , where  $Q$  is the orthogonal matrix  $Q$  of the  $QR$  factorization formed by the routines `sgeqrf/dgeqrf` (see [page 5-8](#)) or `sgeqpf/dgeqpf` (see [page 5-11](#)).

Depending on the parameters `side` and `trans`, the routine can form one of the matrix products  $QC$ ,  $Q^TC$ ,  $CQ$ , or  $CQ^T$  (overwriting the result on  $C$ ).

### Input Parameters

<code>side</code>	CHARACTER*1. Must be either ' <code>L</code> ' or ' <code>R</code> '. If <code>side = 'L'</code> , $Q$ or $Q^T$ is applied to $C$ from the left. If <code>side = 'R'</code> , $Q$ or $Q^T$ is applied to $C$ from the right.
<code>trans</code>	CHARACTER*1. Must be either ' <code>N</code> ' or ' <code>T</code> '. If <code>trans = 'N'</code> , the routine multiplies $C$ by $Q$ . If <code>trans = 'T'</code> , the routine multiplies $C$ by $Q^T$ .
<code>m</code>	INTEGER. The number of rows in the matrix $C$ ( $m \geq 0$ ).
<code>n</code>	INTEGER. The number of columns in $C$ ( $n \geq 0$ ).
<code>k</code>	INTEGER. The number of elementary reflectors whose product defines the matrix $Q$ . Constraints: $0 \leq k \leq m$ if <code>side = 'L'</code> ; $0 \leq k \leq n$ if <code>side = 'R'</code> .
<code>a,work,tau,c</code>	REAL for <code>sgeqrf</code> DOUBLE PRECISION for <code>dgeqrf</code> . Arrays: <code>a(lda,*)</code> and <code>tau(*)</code> are the arrays returned by

`sgeqrf / dgeqrf` or `sgeqpf / dgeqpf`.

The second dimension of `a` must be at least  $\max(1, k)$ .

The dimension of `tau` must be at least  $\max(1, k)$ .

`c(ldc, *)` contains the matrix  $C$ .

The second dimension of `c` must be at least  $\max(1, n)$

`work(lwork)` is a workspace array.

`lda`

`INTEGER`. The first dimension of `a`. Constraints:

`lda`  $\geq \max(1, m)$  if `side = 'L'`;

`lda`  $\geq \max(1, n)$  if `side = 'R'`.

`ldc`

`INTEGER`. The first dimension of `c`. Constraint:

`ldc`  $\geq \max(1, m)$ .

`lwork`

`INTEGER`. The size of the `work` array. Constraints:

`lwork`  $\geq \max(1, n)$  if `side = 'L'`;

`lwork`  $\geq \max(1, m)$  if `side = 'R'`.

See *Application notes* for the suggested value of `lwork`.

## Output Parameters

`c`

Overwritten by the product  $QC$ ,  $Q^TC$ ,  $CQ$ , or  $CQ^T$  (as specified by `side` and `trans`).

`work(1)`

If `info = 0`, on exit `work(1)` contains the minimum value of `lwork` required for optimum performance. Use this `lwork` for subsequent runs.

`info`

`INTEGER`.

If `info = 0`, the execution is successful.

If `info = -i`, the `i`th parameter had an illegal value.

## Application Notes

For better performance, try using `lwork = n*blocksize` (if `side = 'L'`) or `lwork = m*blocksize` (if `side = 'R'`) where `blocksize` is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*. If you are in doubt how much workspace to supply, use a generous value of `lwork` for the first run. On exit, examine `work(1)` and use this value for subsequent runs.

The complex counterpart of this routine is [?unmqr](#).

## ?ungqr

Generates the complex unitary matrix  $Q$  of the  $QR$  factorization formed by ?geqrf.

```
call cungqr ( m, n, k, a, lda, tau, work, lwork, info )
call zungqr ( m, n, k, a, lda, tau, work, lwork, info )
```

### Discussion

The routine generates the whole or part of  $m$  by  $m$  unitary matrix  $Q$  of the  $QR$  factorization formed by the routines `cgeqrf/zgeqrf` (see [page 5-8](#)) or `cgeqpf/zgeqpf` (see [page 5-11](#)). Use this routine after a call to `cgeqrf/zgeqrf` or `cgeqpf/zgeqpf`.

Usually  $Q$  is determined from the  $QR$  factorization of an  $m$  by  $p$  matrix  $A$  with  $m \geq p$ . To compute the whole matrix  $Q$ , use:

```
call ?ungqr ( m, m, p, a, lda, tau, work, lwork, info )
```

To compute the leading  $p$  columns of  $Q$  (which form an orthonormal basis in the space spanned by the columns of  $A$ ):

```
call ?ungqr ( m, p, p, a, lda, tau, work, lwork, info )
```

To compute the matrix  $Q^k$  of the  $QR$  factorization of  $A$ 's leading  $k$  columns:

```
call ?ungqr ( m, m, k, a, lda, tau, work, lwork, info )
```

To compute the leading  $k$  columns of  $Q^k$  (which form an orthonormal basis in the space spanned by  $A$ 's leading  $k$  columns):

```
call ?ungqr ( m, k, k, a, lda, tau, work, lwork, info )
```

### Input Parameters

- $m$             INTEGER. The order of the unitary matrix  $Q$  ( $m \geq 0$ ).
- $n$             INTEGER. The number of columns of  $Q$  to be computed ( $0 \leq n \leq m$ ).
- $k$             INTEGER. The number of elementary reflectors whose product defines the matrix  $Q$  ( $0 \leq k \leq n$ ).

*a*, *tau*, *work*    COMPLEX for `cungqr`  
                   DOUBLE COMPLEX for `zungqr`

Arrays:

*a*(*lda*,\*) and *tau*(\*) are the arrays returned by `cgeqrf/zgeqrf` or `cgeqpf/zgeqpf`.  
     The second dimension of *a* must be at least  $\max(1, n)$ .  
     The dimension of *tau* must be at least  $\max(1, k)$ .

*work*(*lwork*) is a workspace array.

*lda*            INTEGER. The first dimension of *a*; at least  $\max(1, m)$ .

*lwork*            INTEGER. The size of the *work* array ( $lwork \geq n$ )  
     See *Application notes* for the suggested value of *lwork*.

## Output Parameters

*a*            Overwritten by *n* leading columns of the *m* by *m* unitary matrix *Q*.

*work*(1)        If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.

*info*            INTEGER.  
     If *info* = 0, the execution is successful.  
     If *info* = *-i*, the *i*th parameter had an illegal value.

## Application Notes

For better performance, try using  $lwork = n * blocksize$ , where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*. If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run. On exit, examine *work*(1) and use this value for subsequent runs.

The computed *Q* differs from an exactly unitary matrix by a matrix *E* such that  $\|E\|_2 = O(\epsilon) \|A\|_2$  where  $\epsilon$  is the machine precision.

The total number of floating-point operations is approximately  $16 * m * n * k - 8 * (m + n) * k^2 + (16/3) * k^3$ .  
     If *n* = *k*, the number is approximately  $(8/3) * n^2 * (3m - n)$ .

The real counterpart of this routine is [?orgqr](#).

## ?unmqr

*Multiplies a complex matrix by the unitary matrix Q of the QR factorization formed by ?geqrf.*

```
call cunmqr ( side,trans,m,n,k,a,lda,tau,c,ldc,work,lwork,info )
call zunmqr ( side,trans,m,n,k,a,lda,tau,c,ldc,work,lwork,info )
```

### Discussion

The routine multiplies a rectangular complex matrix  $C$  by  $Q$  or  $Q^H$ , where  $Q$  is the unitary matrix  $Q$  of the  $QR$  factorization formed by the routines `cgeqrf/zgeqrf` (see [page 5-8](#)) or `cgeqpf/zgeqpf` (see [page 5-11](#)).

Depending on the parameters `side` and `trans`, the routine can form one of the matrix products  $QC$ ,  $Q^HC$ ,  $CQ$ , or  $CQ^H$  (overwriting the result on  $C$ ).

### Input Parameters

<code>side</code>	CHARACTER*1. Must be either ' <code>L</code> ' or ' <code>R</code> '. If <code>side = 'L'</code> , $Q$ or $Q^H$ is applied to $C$ from the left. If <code>side = 'R'</code> , $Q$ or $Q^H$ is applied to $C$ from the right.
<code>trans</code>	CHARACTER*1. Must be either ' <code>N</code> ' or ' <code>C</code> '. If <code>trans = 'N'</code> , the routine multiplies $C$ by $Q$ . If <code>trans = 'C'</code> , the routine multiplies $C$ by $Q^H$ .
<code>m</code>	INTEGER. The number of rows in the matrix $C$ ( $m \geq 0$ ).
<code>n</code>	INTEGER. The number of columns in $C$ ( $n \geq 0$ ).
<code>k</code>	INTEGER. The number of elementary reflectors whose product defines the matrix $Q$ . Constraints: $0 \leq k \leq m$ if <code>side = 'L'</code> ; $0 \leq k \leq n$ if <code>side = 'R'</code> .
<code>a,work,tau,c</code>	COMPLEX for <code>cgeqrf</code> DOUBLE COMPLEX for <code>zgeqrf</code> . Arrays: <code>a(lda,*)</code> and <code>tau(*)</code> are the arrays returned by

`cgeqrf / zgeqrf` or `cgeqpf / zgeqpf`.

The second dimension of `a` must be at least  $\max(1, k)$ .

The dimension of `tau` must be at least  $\max(1, k)$ .

`c(ldc, *)` contains the matrix  $C$ .

The second dimension of `c` must be at least  $\max(1, n)$

`work(lwork)` is a workspace array.

`lda`

`INTEGER`. The first dimension of `a`. Constraints:

`lda`  $\geq \max(1, m)$  if `side = 'L'`;

`lda`  $\geq \max(1, n)$  if `side = 'R'`.

`ldc`

`INTEGER`. The first dimension of `c`. Constraint:

`ldc`  $\geq \max(1, m)$ .

`lwork`

`INTEGER`. The size of the `work` array. Constraints:

`lwork`  $\geq \max(1, n)$  if `side = 'L'`;

`lwork`  $\geq \max(1, m)$  if `side = 'R'`.

See *Application notes* for the suggested value of `lwork`.

## Output Parameters

`c`

Overwritten by the product  $QC$ ,  $Q^H C$ ,  $CQ$ , or  $CQ^H$  (as specified by `side` and `trans`).

`work(1)`

If `info = 0`, on exit `work(1)` contains the minimum value of `lwork` required for optimum performance. Use this `lwork` for subsequent runs.

`info`

`INTEGER`.

If `info = 0`, the execution is successful.

If `info = -i`, the `i`th parameter had an illegal value.

## Application Notes

For better performance, try using `lwork = n * blocksize` (if `side = 'L'`) or `lwork = m * blocksize` (if `side = 'R'`) where `blocksize` is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*. If you are in doubt how much workspace to supply, use a generous value of `lwork` for the first run. On exit, examine `work(1)` and use this value for subsequent runs.

The real counterpart of this routine is [?ormqr](#).

## ?gelqf

Computes the  $LQ$  factorization of a general  $m$  by  $n$  matrix.

---

```
call sgelqf ( m, n, a, lda, tau, work, lwork, info )
call dgelqf ( m, n, a, lda, tau, work, lwork, info )
call cgelqf ( m, n, a, lda, tau, work, lwork, info )
call zgelqf ( m, n, a, lda, tau, work, lwork, info )
```

### Discussion

The routine forms the  $LQ$  factorization of a general  $m$  by  $n$  matrix  $A$  (see *Orthogonal Factorizations* on [page 5-6](#)). No pivoting is performed.

The routine does not form the matrix  $Q$  explicitly. Instead,  $Q$  is represented as a product of  $\min(m, n)$  elementary reflectors. Routines are provided to work with  $Q$  in this representation.

### Input Parameters

***m***            INTEGER. The number of rows in the matrix  $A$  ( $m \geq 0$ ).

***n***            INTEGER. The number of columns in  $A$  ( $n \geq 0$ ).

***a, work***    REAL for `sgelqf`  
                  DOUBLE PRECISION for `dgelqf`  
                  COMPLEX for `cgelqf`  
                  DOUBLE COMPLEX for `zgelqf`.

Arrays:

***a(lda, \*)*** contains the matrix  $A$ .  
The second dimension of  $a$  must be at least  $\max(1, n)$ .  
***work(lwork)*** is a workspace array.

***lda***          INTEGER. The first dimension of  $a$ ; at least  $\max(1, m)$ .

***lwork***        INTEGER. The size of the  $work$  array; at least  $\max(1, m)$ .  
See *Application notes* for the suggested value of  $lwork$ .

## Output Parameters

*a*

Overwritten by the factorization data as follows:

If  $m \leq n$ , the elements above the diagonal are overwritten by the details of the unitary (orthogonal) matrix  $Q$ , and the lower triangle is overwritten by the corresponding elements of the lower triangular matrix  $L$ .

If  $m > n$ , the strictly upper triangular part is overwritten by the details of the matrix  $Q$ , and the remaining elements are overwritten by the corresponding elements of the  $m$  by  $n$  lower trapezoidal matrix  $L$ .

*tau*

REAL for `sgelqf`

DOUBLE PRECISION for `dgelqf`

COMPLEX for `cgelqf`

DOUBLE COMPLEX for `zgelqf`.

Array, DIMENSION at least  $\max(1, \min(m, n))$ .

Contains additional information on the matrix  $Q$ .

*work(1)*

If  $info = 0$ , on exit `work(1)` contains the minimum value of `lwork` required for optimum performance. Use this `lwork` for subsequent runs.

*info*

INTEGER.

If  $info = 0$ , the execution is successful.

If  $info = -i$ , the  $i$ th parameter had an illegal value.

## Application Notes

For better performance, try using  $lwork = m * blocksize$ , where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use a generous value of `lwork` for the first run. On exit, examine `work(1)` and use this value for subsequent runs.

The computed factorization is the exact factorization of a matrix  $A + E$ , where  $\|E\|_2 = O(\epsilon) \|A\|_2$ .

The approximate number of floating-point operations for real flavors is

$$\begin{aligned} (4/3)\textcolor{red}{n}^3 &\quad \text{if } \textcolor{red}{m} = \textcolor{red}{n}, \\ (2/3)\textcolor{red}{n}^2(3\textcolor{red}{m}-\textcolor{red}{n}) &\quad \text{if } \textcolor{red}{m} > \textcolor{red}{n}, \\ (2/3)\textcolor{red}{m}^2(3\textcolor{red}{n}-\textcolor{red}{m}) &\quad \text{if } \textcolor{red}{m} < \textcolor{red}{n}. \end{aligned}$$

The number of operations for complex flavors is 4 times greater.

To find the minimum-norm solution of an underdetermined least-squares problem minimizing  $\|Ax - b\|_2$  for all columns  $b$  of a given matrix  $B$ , you can call the following:

- ?gelqf (this routine) to factorize  $A = LQ$ ;
- ?trsm (a BLAS routine) to solve  $LY = B$  for  $Y$ ;
- ?ormlq to compute  $X = (Q_1)^T Y$  (for real matrices);
- ?unmlq to compute  $X = (Q_1)^H Y$  (for complex matrices).

(The columns of the computed  $X$  are the minimum-norm solution vectors  $x$ . Here  $A$  is an  $\textcolor{red}{m}$  by  $\textcolor{red}{n}$  matrix with  $\textcolor{red}{m} < \textcolor{red}{n}$ ;  $Q_1$  denotes the first  $\textcolor{red}{m}$  columns of  $Q$ ).

To compute the elements of  $Q$  explicitly, call

- ?orglq (for real matrices)
- ?unglq (for complex matrices).

## ?orglq

*Generates the real orthogonal matrix  $Q$  of the LQ factorization formed by ?gelqf.*

```
call sorglq ( m, n, k, a, lda, tau, work, lwork, info )
call dorglq ( m, n, k, a, lda, tau, work, lwork, info )
```

### Discussion

The routine generates the whole or part of  $n$  by  $n$  orthogonal matrix  $Q$  of the LQ factorization formed by the routines `sgelqf/dgelqf` (see [page 5-25](#)). Use this routine after a call to `sgelqf/dgelqf`.

Usually  $Q$  is determined from the LQ factorization of an  $p$  by  $n$  matrix  $A$  with  $n \geq p$ . To compute the whole matrix  $Q$ , use:

```
call ?orglq ( n, n, p, a, lda, tau, work, lwork, info )
```

To compute the leading  $p$  rows of  $Q$  (which form an orthonormal basis in the space spanned by the rows of  $A$ ):

```
call ?orglq ( p, n, p, a, lda, tau, work, lwork, info )
```

To compute the matrix  $Q^k$  of the LQ factorization of  $A$ 's leading  $k$  rows:

```
call ?orglq ( n, n, k, a, lda, tau, work, lwork, info )
```

To compute the leading  $k$  rows of  $Q^k$  (which form an orthonormal basis in the space spanned by  $A$ 's leading  $k$  rows):

```
call ?orgqr ( k, n, k, a, lda, tau, work, lwork, info )
```

### Input Parameters

$m$             INTEGER. The number of rows of  $Q$  to be computed ( $0 \leq m \leq n$ ).

$n$             INTEGER. The order of the orthogonal matrix  $Q$  ( $n \geq m$ ).

$k$             INTEGER. The number of elementary reflectors whose product defines the matrix  $Q$  ( $0 \leq k \leq m$ ).

`a, tau, work` REAL for `sorglq`  
DOUBLE PRECISION for `dorglq`

Arrays:

`a( lda, * )` and `tau( * )` are the arrays returned by `sgelqf/dgelqf`.

The second dimension of `a` must be at least  $\max(1, n)$ .

The dimension of `tau` must be at least  $\max(1, k)$ .

`work(lwork)` is a workspace array.

`lda` INTEGER. The first dimension of `a`; at least  $\max(1, m)$ .

`lwork` INTEGER. The size of the `work` array; at least  $\max(1, m)$ .  
See *Application notes* for the suggested value of `lwork`.

## Output Parameters

`a` Overwritten by  $m$  leading rows of the  $n$  by  $n$  orthogonal matrix  $Q$ .

`work( 1 )` If `info = 0`, on exit `work( 1 )` contains the minimum value of `lwork` required for optimum performance. Use this `lwork` for subsequent runs.

`info` INTEGER.  
If `info = 0`, the execution is successful.  
If `info = -i`, the  $i$ th parameter had an illegal value.

## Application Notes

For better performance, try using `lwork = m * blocksize`, where `blocksize` is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*. If you are in doubt how much workspace to supply, use a generous value of `lwork` for the first run. On exit, examine `work( 1 )` and use this value for subsequent runs.

The computed  $Q$  differs from an exactly orthogonal matrix by a matrix  $E$  such that  $\|E\|_2 = O(\epsilon) \|A\|_2$  where  $\epsilon$  is the machine precision.

The total number of floating-point operations is approximately  $4 * m * n * k - 2 * (m + n) * k^2 + (4/3) * k^3$ .

If  $m = k$ , the number is approximately  $(2/3) * m^2 * (3n - m)$ .

The complex counterpart of this routine is [?unglq](#).

## ?ormlq

*Multiplies a real matrix by the orthogonal matrix Q of the LQ factorization formed by ?gelqf.*

---

```
call sormlq ( side,trans,m,n,k,a,lda,tau,c,ldc,work,lwork,info )
call dormlq ( side,trans,m,n,k,a,lda,tau,c,ldc,work,lwork,info )
```

### Discussion

The routine multiplies a real  $m$ -by- $n$  matrix  $C$  by  $Q$  or  $Q^T$ , where  $Q$  is the orthogonal matrix  $Q$  of the  $LQ$  factorization formed by the routine [sgelqf/dgelqf](#) (see [page 5-25](#)).

Depending on the parameters `side` and `trans`, the routine can form one of the matrix products  $QC$ ,  $Q^TC$ ,  $CQ^T$ , or  $CQ$  (overwriting the result on  $C$ ).

### Input Parameters

<code>side</code>	CHARACTER*1. Must be either ' <code>L</code> ' or ' <code>R</code> '. If <code>side = 'L'</code> , $Q$ or $Q^T$ is applied to $C$ from the left. If <code>side = 'R'</code> , $Q$ or $Q^T$ is applied to $C$ from the right.
<code>trans</code>	CHARACTER*1. Must be either ' <code>N</code> ' or ' <code>T</code> '. If <code>trans = 'N'</code> , the routine multiplies $C$ by $Q$ . If <code>trans = 'T'</code> , the routine multiplies $C$ by $Q^T$ .
<code>m</code>	INTEGER. The number of rows in the matrix $C$ ( $m \geq 0$ ).
<code>n</code>	INTEGER. The number of columns in $C$ ( $n \geq 0$ ).
<code>k</code>	INTEGER. The number of elementary reflectors whose product defines the matrix $Q$ . Constraints: $0 \leq k \leq m$ if <code>side = 'L'</code> ; $0 \leq k \leq n$ if <code>side = 'R'</code> .
<code>a,work,tau,c</code>	REAL for <code>sormlq</code> DOUBLE PRECISION for <code>dormlq</code> . Arrays: <code>a( lda, * )</code> and <code>tau( * )</code> are arrays returned by <code>?gelqf</code> .

The second dimension of  $a$  must be:

at least  $\max(1, m)$  if  $\text{side} = 'L'$ ;

at least  $\max(1, n)$  if  $\text{side} = 'R'$ .

The dimension of  $\tau_{\alpha}$  must be at least  $\max(1, k)$ .

$c(\text{ldc}, *)$  contains the matrix  $C$ .

The second dimension of  $c$  must be at least  $\max(1, n)$

$\text{work}(\text{lwork})$  is a workspace array.

$\text{lda}$  INTEGER. The first dimension of  $a$ ;  $\text{lda} \geq \max(1, k)$ .

$\text{ldc}$  INTEGER. The first dimension of  $c$ ;  $\text{ldc} \geq \max(1, m)$ .

$\text{lwork}$  INTEGER. The size of the  $\text{work}$  array. Constraints:

$\text{lwork} \geq \max(1, n)$  if  $\text{side} = 'L'$ ;

$\text{lwork} \geq \max(1, m)$  if  $\text{side} = 'R'$ .

See *Application notes* for the suggested value of  $\text{lwork}$ .

## Output Parameters

$c$  Overwritten by the product  $QC$ ,  $Q^T C$ ,  $CQ$ , or  $CQ^T$  (as specified by  $\text{side}$  and  $\text{trans}$ ).

$\text{work}(1)$  If  $\text{info} = 0$ , on exit  $\text{work}(1)$  contains the minimum value of  $\text{lwork}$  required for optimum performance. Use this  $\text{lwork}$  for subsequent runs.

$\text{info}$  INTEGER.  
If  $\text{info} = 0$ , the execution is successful.  
If  $\text{info} = -i$ , the  $i$ th parameter had an illegal value.

## Application Notes

For better performance, try using  $\text{lwork} = n * \text{blocksize}$  (if  $\text{side} = 'L'$ ) or  $\text{lwork} = m * \text{blocksize}$  (if  $\text{side} = 'R'$ ) where  $\text{blocksize}$  is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*. If you are in doubt how much workspace to supply, use a generous value of  $\text{lwork}$  for the first run. On exit, examine  $\text{work}(1)$  and use this value for subsequent runs.

The complex counterpart of this routine is [?unmlq](#).

## ?unglq

*Generates the complex unitary matrix  $Q$  of the  $LQ$  factorization formed by ?gelqf.*

```
call cunglq ( m, n, k, a, lda, tau, work, lwork, info )
call zunqlq ( m, n, k, a, lda, tau, work, lwork, info )
```

### Discussion

The routine generates the whole or part of  $n$  by  $n$  unitary matrix  $Q$  of the  $LQ$  factorization formed by the routines `cgelqf/zgelqf` (see [page 5-25](#)). Use this routine after a call to `cgelqf/zgelqf`.

Usually  $Q$  is determined from the  $LQ$  factorization of an  $p$  by  $n$  matrix  $A$  with  $n \geq p$ . To compute the whole matrix  $Q$ , use:

```
call ?unglq ( n, n, p, a, lda, tau, work, lwork, info )
```

To compute the leading  $p$  rows of  $Q$  (which form an orthonormal basis in the space spanned by the rows of  $A$ ):

```
call ?unglq ( p, n, p, a, lda, tau, work, lwork, info )
```

To compute the matrix  $Q^k$  of the  $LQ$  factorization of  $A$ 's leading  $k$  rows:

```
call ?unglq ( n, n, k, a, lda, tau, work, lwork, info )
```

To compute the leading  $k$  rows of  $Q^k$  (which form an orthonormal basis in the space spanned by  $A$ 's leading  $k$  rows):

```
call ?ungqr ( k, n, k, a, lda, tau, work, lwork, info )
```

### Input Parameters

$m$             INTEGER. The number of rows of  $Q$  to be computed ( $0 \leq m \leq n$ ).

$n$             INTEGER. The order of the unitary matrix  $Q$  ( $n \geq m$ ).

$k$             INTEGER. The number of elementary reflectors whose product defines the matrix  $Q$  ( $0 \leq k \leq m$ ).

---

`a, tau, work` COMPLEX for `cunglq`  
 DOUBLE COMPLEX for `zunglq`

Arrays:

`a( lda, * )` and `tau( * )` are the arrays returned by `sgelqf/dgelqf`.

The second dimension of `a` must be at least  $\max(1, n)$ .  
 The dimension of `tau` must be at least  $\max(1, k)$ .

`work(lwork)` is a workspace array.

`lda` INTEGER. The first dimension of `a`; at least  $\max(1, m)$ .

`lwork` INTEGER. The size of the `work` array; at least  $\max(1, m)$ .  
 See *Application notes* for the suggested value of `lwork`.

## Output Parameters

<code>a</code>	Overwritten by $m$ leading rows of the $n$ by $n$ unitary matrix $Q$ .
<code>work(1)</code>	If <code>info</code> = 0, on exit <code>work(1)</code> contains the minimum value of <code>lwork</code> required for optimum performance. Use this <code>lwork</code> for subsequent runs.
<code>info</code>	INTEGER. If <code>info</code> = 0, the execution is successful. If <code>info</code> = $-i$ , the $i$ th parameter had an illegal value.

## Application Notes

For better performance, try using `lwork = m * blocksize`, where `blocksize` is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*. If you are in doubt how much workspace to supply, use a generous value of `lwork` for the first run. On exit, examine `work(1)` and use this value for subsequent runs.

The computed  $Q$  differs from an exactly unitary matrix by a matrix  $E$  such that  $\|E\|_2 = O(\epsilon) \|A\|_2$  where  $\epsilon$  is the machine precision.

The total number of floating-point operations is approximately  $16 * m * n * k - 8 * (m + n) * k^2 + (16/3) * k^3$ .  
 If  $m = k$ , the number is approximately  $(8/3) * m^2 * (3n - m)$ .

The real counterpart of this routine is [?orglq](#).

## ?unmlq

*Multiplies a complex matrix by the unitary matrix Q of the LQ factorization formed by ?gelqf.*

---

```
call cunmlq ( side,trans,m,n,k,a,lda,tau,c,ldc,work,lwork,info )
call zunmlq ( side,trans,m,n,k,a,lda,tau,c,ldc,work,lwork,info )
```

### Discussion

The routine multiplies a real  $m$ -by- $n$  matrix  $C$  by  $Q$  or  $Q^H$ , where  $Q$  is the unitary matrix  $Q$  of the  $LQ$  factorization formed by the routine `cgelqf/zgelqf` (see [page 5-25](#)).

Depending on the parameters `side` and `trans`, the routine can form one of the matrix products  $QC$ ,  $Q^HC$ ,  $CQ$ , or  $CQ^H$  (overwriting the result on  $C$ ).

### Input Parameters

<code>side</code>	CHARACTER*1. Must be either ' <code>L</code> ' or ' <code>R</code> '. If <code>side = 'L'</code> , $Q$ or $Q^H$ is applied to $C$ from the left. If <code>side = 'R'</code> , $Q$ or $Q^H$ is applied to $C$ from the right.
<code>trans</code>	CHARACTER*1. Must be either ' <code>N</code> ' or ' <code>C</code> '. If <code>trans = 'N'</code> , the routine multiplies $C$ by $Q$ . If <code>trans = 'C'</code> , the routine multiplies $C$ by $Q^H$ .
<code>m</code>	INTEGER. The number of rows in the matrix $C$ ( $m \geq 0$ ).
<code>n</code>	INTEGER. The number of columns in $C$ ( $n \geq 0$ ).
<code>k</code>	INTEGER. The number of elementary reflectors whose product defines the matrix $Q$ . Constraints: $0 \leq k \leq m$ if <code>side = 'L'</code> ; $0 \leq k \leq n$ if <code>side = 'R'</code> .
<code>a,work,tau,c</code>	COMPLEX for <code>cunmlq</code> DOUBLE COMPLEX for <code>zunmlq</code> . Arrays: $a(1:lda, 1:n)$ and $tau(1:k)$ are arrays returned by <code>?gelqf</code> .

The second dimension of  $a$  must be:

at least  $\max(1, m)$  if  $\text{side} = 'L'$ ;

at least  $\max(1, n)$  if  $\text{side} = 'R'$ .

The dimension of  $\tau_{\alpha}$  must be at least  $\max(1, k)$ .

$c(\text{ldc}, *)$  contains the matrix  $C$ .

The second dimension of  $c$  must be at least  $\max(1, n)$

$\text{work}(\text{lwork})$  is a workspace array.

$\text{lda}$  INTEGER. The first dimension of  $a$ ;  $\text{lda} \geq \max(1, k)$ .

$\text{ldc}$  INTEGER. The first dimension of  $c$ ;  $\text{ldc} \geq \max(1, m)$ .

$\text{lwork}$  INTEGER. The size of the  $\text{work}$  array. Constraints:

$\text{lwork} \geq \max(1, n)$  if  $\text{side} = 'L'$ ;

$\text{lwork} \geq \max(1, m)$  if  $\text{side} = 'R'$ .

See *Application notes* for the suggested value of  $\text{lwork}$ .

## Output Parameters

$c$  Overwritten by the product  $QC$ ,  $Q^H C$ ,  $CQ$ , or  $CQ^H$  (as specified by  $\text{side}$  and  $\text{trans}$ ).

$\text{work}(1)$  If  $\text{info} = 0$ , on exit  $\text{work}(1)$  contains the minimum value of  $\text{lwork}$  required for optimum performance. Use this  $\text{lwork}$  for subsequent runs.

$\text{info}$  INTEGER.  
If  $\text{info} = 0$ , the execution is successful.  
If  $\text{info} = -i$ , the  $i$ th parameter had an illegal value.

## Application Notes

For better performance, try using  $\text{lwork} = n * \text{blocksize}$  (if  $\text{side} = 'L'$ ) or  $\text{lwork} = m * \text{blocksize}$  (if  $\text{side} = 'R'$ ) where  $\text{blocksize}$  is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*. If you are in doubt how much workspace to supply, use a generous value of  $\text{lwork}$  for the first run. On exit, examine  $\text{work}(1)$  and use this value for subsequent runs.

The real counterpart of this routine is [?ormlq](#).

## ?geqlf

*Computes the QL factorization of a general m by n matrix.*

---

```
call sgeqlf ( m, n, a, lda, tau, work, lwork, info )
call dgeqlf ( m, n, a, lda, tau, work, lwork, info )
call cgeqlf ( m, n, a, lda, tau, work, lwork, info )
call zgeqlf ( m, n, a, lda, tau, work, lwork, info )
```

### Discussion

The routine forms the  $QL$  factorization of a general  $m$ -by- $n$  matrix  $A$ . No pivoting is performed.

The routine does not form the matrix  $Q$  explicitly. Instead,  $Q$  is represented as a product of  $\min(m, n)$  elementary reflectors. Routines are provided to work with  $Q$  in this representation.

### Input Parameters

<i>m</i>	INTEGER. The number of rows in the matrix $A$ ( $m \geq 0$ ).
<i>n</i>	INTEGER. The number of columns in $A$ ( $n \geq 0$ ).
<i>a, work</i>	REAL for <code>sgeqlf</code> DOUBLE PRECISION for <code>dgeqlf</code> COMPLEX for <code>cgeqlf</code> DOUBLE COMPLEX for <code>zgeqlf</code> .
	Arrays: $a(lda, *)$ contains the matrix $A$ .
	The second dimension of $a$ must be at least $\max(1, n)$ .
	$work(lwork)$ is a workspace array.
<i>lda</i>	INTEGER. The first dimension of $a$ ; at least $\max(1, m)$ .
<i>lwork</i>	INTEGER. The size of the $work$ array; at least $\max(1, n)$ . See <i>Application notes</i> for the suggested value of <i>lwork</i> .

## Output Parameters

- a* Overwritten on exit by the factorization data as follows:
- if  $m \geq n$ , the lower triangle of the subarray  $a(m-n+1:m, 1:n)$  contains the  $n$ -by- $n$  lower triangular matrix  $L$ ;
  - if  $m \leq n$ , the elements on and below the  $(n-m)$ th superdiagonal contain the  $m$ -by- $n$  lower trapezoidal matrix  $L$ ;
  - in both cases, the remaining elements, with the array *tau*, represent the orthogonal/unitary matrix  $Q$  as a product of elementary reflectors.
- tau* `REAL` for `sgeqlf`  
`DOUBLE PRECISION` for `dgeqlf`  
`COMPLEX` for `cgeqlf`  
`DOUBLE COMPLEX` for `zgeqlf`.  
 Array, `DIMENSION` at least  $\max(1, \min(m, n))$ .  
 Contains scalar factors of the elementary reflectors for the matrix  $Q$ .
- work(1)* If *info* = 0, on exit *work(1)* contains the minimum value of *lwork* required for optimum performance.
- info* `INTEGER`.  
 If *info* = 0, the execution is successful.  
 If *info* =  $-i$ , the *i*th parameter had an illegal value.

## Application Notes

For better performance, try using  $lwork = n * blocksize$ , where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*. If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run. On exit, examine *work(1)* and use this value for subsequent runs.

Related routines include:

- [?orgql](#) to generate matrix  $Q$  (for real matrices);
- [?ungql](#) to generate matrix  $Q$  (for complex matrices);
- [?ormql](#) to apply matrix  $Q$  (for real matrices);
- [?unmql](#) to apply matrix  $Q$  (for complex matrices).

---

## ?orgql

*Generates the real matrix  $Q$  of the  $QL$  factorization formed by ?geqlf.*

---

```
call sorgql ( m, n, k, a, lda, tau, work, lwork, info )
call dorgql ( m, n, k, a, lda, tau, work, lwork, info )
```

### Discussion

The routine generates an  $m$ -by- $n$  real matrix  $Q$  with orthonormal columns, which is defined as the last  $n$  columns of a product of  $k$  elementary reflectors  $H_i$  of order  $m$ :  $Q = H_k \cdots H_2 H_1$  as returned by the routines sgeqlf/dgeqlf. Use this routine after a call to sgeqlf/dgeqlf.

### Input Parameters

$m$             INTEGER. The number of rows of the matrix  $Q$  ( $m \geq 0$ ).

$n$             INTEGER. The number of columns of the matrix  $Q$  ( $m \geq n \geq 0$ ).

$k$             INTEGER. The number of elementary reflectors whose product defines the matrix  $Q$  ( $n \geq k \geq 0$ ).

$a, \tau, \text{work}$     REAL for sorgql  
                   DOUBLE PRECISION for dorgql  
                   Arrays:  $a(\text{lda},*)$ ,  $\tau(*)$ ,  $\text{work}(lwork)$ .

On entry, the  $(n - k + i)$ th column of  $a$  must contain the vector which defines the elementary reflector  $H_i$ , for  $i = 1, 2, \dots, k$ , as returned by sgeqlf/dgeqlf in the last  $k$  columns of its array argument  $a$ ;  
 $\tau(i)$  must contain the scalar factor of the elementary reflector  $H_i$ , as returned by sgeqlf/dgeqlf;

The second dimension of  $a$  must be at least  $\max(1, n)$ .  
The dimension of  $\tau$  must be at least  $\max(1, k)$ .

$\text{work}(lwork)$  is a workspace array.

*lda*                    INTEGER. The first dimension of *a*; at least max(1, *m*).  
*lwork*                INTEGER. The size of the *work* array; at least max(1, *n*).  
                         See *Application notes* for the suggested value of *lwork*.

### Output Parameters

*a*                    Overwritten by the *m*-by-*n* matrix *Q*.  
*work(1)*            If *info* = 0, on exit *work(1)* contains the minimum  
                         value of *lwork* required for optimum performance. Use  
                         this *lwork* for subsequent runs.  
*info*                INTEGER.  
                         If *info* = 0, the execution is successful.  
                         If *info* = -*i*, the *i*th parameter had an illegal value.

### Application Notes

For better performance, try using *lwork* = *n*\**blocksize*, where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*. If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run. On exit, examine *work(1)* and use this value for subsequent runs.

The complex counterpart of this routine is [?ungql](#).

---

## ?ungql

*Generates the complex matrix  $Q$  of the  $QL$  factorization formed by ?geqlf.*

---

```
call cungql ( m, n, k, a, lda, tau, work, lwork, info )
call zungql ( m, n, k, a, lda, tau, work, lwork, info )
```

### Discussion

The routine generates an  $m$ -by- $n$  complex matrix  $Q$  with orthonormal columns, which is defined as the last  $n$  columns of a product of  $k$  elementary reflectors  $H_i$  of order  $m$ :  $Q = H_k \cdots H_2 H_1$  as returned by the routines [cgeqlf/zgeqlf](#). Use this routine after a call to [cgeqlf/zgeqlf](#).

### Input Parameters

$m$	<b>INTEGER</b> . The number of rows of the matrix $Q$ ( $m \geq 0$ ).
$n$	<b>INTEGER</b> . The number of columns of the matrix $Q$ ( $m \geq n \geq 0$ ).
$k$	<b>INTEGER</b> . The number of elementary reflectors whose product defines the matrix $Q$ ( $n \geq k \geq 0$ ).
$a, \tau, \text{work}$	<b>COMPLEX</b> for <code>cungql</code> <b>DOUBLE COMPLEX</b> for <code>zungql</code> Arrays: $a(lda, *), \tau(*), \text{work}(lwork)$ .  On entry, the $(n - k + i)$ th column of $a$ must contain the vector which defines the elementary reflector $H_i$ , for $i = 1, 2, \dots, k$ , as returned by <code>cgeqlf/zgeqlf</code> in the last $k$ columns of its array argument $a$ ; $\tau(i)$ must contain the scalar factor of the elementary reflector $H_i$ , as returned by <code>cgeqlf/zgeqlf</code> ; The second dimension of $a$ must be at least $\max(1, n)$ . The dimension of $\tau$ must be at least $\max(1, k)$ . $\text{work}(lwork)$ is a workspace array.

*lda*                    INTEGER. The first dimension of *a*; at least max(1, *m*).  
*lwork*                INTEGER. The size of the *work* array; at least max(1, *n*).  
                         See *Application notes* for the suggested value of *lwork*.

### Output Parameters

*a*                    Overwritten by the *m*-by-*n* matrix *Q*.  
*work(1)*            If *info* = 0, on exit *work(1)* contains the minimum  
                         value of *lwork* required for optimum performance. Use  
                         this *lwork* for subsequent runs.  
*info*                INTEGER.  
                         If *info* = 0, the execution is successful.  
                         If *info* = -*i*, the *i*th parameter had an illegal value.

### Application Notes

For better performance, try using *lwork* = *n*\**blocksize*, where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*. If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run. On exit, examine *work(1)* and use this value for subsequent runs.

The real counterpart of this routine is [?orgl](#).

## ?ormql

*Multiplies a real matrix by the orthogonal matrix Q of the QL factorization formed by ?geqlf.*

---

```
call sormql ( side,trans,m,n,k,a,lda,tau,c,ldc,work,lwork,info )
call dormql ( side,trans,m,n,k,a,lda,tau,c,ldc,work,lwork,info )
```

### Discussion

This routine multiplies a real  $m$ -by- $n$  matrix  $C$  by  $Q$  or  $Q^T$ , where  $Q$  is the orthogonal matrix  $Q$  of the  $QL$  factorization formed by the routine [sgeqlf/dgeqlf](#).

Depending on the parameters `side` and `trans`, the routine `?ormql` can form one of the matrix products  $QC$ ,  $Q^TC$ ,  $CQ$ , or  $CQ^T$  (overwriting the result over  $C$ ).

### Input Parameters

<code>side</code>	CHARACTER*1. Must be either ' <code>L</code> ' or ' <code>R</code> '. If <code>side</code> = ' <code>L</code> ', $Q$ or $Q^T$ is applied to $C$ from the left. If <code>side</code> = ' <code>R</code> ', $Q$ or $Q^T$ is applied to $C$ from the right.
<code>trans</code>	CHARACTER*1. Must be either ' <code>N</code> ' or ' <code>T</code> '. If <code>trans</code> = ' <code>N</code> ', the routine multiplies $C$ by $Q$ . If <code>trans</code> = ' <code>T</code> ', the routine multiplies $C$ by $Q^T$ .
<code>m</code>	INTEGER. The number of rows in the matrix $C$ ( $m \geq 0$ ).
<code>n</code>	INTEGER. The number of columns in $C$ ( $n \geq 0$ ).
<code>k</code>	INTEGER. The number of elementary reflectors whose product defines the matrix $Q$ . Constraints: $0 \leq k \leq m$ if <code>side</code> = ' <code>L</code> '; $0 \leq k \leq n$ if <code>side</code> = ' <code>R</code> '.

`a, tau, c, work`    **REAL** for `sormql`  
**DOUBLE PRECISION** for `dormql`.  
 Arrays: `a(lda, *), tau(*), c(ldc, *), work(lwork)`.

On entry, the  $i$ th column of `a` must contain the vector which defines the elementary reflector  $H_i$ , for  $i = 1, 2, \dots, k$ , as returned by `sgeqlf/dgeqlf` in the last  $k$  columns of its array argument `a`.

The second dimension of `a` must be at least  $\max(1, k)$ .

`tau(i)` must contain the scalar factor of the elementary reflector  $H_i$ , as returned by `sgeqlf/dgeqlf`.

The dimension of `tau` must be at least  $\max(1, k)$ .

`c(ldc, *)` contains the  $m$ -by- $n$  matrix  $C$ .

The second dimension of `c` must be at least  $\max(1, n)$

`work(lwork)` is a workspace array.

`lda`    **INTEGER**. The first dimension of `a`;

if `side = 'L'`, `lda`  $\geq \max(1, m)$ ;

if `side = 'R'`, `lda`  $\geq \max(1, n)$ .

`ldc`    **INTEGER**. The first dimension of `c`; `ldc`  $\geq \max(1, m)$ .

`lwork`    **INTEGER**. The size of the `work` array. Constraints:

`lwork`  $\geq \max(1, n)$  if `side = 'L'`;

`lwork`  $\geq \max(1, m)$  if `side = 'R'`.

See *Application notes* for the suggested value of `lwork`.

## Output Parameters

`c`    Overwritten by the product  $QC$ ,  $Q^TC$ ,  $CQ$ , or  $CQ^T$  (as specified by `side` and `trans`).

`work(1)`    If `info` = 0, on exit `work(1)` contains the minimum value of `lwork` required for optimum performance. Use this `lwork` for subsequent runs.

`info`    **INTEGER**.

If `info` = 0, the execution is successful.

If `info` =  $-i$ , the  $i$ th parameter had an illegal value.

## Application Notes

For better performance, try using `lwork = n*blocksize` (if `side = 'L'`) or `lwork = m*blocksize` (if `side = 'R'`) where `blocksize` is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*. If you are in doubt how much workspace to supply, use a generous value of `lwork` for the first run. On exit, examine `work(1)` and use this value for subsequent runs.

The complex counterpart of this routine is [?unmql](#).

## ?unmql

*Multiplies a complex matrix by the unitary matrix Q of the QL factorization formed by ?geqlf.*

```
call cunmql ( side,trans,m,n,k,a,lda,tau,c,ldc,work,lwork,info )
call zunmql ( side,trans,m,n,k,a,lda,tau,c,ldc,work,lwork,info )
```

### Discussion

The routine multiplies a complex  $m$ -by- $n$  matrix  $C$  by  $Q$  or  $Q^H$ , where  $Q$  is the unitary matrix  $Q$  of the  $QL$  factorization formed by the routine [cgeqlf/zgeqlf](#).

Depending on the parameters `side` and `trans`, the routine `?unmql` can form one of the matrix products  $QC$ ,  $Q^HC$ ,  $CQ$ , or  $CQ^H$  (overwriting the result over  $C$ ).

### Input Parameters

<code>side</code>	CHARACTER*1. Must be either ' <code>L</code> ' or ' <code>R</code> '. If <code>side = 'L'</code> , $Q$ or $Q^H$ is applied to $C$ from the left. If <code>side = 'R'</code> , $Q$ or $Q^H$ is applied to $C$ from the right.
<code>trans</code>	CHARACTER*1. Must be either ' <code>N</code> ' or ' <code>C</code> '. If <code>trans = 'N'</code> , the routine multiplies $C$ by $Q$ . If <code>trans = 'C'</code> , the routine multiplies $C$ by $Q^H$ .
<code>m</code>	INTEGER. The number of rows in the matrix $C$ ( $m \geq 0$ ).
<code>n</code>	INTEGER. The number of columns in $C$ ( $n \geq 0$ ).
<code>k</code>	INTEGER. The number of elementary reflectors whose product defines the matrix $Q$ . Constraints: $0 \leq k \leq m$ if <code>side = 'L'</code> ; $0 \leq k \leq n$ if <code>side = 'R'</code> .

<i>a, tau, c, work</i>	COMPLEX for <code>cunmql</code> DOUBLE COMPLEX for <code>zunmql</code> . Arrays: <i>a( lda, * ), tau( * ), c( ldc, * ), work( lwork )</i> .
	On entry, the <i>i</i> th column of <i>a</i> must contain the vector which defines the elementary reflector $H_i$ , for $i = 1, 2, \dots, k$ , as returned by <code>cgeqlf/zgeqlf</code> in the last <i>k</i> columns of its array argument <i>a</i> . The second dimension of <i>a</i> must be at least $\max(1, k)$ . <i>tau(i)</i> must contain the scalar factor of the elementary reflector $H_i$ , as returned by <code>cgeqlf/zgeqlf</code> . The dimension of <i>tau</i> must be at least $\max(1, k)$ . <i>c( ldc, * )</i> contains the <i>m</i> -by- <i>n</i> matrix <i>C</i> . The second dimension of <i>c</i> must be at least $\max(1, n)$ . <i>work( lwork )</i> is a workspace array.
<i>lda</i>	<b>INTEGER.</b> The first dimension of <i>a</i> ; if <i>side</i> = 'L', <i>lda</i> $\geq \max(1, m)$ ; if <i>side</i> = 'R', <i>lda</i> $\geq \max(1, n)$ .
<i>ldc</i>	<b>INTEGER.</b> The first dimension of <i>c</i> ; <i>ldc</i> $\geq \max(1, m)$ .
<i>lwork</i>	<b>INTEGER.</b> The size of the <i>work</i> array. Constraints: <i>lwork</i> $\geq \max(1, n)$ if <i>side</i> = 'L'; <i>lwork</i> $\geq \max(1, m)$ if <i>side</i> = 'R'. See <i>Application notes</i> for the suggested value of <i>lwork</i> .

## Output Parameters

<i>c</i>	Overwritten by the product $QC$ , $Q^H C$ , $CQ$ , or $CQ^H$ (as specified by <i>side</i> and <i>trans</i> ).
<i>work( 1 )</i>	If <i>info</i> = 0, on exit <i>work( 1 )</i> contains the minimum value of <i>lwork</i> required for optimum performance. Use this <i>lwork</i> for subsequent runs.
<i>info</i>	<b>INTEGER.</b> If <i>info</i> = 0, the execution is successful. If <i>info</i> = <i>-i</i> , the <i>i</i> th parameter had an illegal value.

### Application Notes

For better performance, try using `lwork = n*blocksize` (if `side = 'L'`) or `lwork = m*blocksize` (if `side = 'R'`) where `blocksize` is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*. If you are in doubt how much workspace to supply, use a generous value of `lwork` for the first run. On exit, examine `work(1)` and use this value for subsequent runs.

The real counterpart of this routine is [?ormql](#).

---

## ?gerqf

*Computes the RQ factorization of a general m by n matrix.*

---

```
call sgerqf ( m, n, a, lda, tau, work, lwork, info )
call dgerqf ( m, n, a, lda, tau, work, lwork, info )
call cgerqf ( m, n, a, lda, tau, work, lwork, info )
call zgerqf ( m, n, a, lda, tau, work, lwork, info )
```

### Discussion

The routine forms the *RQ* factorization of a general  $m$ -by- $n$  matrix  $A$ . No pivoting is performed.

The routine does not form the matrix  $Q$  explicitly. Instead,  $Q$  is represented as a product of  $\min(m, n)$  elementary reflectors. Routines are provided to work with  $Q$  in this representation.

### Input Parameters

$m$	INTEGER. The number of rows in the matrix $A$ ( $m \geq 0$ ).
$n$	INTEGER. The number of columns in $A$ ( $n \geq 0$ ).
$a, work$	REAL for <code>sgerqf</code> DOUBLE PRECISION for <code>dgerqf</code> COMPLEX for <code>cgerqf</code> DOUBLE COMPLEX for <code>zgerqf</code> .
Arrays:	
$a(lda, *)$ contains the $m$ -by- $n$ matrix $A$ .	
The second dimension of $a$ must be at least $\max(1, n)$ .	
$work(lwork)$ is a workspace array.	
$lda$	INTEGER. The first dimension of $a$ ; at least $\max(1, m)$ .
$lwork$	INTEGER. The size of the $work$ array; $lwork \geq \max(1, m)$ . See <a href="#">Application notes</a> for the suggested value of $lwork$ .

## Output Parameters

<i>a</i>	Overwritten on exit by the factorization data as follows: if $m \leq n$ , the upper triangle of the subarray $a(1:m, n-m+1:n)$ contains the $m$ -by- $m$ upper triangular matrix $R$ ; if $m \geq n$ , the elements on and above the $(m-n)$ th subdiagonal contain the $m$ -by- $n$ upper trapezoidal matrix $R$ ; in both cases, the remaining elements, with the array <i>tau</i> , represent the orthogonal/unitary matrix $Q$ as a product of $\min(m,n)$ elementary reflectors.
<i>tau</i>	<i>REAL</i> for <i>sgerqf</i> <i>DOUBLE PRECISION</i> for <i>dgerqf</i> <i>COMPLEX</i> for <i>cgerqf</i> <i>DOUBLE COMPLEX</i> for <i>zgerqf</i> . Array, <i>DIMENSION</i> at least $\max(1, \min(m, n))$ . Contains scalar factors of the elementary reflectors for the matrix $Q$ .
<i>work(1)</i>	If <i>info</i> = 0, on exit <i>work(1)</i> contains the minimum value of <i>lwork</i> required for optimum performance.
<i>info</i>	<i>INTEGER</i> . If <i>info</i> = 0, the execution is successful. If <i>info</i> = $-i$ , the <i>i</i> th parameter had an illegal value.

## Application Notes

For better performance, try using *lwork* =  $m * blocksize$ , where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*. If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run. On exit, examine *work(1)* and use this value for subsequent runs.

Related routines include:

<u>?orgqr</u>	to generate matrix Q (for real matrices);
<u>?ungrq</u>	to generate matrix Q (for complex matrices);
<u>?ormrq</u>	to apply matrix Q (for real matrices);
<u>?unmrq</u>	to apply matrix Q (for complex matrices).

---

## ?orgqr

*Generates the real matrix Q of the RQ factorization formed by ?gerqf.*

---

```
call sorgqr ( m, n, k, a, lda, tau, work, lwork, info )
call dorgqr ( m, n, k, a, lda, tau, work, lwork, info )
```

### Discussion

The routine generates an  $m$ -by- $n$  real matrix  $Q$  with orthonormal rows, which is defined as the last  $m$  rows of a product of  $k$  elementary reflectors  $H_i$  of order  $n$ :  $Q = H_1 H_2 \cdots H_k$  as returned by the routines [sgerqf/dgerqf](#). Use this routine after a call to [sgerqf/dgerqf](#).

### Input Parameters

$m$	<b>INTEGER.</b> The number of rows of the matrix $Q$ ( $m \geq 0$ ).
$n$	<b>INTEGER.</b> The number of columns of the matrix $Q$ ( $n \geq m$ ).
$k$	<b>INTEGER.</b> The number of elementary reflectors whose product defines the matrix $Q$ ( $m \geq k \geq 0$ ).
$a, \tau, \text{work}$	<b>REAL</b> for <b>sorgqr</b> <b>DOUBLE PRECISION</b> for <b>dorgqr</b> Arrays: $a(lda, *), \tau(*), \text{work}(lwork)$ .  On entry, the $(m - k + i)$ th row of $a$ must contain the vector which defines the elementary reflector $H_i$ , for $i = 1, 2, \dots, k$ , as returned by <b>sgerqf/dgerqf</b> in the last $k$ rows of its array argument $a$ ; $\tau(i)$ must contain the scalar factor of the elementary reflector $H_i$ , as returned by <b>sgerqf/dgerqf</b> ; The second dimension of $a$ must be at least $\max(1, n)$ . The dimension of $\tau$ must be at least $\max(1, k)$ . $\text{work}(lwork)$ is a workspace array.

*lda*                    INTEGER. The first dimension of *a*; at least max(1, *m*).  
*lwork*                INTEGER. The size of the *work* array; at least max(1, *m*).  
                         See *Application notes* for the suggested value of *lwork*.

### Output Parameters

*a*                    Overwritten by the *m*-by-*n* matrix *Q*.  
*work(1)*            If *info* = 0, on exit *work(1)* contains the minimum  
                         value of *lwork* required for optimum performance. Use  
                         this *lwork* for subsequent runs.  
*info*                INTEGER.  
                         If *info* = 0, the execution is successful.  
                         If *info* = -*i*, the *i*th parameter had an illegal value.

### Application Notes

For better performance, try using *lwork* = *m*\**blocksize*, where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*. If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run. On exit, examine *work(1)* and use this value for subsequent runs.

The complex counterpart of this routine is [?ungrq](#).

---

## ?ungrq

*Generates the complex matrix  $Q$  of the RQ factorization formed by ?gerqf.*

---

```
call cungrq ( m, n, k, a, lda, tau, work, lwork, info )
call zungrq ( m, n, k, a, lda, tau, work, lwork, info )
```

### Discussion

The routine generates an  $m$ -by- $n$  complex matrix  $Q$  with orthonormal rows, which is defined as the last  $m$  rows of a product of  $k$  elementary reflectors  $H_i$  of order  $n$ :  $Q = H_1^H H_2^H \cdots H_k^H$  as returned by the routines sgerqf/dgerqf. Use this routine after a call to sgerqf/dgerqf.

### Input Parameters

$m$	<b>INTEGER</b> . The number of rows of the matrix $Q$ ( $m \geq 0$ ).
$n$	<b>INTEGER</b> . The number of columns of the matrix $Q$ ( $n \geq m$ ).
$k$	<b>INTEGER</b> . The number of elementary reflectors whose product defines the matrix $Q$ ( $m \geq k \geq 0$ ).
$a, \tau, \text{work}$	<b>REAL</b> for cungrq <b>DOUBLE PRECISION</b> for zungrq Arrays: $a(lda, *), \tau(*), \text{work}(lwork)$ .  On entry, the $(m - k + i)$ th row of $a$ must contain the vector which defines the elementary reflector $H_i$ , for $i = 1, 2, \dots, k$ , as returned by sgerqf/dgerqf in the last $k$ rows of its array argument $a$ ; $\tau(i)$ must contain the scalar factor of the elementary reflector $H_i$ , as returned by sgerqf/dgerqf; The second dimension of $a$ must be at least $\max(1, n)$ . The dimension of $\tau$ must be at least $\max(1, k)$ . $\text{work}(lwork)$ is a workspace array.

*lda*                    INTEGER. The first dimension of *a*; at least max(1, *m*).  
*lwork*                INTEGER. The size of the *work* array; at least max(1, *m*).  
                         See *Application notes* for the suggested value of *lwork*.

### Output Parameters

*a*                    Overwritten by the *m*-by-*n* matrix *Q*.  
*work(1)*            If *info* = 0, on exit *work(1)* contains the minimum  
                         value of *lwork* required for optimum performance. Use  
                         this *lwork* for subsequent runs.  
*info*                INTEGER.  
                         If *info* = 0, the execution is successful.  
                         If *info* = -*i*, the *i*th parameter had an illegal value.

### Application Notes

For better performance, try using *lwork* = *m*\**blocksize*, where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*. If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run. On exit, examine *work(1)* and use this value for subsequent runs.

The real counterpart of this routine is [?orgqr](#).

## ?ormrq

*Multiplies a real matrix by the orthogonal matrix Q of the RQ factorization formed by ?gerqf.*

---

```
call sormrq ( side,trans,m,n,k,a,lda,tau,c,ldc,work,lwork,info )
call dormrq ( side,trans,m,n,k,a,lda,tau,c,ldc,work,lwork,info )
```

### Discussion

The routine multiplies a real  $m$ -by- $n$  matrix  $C$  by  $Q$  or  $Q^T$ , where  $Q$  is the real orthogonal matrix defined as a product of  $k$  elementary reflectors  $H_i$ :  $Q = H_1 H_2 \cdots H_k$  as returned by the  $RQ$  factorization routine [sgerqf/dgerqf](#).

Depending on the parameters `side` and `trans`, the routine can form one of the matrix products  $QC$ ,  $Q^TC$ ,  $CQ$ , or  $CQ^T$  (overwriting the result over  $C$ ).

### Input Parameters

<code>side</code>	CHARACTER*1. Must be either ' <code>L</code> ' or ' <code>R</code> '. If <code>side</code> = ' <code>L</code> ', $Q$ or $Q^T$ is applied to $C$ from the left. If <code>side</code> = ' <code>R</code> ', $Q$ or $Q^T$ is applied to $C$ from the right.
<code>trans</code>	CHARACTER*1. Must be either ' <code>N</code> ' or ' <code>T</code> '. If <code>trans</code> = ' <code>N</code> ', the routine multiplies $C$ by $Q$ . If <code>trans</code> = ' <code>T</code> ', the routine multiplies $C$ by $Q^T$ .
<code>m</code>	INTEGER. The number of rows in the matrix $C$ ( $m \geq 0$ ).
<code>n</code>	INTEGER. The number of columns in $C$ ( $n \geq 0$ ).
<code>k</code>	INTEGER. The number of elementary reflectors whose product defines the matrix $Q$ . Constraints: $0 \leq k \leq m$ , if <code>side</code> = ' <code>L</code> '; $0 \leq k \leq n$ , if <code>side</code> = ' <code>R</code> '.

`a, tau, c, work`    REAL for `sormrq`  
DOUBLE PRECISION for `dormrq`.  
 Arrays: `a(lda,*)`, `tau(*)`, `c(ldc,*)`,  
`work(lwork)`.

On entry, the  $i$ th row of `a` must contain the vector which defines the elementary reflector  $H_i$ , for  $i = 1, 2, \dots, k$ , as returned by `sgerqf/dgerqf` in the last  $k$  rows of its array argument `a`.

The second dimension of `a` must be at least  $\max(1, m)$  if `side = 'L'`, and at least  $\max(1, n)$  if `side = 'R'`.

`tau(i)` must contain the scalar factor of the elementary reflector  $H_i$ , as returned by `sgerqf/dgerqf`.

The dimension of `tau` must be at least  $\max(1, k)$ .

`c(ldc,*)` contains the  $m$ -by- $n$  matrix  $C$ .

The second dimension of `c` must be at least  $\max(1, n)$

`work(lwork)` is a workspace array.

`lda`                    INTEGER. The first dimension of `a`;  $lda \geq \max(1, k)$ .

`ldc`                    INTEGER. The first dimension of `c`;  $ldc \geq \max(1, m)$ .

`lwork`                    INTEGER. The size of the `work` array. Constraints:

$lwork \geq \max(1, n)$  if `side = 'L'`;

$lwork \geq \max(1, m)$  if `side = 'R'`.

See *Application notes* for the suggested value of `lwork`.

## Output Parameters

`c`                    Overwritten by the product  $QC$ ,  $Q^TC$ ,  $CQ$ , or  $CQ^T$  (as specified by `side` and `trans`).

`work(1)`                    If `info = 0`, on exit `work(1)` contains the minimum value of `lwork` required for optimum performance. Use this `lwork` for subsequent runs.

`info`                    INTEGER.

If `info = 0`, the execution is successful.

If `info = -i`, the  $i$ th parameter had an illegal value.

## Application Notes

For better performance, try using `lwork = n*blocksize` (if `side = 'L'`) or `lwork = m*blocksize` (if `side = 'R'`) where `blocksize` is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*. If you are in doubt how much workspace to supply, use a generous value of `lwork` for the first run. On exit, examine `work(1)` and use this value for subsequent runs.

The complex counterpart of this routine is [?unmrq](#).

## ?unmrq

*Multiplies a complex matrix by the unitary matrix Q of the RQ factorization formed by ?gerqf.*

```
call cunmrq ( side,trans,m,n,k,a,lda,tau,c,ldc,work,lwork,info )
call zunmrq ( side,trans,m,n,k,a,lda,tau,c,ldc,work,lwork,info )
```

### Discussion

The routine multiplies a complex  $m$ -by- $n$  matrix  $C$  by  $Q$  or  $Q^H$ , where  $Q$  is the complex unitary matrix defined as a product of  $k$  elementary reflectors  $H_i : Q = H_1^H H_2^H \cdots H_k^H$  as returned by the  $RQ$  factorization routine [cgerqf/zgerqf](#).

Depending on the parameters `side` and `trans`, the routine can form one of the matrix products  $QC$ ,  $Q^HC$ ,  $CQ$ , or  $CQ^H$  (overwriting the result over  $C$ ).

### Input Parameters

<code>side</code>	CHARACTER*1. Must be either ' <code>L</code> ' or ' <code>R</code> '. If <code>side = 'L'</code> , $Q$ or $Q^H$ is applied to $C$ from the left. If <code>side = 'R'</code> , $Q$ or $Q^H$ is applied to $C$ from the right.
<code>trans</code>	CHARACTER*1. Must be either ' <code>N</code> ' or ' <code>C</code> '. If <code>trans = 'N'</code> , the routine multiplies $C$ by $Q$ . If <code>trans = 'C'</code> , the routine multiplies $C$ by $Q^H$ .
<code>m</code>	INTEGER. The number of rows in the matrix $C$ ( $m \geq 0$ ).
<code>n</code>	INTEGER. The number of columns in $C$ ( $n \geq 0$ ).
<code>k</code>	INTEGER. The number of elementary reflectors whose product defines the matrix $Q$ . Constraints: $0 \leq k \leq m$ , if <code>side = 'L'</code> ; $0 \leq k \leq n$ , if <code>side = 'R'</code> .

*a, tau, c, work*    COMPLEX for `cunmrq`  
                   DOUBLE COMPLEX for `zunmrq`.  
                   Arrays: *a( lda, \* ), tau( \* ), c( ldc, \* ), work( lwork )*.

On entry, the *i*th row of *a* must contain the vector which defines the elementary reflector  $H_i$ , for  $i = 1, 2, \dots, k$ , as returned by `cgerqf/zgerqf` in the last *k* rows of its array argument *a*.  
     The second dimension of *a* must be at least  $\max(1, m)$  if *side* = 'L', and at least  $\max(1, n)$  if *side* = 'R'.  
     *tau(i)* must contain the scalar factor of the elementary reflector  $H_i$ , as returned by `cgerqf/zgerqf`.  
     The dimension of *tau* must be at least  $\max(1, k)$ .  
     *c( ldc, \* )* contains the *m*-by-*n* matrix *C*.  
     The second dimension of *c* must be at least  $\max(1, n)$   
     *work( lwork )* is a workspace array.

*lda*            INTEGER. The first dimension of *a*; *lda*  $\geq \max(1, k)$ .  
*ldc*            INTEGER. The first dimension of *c*; *ldc*  $\geq \max(1, m)$ .  
*lwork*            INTEGER. The size of the *work* array. Constraints:  
                   *lwork*  $\geq \max(1, n)$  if *side* = 'L';  
                   *lwork*  $\geq \max(1, m)$  if *side* = 'R'.  
     See *Application notes* for the suggested value of *lwork*.

## Output Parameters

*c*               Overwritten by the product  $QC$ ,  $Q^H C$ ,  $CQ$ , or  $CQ^H$   
                   (as specified by *side* and *trans*).  
*work( 1 )*       If *info* = 0, on exit *work( 1 )* contains the minimum  
                   value of *lwork* required for optimum performance. Use  
                   this *lwork* for subsequent runs.  
*info*             INTEGER.  
                   If *info* = 0, the execution is successful.  
                   If *info* = -*i*, the *i*th parameter had an illegal value.

### Application Notes

For better performance, try using `lwork = n*blocksize` (if `side = 'L'`) or `lwork = m*blocksize` (if `side = 'R'`) where `blocksize` is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*. If you are in doubt how much workspace to supply, use a generous value of `lwork` for the first run. On exit, examine `work(1)` and use this value for subsequent runs.

The real counterpart of this routine is [?ormrq](#).

---

**?tzrzf**

*Reduces the upper trapezoidal matrix A  
to upper triangular form.*

---

```
call stzrzf ( m, n, a, lda, tau, work, lwork, info )
call dtzrzf ( m, n, a, lda, tau, work, lwork, info )
call ctzrzf ( m, n, a, lda, tau, work, lwork, info )
call ztzrzf ( m, n, a, lda, tau, work, lwork, info )
```

**Discussion**

This routine reduces the  $m$ -by- $n$  ( $m \leq n$ ) real/complex upper trapezoidal matrix  $A$  to upper triangular form by means of orthogonal/unitary transformations. The upper trapezoidal matrix  $A$  is factored as

$$A = (R \ 0) * Z,$$

where  $Z$  is an  $n$ -by- $n$  orthogonal/unitary matrix and  $R$  is an  $m$ -by- $m$  upper triangular matrix.

**Input Parameters**

<i>m</i>	<b>INTEGER.</b> The number of rows in the matrix $A$ ( $m \geq 0$ ).
<i>n</i>	<b>INTEGER.</b> The number of columns in $A$ ( $n \geq m$ ).
<i>a, work</i>	<b>REAL</b> for <b>stzrzf</b> <b>DOUBLE PRECISION</b> for <b>dtzrzf</b> <b>COMPLEX</b> for <b>ctzrzf</b> <b>DOUBLE COMPLEX</b> for <b>ztzrzf</b> . Arrays: <i>a(lda,*)</i> , <i>work(lwork)</i> . The leading $m$ -by- $n$ upper trapezoidal part of the array <i>a</i> contains the matrix $A$ to be factorized. The second dimension of <i>a</i> must be at least $\max(1, n)$ . <i>work</i> is a workspace array.
<i>lda</i>	<b>INTEGER.</b> The first dimension of <i>a</i> ; at least $\max(1, m)$ .
<i>lwork</i>	<b>INTEGER.</b> The size of the <i>work</i> array;

$lwork \geq \max(1, m)$ .

See *Application notes* for the suggested value of  $lwork$ .

## Output Parameters

$a$	Overwritten on exit by the factorization data as follows: the leading $m$ -by- $m$ upper triangular part of $a$ contains the upper triangular matrix $R$ , and elements $m + 1$ to $n$ of the first $m$ rows of $a$ , with the array $tau$ , represent the orthogonal matrix $Z$ as a product of $m$ elementary reflectors.
$tau$	<b>REAL</b> for <code>stzrzf</code> <b>DOUBLE PRECISION</b> for <code>dtzrzf</code> <b>COMPLEX</b> for <code>ctzrzf</code> <b>DOUBLE COMPLEX</b> for <code>ztzrzf</code> . Array, <b>DIMENSION</b> at least $\max(1, m)$ . Contains scalar factors of the elementary reflectors for the matrix $Z$ .
$work(1)$	If $info = 0$ , on exit $work(1)$ contains the minimum value of $lwork$ required for optimum performance. Use this $lwork$ for subsequent runs.
$info$	<b>INTEGER</b> . If $info = 0$ , the execution is successful. If $info = -i$ , the $i$ th parameter had an illegal value.

## Application Notes

For better performance, try using  $lwork = m * blocksize$ , where  $blocksize$  is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use a generous value of  $lwork$  for the first run. On exit, examine  $work(1)$  and use this value for subsequent runs.

Related routines include:

- |                                     |   |
|-------------------------------------|---|
| <a href="#"><code>?ormrz</code></a> | to apply matrix Q (for real matrices);    |
| <a href="#"><code>?unmrz</code></a> | to apply matrix Q (for complex matrices). |

---

## ?ormrz

*Multiplies a real matrix by the orthogonal matrix defined from the factorization formed by ?tzrzf.*

---

```
call sormrz ( side,trans,m,n,k,l,a,lda,tau,c,ldc,work,lwork,info )
call dormrz ( side,trans,m,n,k,l,a,lda,tau,c,ldc,work,lwork,info )
```

### Discussion

The routine multiplies a real  $m$ -by- $n$  matrix  $C$  by  $Q$  or  $Q^T$ , where  $Q$  is the real orthogonal matrix defined as a product of  $k$  elementary reflectors  $H_i$ :  $Q = H_1 H_2 \cdots H_k$  as returned by the factorization routine [stzrzf/dtzrzf](#).

Depending on the parameters `side` and `trans`, the routine can form one of the matrix products  $QC$ ,  $Q^TC$ ,  $CQ$ , or  $CQ^T$  (overwriting the result over  $C$ ).

The matrix  $Q$  is of order  $m$  if `side = 'L'` and of order  $n$  if `side = 'R'`.

### Input Parameters

<code>side</code>	CHARACTER*1. Must be either ' <code>L</code> ' or ' <code>R</code> '. If <code>side = 'L'</code> , $Q$ or $Q^T$ is applied to $C$ from the left. If <code>side = 'R'</code> , $Q$ or $Q^T$ is applied to $C$ from the right.
<code>trans</code>	CHARACTER*1. Must be either ' <code>N</code> ' or ' <code>T</code> '. If <code>trans = 'N'</code> , the routine multiplies $C$ by $Q$ . If <code>trans = 'T'</code> , the routine multiplies $C$ by $Q^T$ .
<code>m</code>	INTEGER. The number of rows in the matrix $C$ ( $m \geq 0$ ).
<code>n</code>	INTEGER. The number of columns in $C$ ( $n \geq 0$ ).
<code>k</code>	INTEGER. The number of elementary reflectors whose product defines the matrix $Q$ . Constraints: $0 \leq k \leq m$ , if <code>side = 'L'</code> ; $0 \leq k \leq n$ , if <code>side = 'R'</code> .
<code>l</code>	INTEGER.

The number of columns of the matrix  $A$  containing the meaningful part of the Householder reflectors.

Constraints:

$0 \leq l \leq m$ , if  $\text{side} = 'L'$ ;  
 $0 \leq l \leq n$ , if  $\text{side} = 'R'$ .

$a, tau, c, work$

REAL for `sormrz`

DOUBLE PRECISION for `dormrz`.

Arrays:  $a(1:\text{lda}, *)$ ,  $tau(*)$ ,  $c(1:\text{ldc}, *)$ ,  $work(1:\text{lwork})$ .

On entry, the  $i$ th row of  $a$  must contain the vector which defines the elementary reflector  $H_i$ , for  $i = 1, 2, \dots, k$ , as returned by `stzrzf/dtzrzf` in the last  $k$  rows of its array argument  $a$ .

The second dimension of  $a$  must be at least  $\max(1, m)$  if  $\text{side} = 'L'$ , and at least  $\max(1, n)$  if  $\text{side} = 'R'$ .

$tau(i)$  must contain the scalar factor of the elementary reflector  $H_i$ , as returned by `stzrzf/dtzrzf`.

The dimension of  $tau$  must be at least  $\max(1, k)$ .

$c(1:\text{ldc}, *)$  contains the  $m$ -by- $n$  matrix  $C$ .

The second dimension of  $c$  must be at least  $\max(1, n)$

$work(1:\text{lwork})$  is a workspace array.

$lda$

INTEGER. The first dimension of  $a$ ;  $lda \geq \max(1, k)$ .

$ldc$

INTEGER. The first dimension of  $c$ ;  $ldc \geq \max(1, m)$ .

$lwork$

INTEGER. The size of the  $work$  array. Constraints:

$lwork \geq \max(1, n)$  if  $\text{side} = 'L'$ ;

$lwork \geq \max(1, m)$  if  $\text{side} = 'R'$ .

See *Application notes* for the suggested value of  $lwork$ .

## Output Parameters

$c$

Overwritten by the product  $QC$ ,  $Q^T C$ ,  $CQ$ , or  $CQ^T$  (as specified by  $\text{side}$  and  $\text{trans}$ ).

$work(1)$

If  $\text{info} = 0$ , on exit  $work(1)$  contains the minimum value of  $lwork$  required for optimum performance. Use this  $lwork$  for subsequent runs.

*info*                    INTEGER.  
If *info* = 0, the execution is successful.  
If *info* = -*i*, the *i*th parameter had an illegal value.

### Application Notes

For better performance, try using *lwork* = *n*\**blocksize* (if *side* = 'L') or *lwork* = *m*\**blocksize* (if *side* = 'R') where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*. If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run. On exit, examine *work(1)* and use this value for subsequent runs.

The complex counterpart of this routine is [?unmrz](#).

## ?unmrz

*Multiplies a complex matrix by the unitary matrix defined from the factorization formed by ?tzrzf.*

```
call cunmrz ( side,trans,m,n,k,l,a,lda,tau,c,ldc,work,lwork,info )
call zunmrz ( side,trans,m,n,k,l,a,lda,tau,c,ldc,work,lwork,info )
```

### Discussion

The routine multiplies a complex  $m$ -by- $n$  matrix  $C$  by  $Q$  or  $Q^H$ , where  $Q$  is the unitary matrix defined as a product of  $k$  elementary reflectors  $H_i$ :

$Q = H_1^H H_2^H \cdots H_k^H$  as returned by the factorization routine

[ctzrzf/ztzrzf](#).

Depending on the parameters `side` and `trans`, the routine can form one of the matrix products  $QC$ ,  $Q^HC$ ,  $CQ$ , or  $CQ^H$  (overwriting the result over  $C$ ).

The matrix  $Q$  is of order  $m$  if `side = 'L'` and of order  $n$  if `side = 'R'`.

### Input Parameters

<code>side</code>	CHARACTER*1. Must be either ' <code>L</code> ' or ' <code>R</code> '. If <code>side = 'L'</code> , $Q$ or $Q^H$ is applied to $C$ from the left. If <code>side = 'R'</code> , $Q$ or $Q^H$ is applied to $C$ from the right.
<code>trans</code>	CHARACTER*1. Must be either ' <code>N</code> ' or ' <code>C</code> '. If <code>trans = 'N'</code> , the routine multiplies $C$ by $Q$ . If <code>trans = 'C'</code> , the routine multiplies $C$ by $Q^H$ .
<code>m</code>	INTEGER. The number of rows in the matrix $C$ ( $m \geq 0$ ).
<code>n</code>	INTEGER. The number of columns in $C$ ( $n \geq 0$ ).
<code>k</code>	INTEGER. The number of elementary reflectors whose product defines the matrix $Q$ . Constraints: $0 \leq k \leq m$ , if <code>side = 'L'</code> ; $0 \leq k \leq n$ , if <code>side = 'R'</code> .
<code>l</code>	INTEGER.

The number of columns of the matrix  $A$  containing the meaningful part of the Householder reflectors.

Constraints:

$0 \leq l \leq m$ , if  $\text{side} = 'L'$ ;

$0 \leq l \leq n$ , if  $\text{side} = 'R'$ .

$a, tau, c, work$  COMPLEX for `cunmrz`

DOUBLE COMPLEX for `zunmrz`.

Arrays:  $a(lda, *)$ ,  $tau(*)$ ,  $c(ldc, *)$ ,  
 $work(lwork)$ .

On entry, the  $i$ th row of  $a$  must contain the vector which defines the elementary reflector  $H_i$ , for  $i = 1, 2, \dots, k$ , as returned by `ctzrzf/ztzrzf` in the last  $k$  rows of its array argument  $a$ .

The second dimension of  $a$  must be at least  $\max(1, m)$  if  $\text{side} = 'L'$ , and at least  $\max(1, n)$  if  $\text{side} = 'R'$ .

$tau(i)$  must contain the scalar factor of the elementary reflector  $H_i$ , as returned by `ctzrzf/ztzrzf`.

The dimension of  $tau$  must be at least  $\max(1, k)$ .

$c(ldc, *)$  contains the  $m$ -by- $n$  matrix  $C$ .

The second dimension of  $c$  must be at least  $\max(1, n)$   
 $work(lwork)$  is a workspace array.

$lda$  INTEGER. The first dimension of  $a$ ;  $lda \geq \max(1, k)$ .

$ldc$  INTEGER. The first dimension of  $c$ ;  $ldc \geq \max(1, m)$ .

$lwork$  INTEGER. The size of the  $work$  array. Constraints:

$lwork \geq \max(1, n)$  if  $\text{side} = 'L'$ ;

$lwork \geq \max(1, m)$  if  $\text{side} = 'R'$ .

See *Application notes* for the suggested value of  $lwork$ .

## Output Parameters

$c$  Overwritten by the product  $QC$ ,  $Q^H C$ ,  $CQ$ , or  $CQ^H$  (as specified by  $\text{side}$  and  $\text{trans}$ ).

$work(1)$  If  $\text{info} = 0$ , on exit  $work(1)$  contains the minimum value of  $lwork$  required for optimum performance. Use this  $lwork$  for subsequent runs.

*info*

INTEGER.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*th parameter had an illegal value.

### Application Notes

For better performance, try using *lwork* = *n*\**blocksize* (if *side* = 'L') or *lwork* = *m*\**blocksize* (if *side* = 'R') where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*. If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run. On exit, examine *work*(1) and use this value for subsequent runs.

The real counterpart of this routine is [?ormrz](#).

## ?ggqrft

*Computes the generalized QR factorization of two matrices.*

---

```
call sggqrft (n, m, p, a, lda, taua, b, ldb, taub, work, lwork, info)
call dggqrft (n, m, p, a, lda, taua, b, ldb, taub, work, lwork, info)
call cggqrft (n, m, p, a, lda, taua, b, ldb, taub, work, lwork, info)
call zggqrft (n, m, p, a, lda, taua, b, ldb, taub, work, lwork, info)
```

### Discussion

The routine forms the generalized *QR* factorization of an *n*-by-*m* matrix *A* and an *n*-by-*p* matrix *B* as  $A = Q R$ ,  $B = Q T Z$ , where *Q* is an *n*-by-*n* orthogonal/unitary matrix, *Z* is a *p*-by-*p* orthogonal/unitary matrix, and *R* and *T* assume one of the forms:

$$R = \begin{matrix} & m \\ \begin{matrix} n-m & \end{matrix} & \begin{pmatrix} R_{11} \\ 0 \end{pmatrix} \end{matrix}, \quad \text{if } n \geq m$$

or

$$R = \begin{matrix} n & m-n \\ n & \begin{pmatrix} R_{11} & R_{12} \end{pmatrix} \end{matrix}, \quad \text{if } n < m,$$

where *R*<sub>11</sub> is upper triangular, and

$$T = \begin{matrix} p-n & n \\ n & \begin{pmatrix} 0 & T_{12} \end{pmatrix} \end{matrix}, \quad \text{if } n \leq p, \text{ or}$$

$$T = \begin{matrix} p \\ n-p & \begin{pmatrix} T_{11} \\ T_{21} \end{pmatrix} \end{matrix}, \quad \text{if } n > p$$

where *T*<sub>12</sub> or *T*<sub>21</sub> is a *p*-by-*p* upper triangular matrix.

In particular, if  $B$  is square and nonsingular, the  $GQR$  factorization of  $A$  and  $B$  implicitly gives the  $QR$  factorization of  $B^{-1}A$  as:

$$B^{-1}A = Z^H(T^{-1}R)$$

### Input Parameters

$n$	<code>INTEGER</code> . The number of rows of the matrices $A$ and $B$ ( $n \geq 0$ ).
$m$	<code>INTEGER</code> . The number of columns in $A$ ( $m \geq 0$ ).
$p$	<code>INTEGER</code> . The number of columns in $B$ ( $p \geq 0$ ).
$a, b, work$	<code>REAL</code> for <code>sggqr</code> <code>DOUBLE PRECISION</code> for <code>dggqr</code> <code>COMPLEX</code> for <code>cggqr</code> <code>DOUBLE COMPLEX</code> for <code>zggqr</code> . Arrays: $a(1..n, 1..m)$ contains the matrix $A$ . The second dimension of $a$ must be at least $\max(1, m)$ . $b(1..n, 1..p)$ contains the matrix $B$ . The second dimension of $b$ must be at least $\max(1, p)$ . $work(1..lwork)$ is a workspace array. $lda$ <code>INTEGER</code> . The first dimension of $a$ ; at least $\max(1, n)$ . $ldb$ <code>INTEGER</code> . The first dimension of $b$ ; at least $\max(1, n)$ . $lwork$ <code>INTEGER</code> . The size of the $work$ array; must be at least $\max(1, n, m, p)$ See <i>Application notes</i> for the suggested value of $lwork$ .

### Output Parameters

$a, b$	Overwritten by the factorization data as follows:  on exit, the elements on and above the diagonal of the array $a$ contain the $\min(n, m)$ -by- $m$ upper trapezoidal matrix $R$ ( $R$ is upper triangular if $n \geq m$ ); the elements below the diagonal, with the array $tau_a$ , represent the orthogonal/unitary matrix $Q$ as a product of $\min(n, m)$ elementary reflectors ;
--------	--

if  $n \leq p$ , the upper triangle of the subarray  
 $b(1:n, p-n+1:p)$  contains the  $n$ -by- $n$  upper triangular matrix  $T$ ;  
 if  $n > p$ , the elements on and above the  $(n-p)$ th subdiagonal contain the  $n$ -by- $p$  upper trapezoidal matrix  $T$ ; the remaining elements, with the array `taub`, represent the orthogonal/unitary matrix  $Z$  as a product of elementary reflectors.

`tauua, taub``REAL` for `sggqr``DOUBLE PRECISION` for `dggqr``COMPLEX` for `cggqr``DOUBLE COMPLEX` for `zggqr`.

Arrays, `DIMENSION` at least  $\max(1, \min(n, m))$  for `tauua` and at least  $\max(1, \min(n, p))$  for `taub`.

The array `tauua` contains the scalar factors of the elementary reflectors which represent the orthogonal/unitary matrix  $Q$ .

The array `taub` contains the scalar factors of the elementary reflectors which represent the orthogonal/unitary matrix  $Z$ .

`work(1)`

If `info = 0`, on exit `work(1)` contains the minimum value of `lwork` required for optimum performance. Use this `lwork` for subsequent runs.

`info``INTEGER`.

If `info = 0`, the execution is successful.

If `info = -i`, the  $i$ th parameter had an illegal value.

## Application Notes

For better performance, try using

`lwork`  $\geq \max(n, m, p) * \max(nb1, nb2, nb3)$ ,

where  $nb1$  is the optimal blocksize for the  $QR$  factorization of an  $n$ -by- $m$  matrix,  $nb2$  is the optimal blocksize for the  $RQ$  factorization of an  $n$ -by- $p$  matrix, and  $nb3$  is the optimal blocksize for a call of `?ormqr / ?unmqr`.

## ?gqrqf

*Computes the generalized RQ factorization of two matrices.*

```
call sggrqf (m, p, n, a, lda, taua, b, ldb, taub, work, lwork, info)
call dggrqf (m, p, n, a, lda, taua, b, ldb, taub, work, lwork, info)
call cggrqf (m, p, n, a, lda, taua, b, ldb, taub, work, lwork, info)
call zggrqf (m, p, n, a, lda, taua, b, ldb, taub, work, lwork, info)
```

### Discussion

The routine forms the generalized  $RQ$  factorization of an  $m$ -by- $n$  matrix  $A$  and an  $p$ -by- $n$  matrix  $B$  as  $A = R Q$ ,  $B = Z T Q$ , where  $Q$  is an  $n$ -by- $n$  orthogonal/unitary matrix,  $Z$  is a  $p$ -by- $p$  orthogonal/unitary matrix, and  $R$  and  $T$  assume one of the forms:

$$R = \begin{matrix} n-m & m \\ m & (0 & R_{12}) \end{matrix}, \quad \text{if } m \leq n,$$

or

$$R = \begin{matrix} n \\ m-n \\ n \end{matrix} \begin{pmatrix} R_{11} \\ R_{21} \end{pmatrix}, \quad \text{if } m > n$$

where  $R_{11}$  or  $R_{21}$  is upper triangular, and

$$T = \begin{matrix} n \\ p-n \\ 0 \end{matrix} \begin{pmatrix} T_{11} \\ T_{12} \end{pmatrix}, \quad \text{if } p \geq n$$

or

$$T = \begin{matrix} p & n-p \\ p & (T_{11} & T_{12}) \end{matrix}, \quad \text{if } p < n,$$

where  $T_{11}$  is upper triangular.

In particular, if  $B$  is square and nonsingular, the  $GRQ$  factorization of  $A$  and  $B$  implicitly gives the  $RQ$  factorization of  $AB^{-1}$  as:

$$AB^{-1} = (R \ T^{-1}) Z^H$$

### Input Parameters

*m*                    INTEGER. The number of rows of the matrix  $A$  ( $m \geq 0$ ).

*p*                    INTEGER. The number of rows in  $B$  ( $p \geq 0$ ).

*n*                    INTEGER. The number of columns of the matrices  $A$  and  $B$  ( $n \geq 0$ ).

*a, b, work*        REAL for `sggrqf`

DOUBLE PRECISION for `dgrqf`

COMPLEX for `cggrqf`

DOUBLE COMPLEX for `zggrqf`.

Arrays:

*a( lda, \* )* contains the  $m$ -by- $n$  matrix  $A$ .

The second dimension of *a* must be at least  $\max(1, n)$ .

*b( ldb, \* )* contains the  $p$ -by- $n$  matrix  $B$ .

The second dimension of *b* must be at least  $\max(1, n)$ .

*work( lwork )* is a workspace array.

*lda*                  INTEGER. The first dimension of *a*; at least  $\max(1, m)$ .

*ldb*                  INTEGER. The first dimension of *b*; at least  $\max(1, p)$ .

*lwork*                INTEGER. The size of the *work* array; must be at least  $\max(1, n, m, p)$

See *Application notes* for the suggested value of *lwork*.

### Output Parameters

*a, b*                Overwritten by the factorization data as follows:

on exit, if  $m \leq n$ , the upper triangle of the subarray  
 $a(1:m, n-m+1:n)$  contains the  $m$ -by- $m$  upper triangular  
matrix  $R$ ;

if  $m > n$ , the elements on and above the  $(m-n)$ th  
subdiagonal contain the  $m$ -by- $n$  upper trapezoidal

matrix  $R$ ; the remaining elements, with the array `taua`, represent the orthogonal/unitary matrix  $Q$  as a product of elementary reflectors; the elements on and above the diagonal of the array `b` contain the  $\min(p,n)$ -by- $n$  upper trapezoidal matrix  $T$  ( $T$  is upper triangular if  $p \geq n$ ); the elements below the diagonal, with the array `taub`, represent the orthogonal/unitary matrix  $Z$  as a product of elementary reflectors.

<code>taua, taub</code>	<code>REAL</code> for <code>sggrqf</code> <code>DOUBLE PRECISION</code> for <code>dgrqf</code> <code>COMPLEX</code> for <code>cggrqf</code> <code>DOUBLE COMPLEX</code> for <code>zggrqf</code> .
	Arrays, <code>DIMENSION</code> at least $\max(1, \min(m, n))$ for <code>taua</code> and at least $\max(1, \min(p, n))$ for <code>taub</code> .
	The array <code>taua</code> contains the scalar factors of the elementary reflectors which represent the orthogonal/unitary matrix $Q$ .
	The array <code>taub</code> contains the scalar factors of the elementary reflectors which represent the orthogonal/unitary matrix $Z$ .
<code>work(1)</code>	If <code>info = 0</code> , on exit <code>work(1)</code> contains the minimum value of <code>lwork</code> required for optimum performance. Use this <code>lwork</code> for subsequent runs.
<code>info</code>	<code>INTEGER</code> . If <code>info = 0</code> , the execution is successful. If <code>info = -i</code> , the $i$ th parameter had an illegal value.

## Application Notes

For better performance, try using

$$lwork \geq \max(n, m, p) * \max(nb1, nb2, nb3),$$

where  $nb1$  is the optimal blocksize for the  $RQ$  factorization of an  $m$ -by- $n$  matrix,  $nb2$  is the optimal blocksize for the  $QR$  factorization of an  $p$ -by- $n$  matrix, and  $nb3$  is the optimal blocksize for a call of `?ormrq/?unmrq`.

If you are in doubt how much workspace to supply, use a generous value of `lwork` for the first run. On exit, examine `work(1)` and use this value for subsequent runs.

## Singular Value Decomposition

This section describes LAPACK routines for computing the *singular value decomposition* (SVD) of a general  $m$  by  $n$  matrix  $A$ :

$$A = U\Sigma V^H.$$

In this decomposition,  $U$  and  $V$  are unitary (for complex  $A$ ) or orthogonal (for real  $A$ );  $\Sigma$  is an  $m$  by  $n$  diagonal matrix with real diagonal elements  $\sigma_i$ :

$$\sigma_1 \geq \sigma_2 \geq \dots \geq \sigma_{\min(m, n)} \geq 0.$$

The diagonal elements  $\sigma_i$  are *singular values* of  $A$ . The first  $\min(m, n)$  columns of the matrices  $U$  and  $V$  are, respectively, *left* and *right singular vectors* of  $A$ . The singular values and singular vectors satisfy

$$Av_i = \sigma_i u_i \text{ and } A^H v_i = \sigma_i v_i$$

where  $u_i$  and  $v_i$  are the  $i$ th columns of  $U$  and  $V$ , respectively.

To find the SVD of a general matrix  $A$ , call the LAPACK routine [?gebrd](#) or [?gbbrd](#) for reducing  $A$  to a bidiagonal matrix  $B$  by a unitary (orthogonal) transformation:  $A = QBP^H$ . Then call [?bdsqr](#), which forms the SVD of a bidiagonal matrix:  $B = U_1 \Sigma V_1^H$ .

Thus, the sought-for SVD of  $A$  is given by  $A = U\Sigma V^H = (QU_1)\Sigma(V_1^HP^H)$ .

**Table 5-2 Computational Routines for Singular Value Decomposition (SVD)**

Operation	Real matrices	Complex matrices
Reduce $A$ to a bidiagonal matrix $B$ : $A = QBP^H$ (full storage)	<a href="#">?gebrd</a>	<a href="#">?gebrd</a>
Reduce $A$ to a bidiagonal matrix $B$ : $A = QBP^H$ (band storage)	<a href="#">?gbbrd</a>	<a href="#">?gbbrd</a>
Generate the orthogonal (unitary) matrix $Q$ or $P$	<a href="#">?orgbr</a>	<a href="#">?ungbr</a>
Apply the orthogonal (unitary) matrix $Q$ or $P$	<a href="#">?ormbr</a>	<a href="#">?unmbr</a>
Form singular value decomposition of the bidiagonal matrix $B$ : $B = U\Sigma V^H$	<a href="#">?bdsqr</a> <a href="#">?bdsdc</a>	<a href="#">?bdsqr</a>

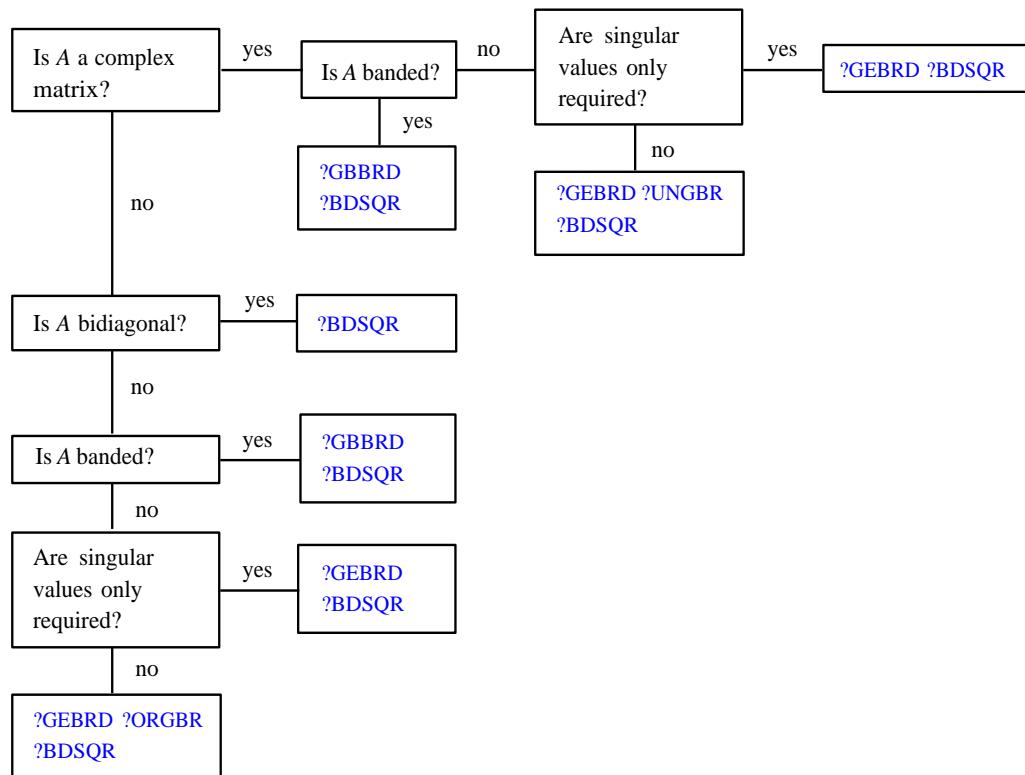
**Figure 5-1 Decision Tree: Singular Value Decomposition**

Figure 5-1 presents a decision tree that helps you choose the right sequence of routines for SVD, depending on whether you need singular values only or singular vectors as well, whether  $A$  is real or complex, and so on.

You can use the SVD to find a minimum-norm solution to a (possibly) rank-deficient least-squares problem of minimizing  $\|Ax - b\|_2$ . The effective rank  $k$  of the matrix  $A$  can be determined as the number of singular values which exceed a suitable threshold. The minimum-norm solution is

$$x = V_k(\Sigma_k)^{-1}c$$

where  $\Sigma_k$  is the leading  $k$  by  $k$  submatrix of  $\Sigma$ , the matrix  $V_k$  consists of the first  $k$  columns of  $V = PV_1$ , and the vector  $c$  consists of the first  $k$  elements of  $U^H b = U_1^H Q^H b$ .

## ?gebrd

*Reduces a general matrix to bidiagonal form.*

---

```
call sgebrd ( m, n, a, lda, d, e, tauq, taup, work, lwork, info )
call dgebrd ( m, n, a, lda, d, e, tauq, taup, work, lwork, info )
call cgebrd ( m, n, a, lda, d, e, tauq, taup, work, lwork, info )
call zgebrd ( m, n, a, lda, d, e, tauq, taup, work, lwork, info )
```

### Discussion

The routine reduces a general  $m$  by  $n$  matrix  $A$  to a bidiagonal matrix  $B$  by an orthogonal (unitary) transformation.

If  $m \geq n$ , the reduction is given by

$$A = QBP^H = Q \begin{pmatrix} B_1 \\ 0 \end{pmatrix} P^H = Q_1 B_1 P^H,$$

where  $B_1$  is an  $n$  by  $n$  upper diagonal matrix,  $Q$  and  $P$  are orthogonal or, for a complex  $A$ , unitary matrices;  $Q_1$  consists of the first  $n$  columns of  $Q$ .

If  $m < n$ , the reduction is given by

$$A = QBP^H = Q(B_1 0)P^H = Q_1 B_1 P_1^H,$$

where  $B_1$  is an  $m$  by  $m$  lower diagonal matrix,  $Q$  and  $P$  are orthogonal or, for a complex  $A$ , unitary matrices;  $P_1$  consists of the first  $m$  rows of  $P$ .

The routine does not form the matrices  $Q$  and  $P$  explicitly, but represents them as products of elementary reflectors. Routines are provided to work with the matrices  $Q$  and  $P$  in this representation:

If the matrix  $A$  is real,

- to compute  $Q$  and  $P$  explicitly, call [?orgbr](#).
- to multiply a general matrix by  $Q$  or  $P$ , call [?ormbr](#).

If the matrix  $A$  is complex,

- to compute  $Q$  and  $P$  explicitly, call [?ungbr](#).
- to multiply a general matrix by  $Q$  or  $P$ , call [?unmbr](#).

## Input Parameters

<i>m</i>	INTEGER. The number of rows in the matrix <i>A</i> ( <i>m</i> ≥ 0).
<i>n</i>	INTEGER. The number of columns in <i>A</i> ( <i>n</i> ≥ 0).
<i>a</i> , <i>work</i>	REAL for <i>sgebrd</i> DOUBLE PRECISION for <i>dgebrd</i> COMPLEX for <i>cgebrd</i> DOUBLE COMPLEX for <i>zgebrd</i> .
	Arrays: <i>a</i> ( <i>lda</i> , *) contains the matrix <i>A</i> . The second dimension of <i>a</i> must be at least $\max(1, n)$ . <i>work</i> ( <i>lwork</i> ) is a workspace array.
<i>lda</i>	INTEGER. The first dimension of <i>a</i> ; at least $\max(1, m)$ .
<i>lwork</i>	INTEGER. The dimension of <i>work</i> ; at least $\max(1, m, n)$ . See <i>Application notes</i> for the suggested value of <i>lwork</i> .

## Output Parameters

<i>a</i>	If <i>m</i> ≥ <i>n</i> , the diagonal and first super-diagonal of <i>a</i> are overwritten by the upper bidiagonal matrix <i>B</i> . Elements below the diagonal are overwritten by details of <i>Q</i> , and the remaining elements are overwritten by details of <i>P</i> . If <i>m</i> < <i>n</i> , the diagonal and first sub-diagonal of <i>a</i> are overwritten by the lower bidiagonal matrix <i>B</i> . Elements above the diagonal are overwritten by details of <i>P</i> , and the remaining elements are overwritten by details of <i>Q</i> .
<i>d</i>	REAL for single-precision flavors DOUBLE PRECISION for double-precision flavors. Array, DIMENSION at least $\max(1, \min(m, n))$ . Contains the diagonal elements of <i>B</i> .
<i>e</i>	REAL for single-precision flavors DOUBLE PRECISION for double-precision flavors. Array, DIMENSION at least $\max(1, \min(m, n) - 1)$ . Contains the off-diagonal elements of <i>B</i> .

<i>taug, taup</i>	<small>REAL for <code>sgebrd</code> DOUBLE PRECISION for <code>dgebrd</code> COMPLEX for <code>cgebrd</code> DOUBLE COMPLEX for <code>zgebrd</code>.</small>
	<small>Arrays, DIMENSION at least max (1, min(<i>m, n</i>)). Contain further details of the matrices <i>Q</i> and <i>P</i>.</small>
<i>work(1)</i>	<small>If <i>info</i> = 0, on exit <i>work(1)</i> contains the minimum value of <i>lwork</i> required for optimum performance. Use this <i>lwork</i> for subsequent runs.</small>
<i>info</i>	<small>INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = -<i>i</i>, the <i>i</i>th parameter had an illegal value.</small>

### Application Notes

For better performance, try using  $lwork = (m + n) * blocksize$ , where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run. On exit, examine *work(1)* and use this value for subsequent runs.

The computed matrices *Q*, *B*, and *P* satisfy  $QB^H = A + E$ , where  $\|E\|_2 = c(n)\epsilon \|A\|_2$ ,  $c(n)$  is a modestly increasing function of *n*, and  $\epsilon$  is the machine precision.

The approximate number of floating-point operations for real flavors is  
 $(4/3) * n^2 * (3 * m - n)$  for  $m \geq n$ ,  
 $(4/3) * m^2 * (3 * n - m)$  for  $m < n$ .

The number of operations for complex flavors is four times greater.

If *n* is much less than *m*, it can be more efficient to first form the *QR* factorization of *A* by calling `?geqrf` and then reduce the factor *R* to bidiagonal form. This requires approximately  $2 * n^2 * (m + n)$  floating-point operations.

If *m* is much less than *n*, it can be more efficient to first form the *LQ* factorization of *A* by calling `?gelqf` and then reduce the factor *L* to bidiagonal form. This requires approximately  $2 * m^2 * (m + n)$  floating-point operations.

## ?gbbrd

Reduces a general band matrix to bidiagonal form.

```
call sgbb rd ( vect, m, n, ncc, kl, ku, ab, ldab, d, e, q, ldq, pt,
    ldpt, c, ldc, work, info )
call dgbb rd ( vect, m, n, ncc, kl, ku, ab, ldab, d, e, q, ldq, pt,
    ldpt, c, ldc, work, info )
call cgbb rd ( vect, m, n, ncc, kl, ku, ab, ldab, d, e, q, ldq, pt,
    ldpt, c, ldc, work, rwork, info )
call zgbb rd ( vect, m, n, ncc, kl, ku, ab, ldab, d, e, q, ldq, pt,
    ldpt, c, ldc, work, rwork, info )
```

### Discussion

This routine reduces an  $m$  by  $n$  band matrix  $A$  to upper bidiagonal matrix  $B$ :  $A = QBP^H$ . Here the matrices  $Q$  and  $P$  are orthogonal (for real  $A$ ) or unitary (for complex  $A$ ). They are determined as products of Givens rotation matrices, and may be formed explicitly by the routine if required. The routine can also update a matrix  $C$  as follows:  $C = Q^H C$ .

### Input Parameters

<code>vect</code>	CHARACTER*1. Must be ' <code>N</code> ' or ' <code>Q</code> ' or ' <code>P</code> ' or ' <code>B</code> '. If <code>vect</code> = ' <code>N</code> ', neither $Q$ nor $P^H$ is generated. If <code>vect</code> = ' <code>Q</code> ', the routine generates the matrix $Q$ . If <code>vect</code> = ' <code>P</code> ', the routine generates the matrix $P^H$ . If <code>vect</code> = ' <code>B</code> ', the routine generates both $Q$ and $P^H$ .
<code>m</code>	INTEGER. The number of rows in the matrix $A$ ( $m \geq 0$ ).
<code>n</code>	INTEGER. The number of columns in $A$ ( $n \geq 0$ ).
<code>ncc</code>	INTEGER. The number of columns in $C$ ( $ncc \geq 0$ ).
<code>kl</code>	INTEGER. The number of sub-diagonals within the band of $A$ ( $kl \geq 0$ ).
<code>ku</code>	INTEGER. The number of super-diagonals within the band of $A$ ( $ku \geq 0$ ).

<i>ab, c, work</i>	<b>REAL</b> for <b>sgbbrd</b> <b>DOUBLE PRECISION</b> for <b>dgbbrd</b> <b>COMPLEX</b> for <b>cgbbrd</b> <b>DOUBLE COMPLEX</b> for <b>zgbbrd</b> . Arrays: <i>ab( ldat, * )</i> contains the matrix <i>A</i> in band storage (see <a href="#">Matrix Storage Schemes</a> ). The second dimension of <i>a</i> must be at least $\max(1, n)$ . <i>c( ldc, * )</i> contains an <i>m</i> by <i>ncc</i> matrix <i>C</i> . If <i>ncc</i> = 0, the array <i>c</i> is not referenced. The second dimension of <i>c</i> must be at least $\max(1, ncc)$ . <i>work( * )</i> is a workspace array. The dimension of <i>work</i> must be at least $2 * \max(m, n)$ for real flavors, or $\max(m, n)$ for complex flavors.
<i>ldat</i>	<b>INTEGER</b> . The first dimension of the array <i>ab</i> ( <i>ldat</i> $\geq k_l + k_u + 1$ ).
<i>ldq</i>	<b>INTEGER</b> . The first dimension of the output array <i>q</i> . <i>ldq</i> $\geq \max(1, m)$ if <i>vect</i> = 'Q' or 'B', <i>ldq</i> $\geq 1$ otherwise.
<i>ldpt</i>	<b>INTEGER</b> . The first dimension of the output array <i>pt</i> . <i>ldpt</i> $\geq \max(1, n)$ if <i>vect</i> = 'P' or 'B', <i>ldpt</i> $\geq 1$ otherwise.
<i>ldc</i>	<b>INTEGER</b> . The first dimension of the array <i>c</i> . <i>ldc</i> $\geq \max(1, m)$ if <i>ncc</i> > 0; <i>ldc</i> $\geq 1$ if <i>ncc</i> = 0.
<i>rwork</i>	<b>REAL</b> for <b>cgbbrd</b> <b>DOUBLE PRECISION</b> for <b>zgbbrd</b> . A workspace array, <b>DIMENSION</b> at least $\max(m, n)$ .

## Output Parameters

<i>ab</i>	Overwritten by values generated during the reduction.
<i>d</i>	<b>REAL</b> for single-precision flavors <b>DOUBLE PRECISION</b> for double-precision flavors. Array, <b>DIMENSION</b> at least $\max(1, \min(m, n))$ . Contains the diagonal elements of the matrix <i>B</i> .

---

<i>e</i>	<code>REAL</code> for single-precision flavors <code>DOUBLE PRECISION</code> for double-precision flavors. Array, <code>DIMENSION</code> at least $\max(1, \min(m, n) - 1)$ . Contains the off-diagonal elements of $B$ .
<i>q, pt</i>	<code>REAL</code> for <code>sgebrd</code> <code>DOUBLE PRECISION</code> for <code>dgebrd</code> <code>COMPLEX</code> for <code>cgebrd</code> <code>DOUBLE COMPLEX</code> for <code>zgebrd</code> . Arrays:  <i>q</i> ( <i>ldq</i> , *) contains the output <i>m</i> by <i>m</i> matrix $Q$ . The second dimension of <i>q</i> must be at least $\max(1, m)$ .  <i>p</i> ( <i>ldpt</i> , *) contains the output <i>n</i> by <i>n</i> matrix $P^H$ . The second dimension of <i>pt</i> must be at least $\max(1, n)$ .
<i>info</i>	<code>INTEGER</code> . If <i>info</i> = 0, the execution is successful. If <i>info</i> = <i>-i</i> , the <i>i</i> th parameter had an illegal value.

### Application Notes

The computed matrices  $Q$ ,  $B$ , and  $P$  satisfy  $QBP^H = A + E$ , where  $\|E\|_2 = c(n)\epsilon \|A\|_2$ ,  $c(n)$  is a modestly increasing function of  $n$ , and  $\epsilon$  is the machine precision.

If  $m = n$ , the total number of floating-point operations for real flavors is approximately the sum of:

$$\begin{aligned} 6 * n^2 * (k_l + k_u) & \quad \text{if } \text{vect} = 'N' \text{ and } ncc = 0, \\ 3 * n^2 * ncc * (k_l + k_u - 1) / (k_l + k_u) & \quad \text{if } C \text{ is updated, and} \\ 3 * n^3 * (k_l + k_u - 1) / (k_l + k_u) & \quad \text{if either } Q \text{ or } P^H \text{ is generated} \\ & \quad (\text{double this if both}). \end{aligned}$$

To estimate the number of operations for complex flavors, use the same formulas with the coefficients 20 and 10 (instead of 6 and 3).

---

## ?orgbr

*Generates the real orthogonal matrix  $Q$  or  $P^T$  determined by ?gebrd.*

---

```
call sorgbr ( vect, m, n, k, a, lda, tau, work, lwork, info )
call dorgbr ( vect, m, n, k, a, lda, tau, work, lwork, info )
```

### Discussion

The routine generates the whole or part of the orthogonal matrices  $Q$  and  $P^T$  formed by the routines `sgebrd/dgebrd` (see [page 5-76](#)). Use this routine after a call to `sgebrd/dgebrd`. All valid combinations of arguments are described in *Input parameters*. In most cases you'll need the following:

To compute the whole  $m$  by  $m$  matrix  $Q$ :

```
call ?orgbr ( 'Q', m, m, n, a ... )
(note that the array a must have at least m columns).
```

To form the  $n$  leading columns of  $Q$  if  $m > n$ :

```
call ?orgbr ( 'Q', m, n, n, a ... )
```

To compute the whole  $n$  by  $n$  matrix  $P^T$ :

```
call ?orgbr ( 'P', n, n, m, a ... )
(note that the array a must have at least n rows).
```

To form the  $m$  leading rows of  $P^T$  if  $m < n$ :

```
call ?orgbr ( 'P', m, n, m, a ... )
```

### Input Parameters

<code>vect</code>	CHARACTER*1. Must be ' <code>Q</code> ' or ' <code>P</code> '. If <code>vect = 'Q'</code> , the routine generates the matrix $Q$ . If <code>vect = 'P'</code> , the routine generates the matrix $P^T$ .
<code>m</code>	INTEGER. The number of required rows of $Q$ or $P^T$ .
<code>n</code>	INTEGER. The number of required columns of $Q$ or $P^T$ .
<code>k</code>	INTEGER. One of the dimensions of $A$ in ?gebrd: If <code>vect = 'Q'</code> , the number of columns in $A$ ; If <code>vect = 'P'</code> , the number of rows in $A$ .

Constraints:  $m \geq 0$ ,  $n \geq 0$ ,  $k \geq 0$ .

For  $\text{vect} = 'Q'$ :  $k \leq n \leq m$  if  $m > k$ , or  $m = n$  if  $m \leq k$ .

For  $\text{vect} = 'P'$ :  $k \leq m \leq n$  if  $n > k$ , or  $m = n$  if  $n \leq k$ .

$a$ , $\text{work}$	REAL for <code>sorgbr</code> DOUBLE PRECISION for <code>dorgbr</code> . Arrays: $a(1:\text{lda}, :)$ is the array $a$ as returned by <code>?gebrd</code> . The second dimension of $a$ must be at least $\max(1, n)$ . $\text{work}(1:\text{lwork})$ is a workspace array.
$\text{lda}$	INTEGER. The first dimension of $a$ ; at least $\max(1, m)$ .
$\tau$	REAL for <code>sorgbr</code> DOUBLE PRECISION for <code>dorgbr</code> . For $\text{vect} = 'Q'$ , the array $\tau_{\text{aq}}$ as returned by <code>?gebrd</code> . For $\text{vect} = 'P'$ , the array $\tau_{\text{ap}}$ as returned by <code>?gebrd</code> . The dimension of $\tau$ must be at least $\max(1, \min(m, k))$ for $\text{vect} = 'Q'$ , or $\max(1, \min(m, k))$ for $\text{vect} = 'P'$ .
$\text{lwork}$	INTEGER. The size of the $\text{work}$ array. See <i>Application notes</i> for the suggested value of $\text{lwork}$ .

## Output Parameters

$a$	Overwritten by the orthogonal matrix $Q$ or $P^T$ (or the leading rows or columns thereof) as specified by $\text{vect}$ , $m$ , and $n$ .
$\text{work}(1)$	If $\text{info} = 0$ , on exit $\text{work}(1)$ contains the minimum value of $\text{lwork}$ required for optimum performance. Use this $\text{lwork}$ for subsequent runs.
$\text{info}$	INTEGER. If $\text{info} = 0$ , the execution is successful. If $\text{info} = -i$ , the $i$ th parameter had an illegal value.

## Application Notes

For better performance, try using  $\text{lwork} = \min(m, n) * \text{blocksize}$ , where  $\text{blocksize}$  is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use a generous value of  $\text{lwork}$  for the first run. On exit, examine  $\text{work}(1)$  and use this value for subsequent runs.

The computed matrix  $Q$  differs from an exactly orthogonal matrix by a matrix  $E$  such that  $\|E\|_2 = O(\epsilon)$ .

The approximate numbers of floating-point operations for the cases listed in *Discussion* are as follows:

To form the whole of  $Q$ :

$$\begin{aligned} (4/3)n(3m^2 - 3m*n + n^2) & \quad \text{if } m > n; \\ (4/3)m^3 & \quad \text{if } m \leq n. \end{aligned}$$

To form the  $n$  leading columns of  $Q$  when  $m > n$ :

$$(2/3)n^2(3m - n^2) \quad \text{if } m > n.$$

To form the whole of  $P^T$ :

$$\begin{aligned} (4/3)n^3 & \quad \text{if } m \geq n; \\ (4/3)m(3n^2 - 3m*n + m^2) & \quad \text{if } m < n. \end{aligned}$$

To form the  $m$  leading columns of  $P^T$  when  $m < n$ :

$$(2/3)n^2(3m - n^2) \quad \text{if } m > n.$$

The complex counterpart of this routine is [?ungbr](#).

## ?ormbr

*Multiples an arbitrary real matrix by  
the real orthogonal matrix  $Q$  or  $P^T$   
determined by ?gebrd.*

```
call sormbr (vect,side,trans,m,n,k,a,lda,tau,c,ldc,work,lwork,info)
call dormbr (vect,side,trans,m,n,k,a,lda,tau,c,ldc,work,lwork,info)
```

### Discussion

Given an arbitrary real matrix  $C$ , this routine forms one of the matrix products  $QC$ ,  $Q^TC$ ,  $CQ$ ,  $CQ^T$ ,  $PC$ ,  $P^TC$ ,  $CP$ , or  $CP^T$ , where  $Q$  and  $P$  are orthogonal matrices computed by a call to `sgebrd/dgebrd` (see [page 5-76](#)). The routine overwrites the product on  $C$ .

### Input Parameters

In the descriptions below,  $r$  denotes the order of  $Q$  or  $P^T$ :

If  $\text{side} = \text{'L'}$ ,  $r = m$ ; if  $\text{side} = \text{'R'}$ ,  $r = n$ .

<code>vect</code>	CHARACTER*1. Must be ' <code>Q</code> ' or ' <code>P</code> '. If <code>vect = 'Q'</code> , then $Q$ or $Q^T$ is applied to $C$ . If <code>vect = 'P'</code> , then $P$ or $P^T$ is applied to $C$ .
<code>side</code>	CHARACTER*1. Must be ' <code>L</code> ' or ' <code>R</code> '. If <code>side = 'L'</code> , multipliers are applied to $C$ from the left. If <code>side = 'R'</code> , they are applied to $C$ from the right.
<code>trans</code>	CHARACTER*1. Must be ' <code>N</code> ' or ' <code>T</code> '. If <code>trans = 'N'</code> , then $Q$ or $P$ is applied to $C$ . If <code>trans = 'T'</code> , then $Q^T$ or $P^T$ is applied to $C$ .
<code>m</code>	INTEGER. The number of rows in $C$ .
<code>n</code>	INTEGER. The number of columns in $C$ .
<code>k</code>	INTEGER. One of the dimensions of $A$ in ?gebrd: If <code>vect = 'Q'</code> , the number of columns in $A$ ; If <code>vect = 'P'</code> , the number of rows in $A$ .
Constraints: $m \geq 0$ , $n \geq 0$ , $k \geq 0$ .	

<i>a, c, work</i>	<b>REAL</b> for <b>sormbr</b> <b>DOUBLE PRECISION</b> for <b>dormbr</b> . Arrays: <i>a( lda, * )</i> is the array <i>a</i> as returned by <b>?gebrd</b> . Its second dimension must be at least $\max(1, \min(r, k))$ for <i>vect</i> = 'Q', or $\max(1, r)$ for <i>vect</i> = 'P'. <i>c( ldc, * )</i> holds the matrix <i>C</i> . Its second dimension must be at least $\max(1, n)$ . <i>work( lwork )</i> is a workspace array.
<i>lda</i>	<b>INTEGER</b> . The first dimension of <i>a</i> . Constraints: <i>lda</i> $\geq \max(1, r)$ if <i>vect</i> = 'Q'; <i>lda</i> $\geq \max(1, \min(r, k))$ if <i>vect</i> = 'P'.
<i>ldc</i>	<b>INTEGER</b> . The first dimension of <i>c</i> ; <i>ldc</i> $\geq \max(1, m)$ .
<i>tau</i>	<b>REAL</b> for <b>sormbr</b> <b>DOUBLE PRECISION</b> for <b>dormbr</b> . Array, <b>DIMENSION</b> at least $\max(1, \min(r, k))$ . For <i>vect</i> = 'Q', the array <i>taug</i> as returned by <b>?gebrd</b> . For <i>vect</i> = 'P', the array <i>taup</i> as returned by <b>?gebrd</b> .
<i>lwork</i>	<b>INTEGER</b> . The size of the <i>work</i> array. Constraints: <i>lwork</i> $\geq \max(1, n)$ if <i>side</i> = 'L'; <i>lwork</i> $\geq \max(1, m)$ if <i>side</i> = 'R'. See <i>Application notes</i> for the suggested value of <i>lwork</i> .

## Output Parameters

<i>c</i>	Overwritten by the product $QC$ , $Q^TC$ , $CQ$ , $CQ^T$ , $PC$ , $P^TC$ , $CP$ , or $CP^T$ , as specified by <i>vect</i> , <i>side</i> , and <i>trans</i> .
<i>work( 1 )</i>	If <i>info</i> = 0, on exit <i>work( 1 )</i> contains the minimum value of <i>lwork</i> required for optimum performance. Use this <i>lwork</i> for subsequent runs.
<i>info</i>	<b>INTEGER</b> . If <i>info</i> = 0, the execution is successful. If <i>info</i> = <i>-i</i> , the <i>i</i> th parameter had an illegal value.

## Application Notes

For better performance, try using

`lwork = n*blocksize` for `side = 'L'`, or

`lwork = m*blocksize` for `side = 'R'`,

where `blocksize` is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use a generous value of `lwork` for the first run. On exit, examine `work(1)` and use this value for subsequent runs.

The computed product differs from the exact product by a matrix  $E$  such that  $\|E\|_2 = O(\epsilon) \|C\|_2$ .

The total number of floating-point operations is approximately

$$\begin{aligned} 2 * n * k (2 * m - k) & \quad \text{if } \text{side} = 'L' \text{ and } m \geq k; \\ 2 * m * k (2 * n - k) & \quad \text{if } \text{side} = 'R' \text{ and } n \geq k; \\ 2 * m^2 * n & \quad \text{if } \text{side} = 'L' \text{ and } m < k; \\ 2 * n^2 * m & \quad \text{if } \text{side} = 'R' \text{ and } n < k. \end{aligned}$$

The complex counterpart of this routine is [?unmbr](#).

---

## ?ungbr

*Generates the complex unitary matrix  $Q$  or  $P^H$  determined by ?gebrd.*

---

```
call cungbr ( vect, m, n, k, a, lda, tau, work, lwork, info )
call zungbr ( vect, m, n, k, a, lda, tau, work, lwork, info )
```

### Discussion

The routine generates the whole or part of the unitary matrices  $Q$  and  $P^H$  formed by the routines `cgebrd/zgebrd` (see [page 5-76](#)). Use this routine after a call to `cgebrd/zgebrd`. All valid combinations of arguments are described in *Input Parameters*; in most cases you'll need the following:

To compute the whole  $m$  by  $m$  matrix  $Q$ :

```
call ?ungbr ( 'Q', m, m, n, a ... )
(note that the array a must have at least m columns).
```

To form the  $n$  leading columns of  $Q$  if  $m > n$ :

```
call ?ungbr ( 'Q', m, n, n, a ... )
```

To compute the whole  $n$  by  $n$  matrix  $P^H$ :

```
call ?ungbr ( 'P', n, n, m, a ... )
(note that the array a must have at least n rows).
```

To form the  $m$  leading rows of  $P^H$  if  $m < n$ :

```
call ?ungbr ( 'P', m, n, m, a ... )
```

### Input Parameters

<code>vect</code>	CHARACTER*1. Must be ' <code>Q</code> ' or ' <code>P</code> '. If <code>vect = 'Q'</code> , the routine generates the matrix $Q$ . If <code>vect = 'P'</code> , the routine generates the matrix $P^H$ .
<code>m</code>	INTEGER. The number of required rows of $Q$ or $P^H$ .
<code>n</code>	INTEGER. The number of required columns of $Q$ or $P^H$ .
<code>k</code>	INTEGER. One of the dimensions of $A$ in ?gebrd: If <code>vect = 'Q'</code> , the number of columns in $A$ ; If <code>vect = 'P'</code> , the number of rows in $A$ .

Constraints:  $m \geq 0$ ,  $n \geq 0$ ,  $k \geq 0$ .  
 For  $\text{vect} = 'Q'$ :  $k \leq n \leq m$  if  $m > k$ , or  $m = n$  if  $m \leq k$ .  
 For  $\text{vect} = 'P'$ :  $k \leq m \leq n$  if  $n > k$ , or  $m = n$  if  $n \leq k$ .

$a$ , $\text{work}$	COMPLEX for <code>cungbr</code> DOUBLE COMPLEX for <code>zungbr</code> . Arrays: $a(1:\text{lda}, :)$ is the array $a$ as returned by <code>?gebrd</code> . The second dimension of $a$ must be at least $\max(1, n)$ . $\text{work}(1:\text{lwork})$ is a workspace array.
$\text{lda}$	INTEGER. The first dimension of $a$ ; at least $\max(1, m)$ .
$\tau$	COMPLEX for <code>cungbr</code> DOUBLE COMPLEX for <code>zungbr</code> . For $\text{vect} = 'Q'$ , the array $\tau_{\text{aq}}$ as returned by <code>?gebrd</code> . For $\text{vect} = 'P'$ , the array $\tau_{\text{ap}}$ as returned by <code>?gebrd</code> . The dimension of $\tau$ must be at least $\max(1, \min(m, k))$ for $\text{vect} = 'Q'$ , or $\max(1, \min(m, k))$ for $\text{vect} = 'P'$ .
$\text{lwork}$	INTEGER. The size of the $\text{work}$ array. Constraint: $\text{lwork} \geq \max(1, \min(m, n))$ . See <i>Application notes</i> for the suggested value of $\text{lwork}$ .

## Output Parameters

$a$	Overwritten by the orthogonal matrix $Q$ or $P^T$ (or the leading rows or columns thereof) as specified by $\text{vect}$ , $m$ , and $n$ .
$\text{work}(1)$	If $\text{info} = 0$ , on exit $\text{work}(1)$ contains the minimum value of $\text{lwork}$ required for optimum performance. Use this $\text{lwork}$ for subsequent runs.
$\text{info}$	INTEGER. If $\text{info} = 0$ , the execution is successful. If $\text{info} = -i$ , the $i$ th parameter had an illegal value.

## Application Notes

For better performance, try using  $\text{lwork} = \min(\text{m}, \text{n}) * \text{blocksize}$ , where  $\text{blocksize}$  is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use a generous value of  $\text{lwork}$  for the first run. On exit, examine  $\text{work}(1)$  and use this value for subsequent runs.

The computed matrix  $Q$  differs from an exactly orthogonal matrix by a matrix  $E$  such that  $\|E\|_2 = O(\epsilon)$ .

The approximate numbers of floating-point operations for the cases listed in *Discussion* are as follows:

To form the whole of  $Q$ :

$$\begin{aligned} (16/3)\text{n}(3\text{m}^2 - 3\text{m}\ast\text{n} + \text{n}^2) &\quad \text{if } \text{m} > \text{n}; \\ (16/3)\text{m}^3 &\quad \text{if } \text{m} \leq \text{n}. \end{aligned}$$

To form the  $\text{n}$  leading columns of  $Q$  when  $\text{m} > \text{n}$ :

$$(8/3)\text{n}^2(3\text{m} - \text{n}^2) \quad \text{if } \text{m} > \text{n}.$$

To form the whole of  $P^T$ :

$$\begin{aligned} (16/3)\text{n}^3 &\quad \text{if } \text{m} \geq \text{n}; \\ (16/3)\text{m}(3\text{n}^2 - 3\text{m}\ast\text{n} + \text{m}^2) &\quad \text{if } \text{m} < \text{n}. \end{aligned}$$

To form the  $\text{m}$  leading columns of  $P^T$  when  $\text{m} < \text{n}$ :

$$(8/3)\text{n}^2(3\text{m} - \text{n}^2) \quad \text{if } \text{m} > \text{n}.$$

The real counterpart of this routine is [?orgbr](#).

## ?unmbr

*Multiplies an arbitrary complex matrix by the unitary matrix Q or P determined by ?gebrd.*

```
call cunmbr (vect,side,trans,m,n,k,a,lda,tau,c,ldc,work,lwork,info)
call zunmbr (vect,side,trans,m,n,k,a,lda,tau,c,ldc,work,lwork,info)
```

### Discussion

Given an arbitrary complex matrix  $C$ , this routine forms one of the matrix products  $QC$ ,  $Q^H C$ ,  $CQ$ ,  $CQ^H$ ,  $PC$ ,  $P^H C$ ,  $CP$ , or  $CP^H$ , where  $Q$  and  $P$  are orthogonal matrices computed by a call to [cgebrd/zgebrd](#) (see [page 5-76](#)). The routine overwrites the product on  $C$ .

### Input Parameters

In the descriptions below,  $r$  denotes the order of  $Q$  or  $P^H$ :

If  $\text{side} = \text{'L'}$ ,  $r = m$ ; if  $\text{side} = \text{'R'}$ ,  $r = n$ .

$\text{vect}$	CHARACTER*1. Must be ' $Q$ ' or ' $P$ '. If $\text{vect} = \text{'Q'}$ , then $Q$ or $Q^H$ is applied to $C$ . If $\text{vect} = \text{'P'}$ , then $P$ or $P^H$ is applied to $C$ .
$\text{side}$	CHARACTER*1. Must be ' $L$ ' or ' $R$ '. If $\text{side} = \text{'L'}$ , multipliers are applied to $C$ from the left. If $\text{side} = \text{'R'}$ , they are applied to $C$ from the right.
$\text{trans}$	CHARACTER*1. Must be ' $N$ ' or ' $C$ '. If $\text{trans} = \text{'N'}$ , then $Q$ or $P$ is applied to $C$ . If $\text{trans} = \text{'C'}$ , then $Q^H$ or $P^H$ is applied to $C$ .
$m$	INTEGER. The number of rows in $C$ .
$n$	INTEGER. The number of columns in $C$ .
$k$	INTEGER. One of the dimensions of $A$ in <a href="#">?gebrd</a> : If $\text{vect} = \text{'Q'}$ , the number of columns in $A$ ; If $\text{vect} = \text{'P'}$ , the number of rows in $A$ .

Constraints:  $m \geq 0$ ,  $n \geq 0$ ,  $k \geq 0$ .

<i>a, c, work</i>	<i>COMPLEX</i> for <i>cunmbr</i> <i>DOUBLE COMPLEX</i> for <i>zunmbr</i> . Arrays: <i>a( lda, * )</i> is the array <i>a</i> as returned by <i>?gebrd</i> . Its second dimension must be at least $\max(1, \min(r, k))$ for <i>vect</i> = 'Q', or $\max(1, r)$ for <i>vect</i> = 'P'. <i>c( ldc, * )</i> holds the matrix <i>C</i> . Its second dimension must be at least $\max(1, n)$ . <i>work( lwork )</i> is a workspace array.
<i>lda</i>	<i>INTEGER</i> . The first dimension of <i>a</i> . Constraints: <i>lda</i> $\geq \max(1, r)$ if <i>vect</i> = 'Q'; <i>lda</i> $\geq \max(1, \min(r, k))$ if <i>vect</i> = 'P'.
<i>ldc</i>	<i>INTEGER</i> . The first dimension of <i>c</i> ; <i>ldc</i> $\geq \max(1, m)$ .
<i>tau</i>	<i>COMPLEX</i> for <i>cunmbr</i> <i>DOUBLE COMPLEX</i> for <i>zunmbr</i> . Array, <i>DIMENSION</i> at least $\max(1, \min(r, k))$ . For <i>vect</i> = 'Q', the array <i>taug</i> as returned by <i>?gebrd</i> . For <i>vect</i> = 'P', the array <i>taup</i> as returned by <i>?gebrd</i> .
<i>lwork</i>	<i>INTEGER</i> . The size of the <i>work</i> array. Constraints: <i>lwork</i> $\geq \max(1, n)$ if <i>side</i> = 'L'; <i>lwork</i> $\geq \max(1, m)$ if <i>side</i> = 'R'. See <i>Application notes</i> for the suggested value of <i>lwork</i> .

## Output Parameters

<i>c</i>	Overwritten by the product $QC$ , $Q^H C$ , $CQ$ , $CQ^H$ , $PC$ , $P^H C$ , $CP$ , or $CP^H$ , as specified by <i>vect</i> , <i>side</i> , and <i>trans</i> .
<i>work( 1 )</i>	If <i>info</i> = 0, on exit <i>work( 1 )</i> contains the minimum value of <i>lwork</i> required for optimum performance. Use this <i>lwork</i> for subsequent runs.
<i>info</i>	<i>INTEGER</i> . If <i>info</i> = 0, the execution is successful. If <i>info</i> = <i>-i</i> , the <i>i</i> th parameter had an illegal value.

## Application Notes

For better performance, try using

`lwork = n*blocksize` for `side = 'L'`, or

`lwork = m*blocksize` for `side = 'R'`,

where `blocksize` is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use a generous value of `lwork` for the first run. On exit, examine `work(1)` and use this value for subsequent runs.

The computed product differs from the exact product by a matrix  $E$  such that  $\|E\|_2 = O(\epsilon) \|C\|_2$ .

The total number of floating-point operations is approximately

$$\begin{aligned} 8 * \textcolor{red}{n} * k (2 * m - k) & \quad \text{if } \textcolor{red}{side} = 'L' \text{ and } m \geq k; \\ 8 * \textcolor{red}{m} * k (2 * \textcolor{red}{n} - k) & \quad \text{if } \textcolor{red}{side} = 'R' \text{ and } \textcolor{red}{n} \geq k; \\ 8 * \textcolor{red}{m}^2 * \textcolor{red}{n} & \quad \text{if } \textcolor{red}{side} = 'L' \text{ and } m < k; \\ 8 * \textcolor{red}{n}^2 * \textcolor{red}{m} & \quad \text{if } \textcolor{red}{side} = 'R' \text{ and } \textcolor{red}{n} < k. \end{aligned}$$

The real counterpart of this routine is [?ormbr](#).

## ?bdsqr

*Computes the singular value decomposition  
of a general matrix that has been reduced  
to bidiagonal form.*

---

```
call sbdsqr ( uplo, n, ncvt, nru, ncc, d, e, vt, ldvt, u, ldu,
              c, ldc, work, info )
call dbdsqr ( uplo, n, ncvt, nru, ncc, d, e, vt, ldvt, u, ldu,
              c, ldc, work, info )
call cbdsqr ( uplo, n, ncvt, nru, ncc, d, e, vt, ldvt, u, ldu,
              c, ldc, work, info )
call zbddsr ( uplo, n, ncvt, nru, ncc, d, e, vt, ldvt, u, ldu,
              c, ldc, work, info )
```

### Discussion

This routine computes the singular values and (optionally) the left or right singular vectors of a bidiagonal matrix  $B$ . In other words, it can compute the [Singular Value Decomposition](#) (SVD):  $B = U\Sigma V^H$ . Here  $U$  and  $V$  are unitary (or, for real  $B$ , orthogonal) matrices whose columns are singular vectors;  $\Sigma$  is a diagonal matrix with real diagonal elements (see [page 5-74](#)). The singular vectors are normalized so that  $\|u_i\|_2 = \|v_i\|_2 = 1$ .

You can also use the routine for computing the SVD of a general matrix  $A$  that has been transformed to bidiagonal form  $B$  by a call to [?gebrd](#):  
 $A = QBPH$ . In this case, before calling [?bdsqr](#) call [?orgbr](#) (for real  $A$ ) or [?ungbr](#) (for complex  $A$ ) to form the matrices  $Q$  and/or  $P^H$  explicitly.

Finally, this routine can also compute the product  $U^H C$ . You'll need this product when using the SVD to solve linear least-squares problems.

### Input Parameters

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. If <i>uplo</i> = 'U', $B$ is an upper bidiagonal matrix. If <i>uplo</i> = 'L', $B$ is a lower bidiagonal matrix.
<i>n</i>	INTEGER. The order of the matrix $B$ ( $n \geq 0$ ).

<i>ncvt</i>	<b>INTEGER.</b> The number of columns in $V^T$ , that is, the number of right singular vectors ( $ncvt \geq 0$ ). Set $ncvt = 0$ if no right singular vectors are required.
<i>nru</i>	<b>INTEGER.</b> The number of rows in $U$ , that is, the number of left singular vectors ( $nru \geq 0$ ). Set $nru = 0$ if no left singular vectors are required.
<i>ncc</i>	<b>INTEGER.</b> The number of columns in the matrix $C$ used for computing the product $U^H C$ ( $ncc \geq 0$ ). Set $ncc = 0$ if no matrix $C$ is supplied.
<i>d, e, work</i>	<b>REAL</b> for single-precision flavors <b>DOUBLE PRECISION</b> for double-precision flavors. Arrays: $d(*)$ contains the diagonal elements of $B$ . The dimension of $d$ must be at least $\max(1, n)$ . $e(*)$ contains the off-diagonal elements of $B$ . The dimension of $e$ must be at least $\max(1, n-1)$ . $work(*)$ is a workspace array. The dimension of $work$ must be at least $\max(1, 2*n)$ if $ncvt = nru = ncc = 0$ ; $\max(1, 4*(n-1))$ otherwise.
<i>vt, u, c</i>	<b>REAL</b> for <b>sbdsqr</b> <b>DOUBLE PRECISION</b> for <b>dbdsqr</b> <b>COMPLEX</b> for <b>cbdsqr</b> <b>DOUBLE COMPLEX</b> for <b>zbdsqr</b> . Arrays: $vt(ldvt,*)$ contains an $n$ by $ncvt$ unit matrix if right singular vectors of $B$ are required, or the matrix $P^H$ (returned by <b>?orgbr</b> or <b>?ungbr</b> with $vect = 'P'$ ) if right singular vectors of $A$ are required. $vt$ is not referenced if $ncvt = 0$ . $u(ldu,*)$ contains an $nru$ by $n$ unit matrix if right singular vectors of $B$ are required, or the matrix $Q$ (returned by <b>?orgbr</b> or <b>?ungbr</b> with $vect = 'Q'$ ) if right singular vectors of $A$ are required; $u$ is not referenced if $nru = 0$ .

$c(\text{ldc}, *)$  contains the matrix  $C$  for computing the product  $U^T C$ . The second dimension of  $c$  must be at least  $\max(1, ncc)$ . The array is not referenced if  $ncc = 0$ .

$work(*)$  is a workspace array.

The dimension of  $work$  must be at least  
 $\max(1, 2 * n)$  if  $ncvt = nru = ncc = 0$ ;  
 $\max(1, 4 * (n - 1))$  otherwise.

$ldvt$  **INTEGER**. The first dimension of  $vt$ . Constraints:

$ldvt \geq \max(1, n)$  if  $ncvt > 0$ ;  
 $ldvt \geq 1$  otherwise.

$ldu$  **INTEGER**. The first dimension of  $u$ . Constraint:

$ldu \geq \max(1, nru)$ .

$ldc$  **INTEGER**. The first dimension of  $c$ . Constraints:

$ldc \geq \max(1, n)$  if  $ncc > 0$ ;  
 $ldc \geq 1$  otherwise.

## Output Parameters

$d$  Overwritten by the singular values in decreasing order, unless  $info > 0$  (see  $info$ ).

$e$  See  $info$  below.

$c$  Overwritten by the product  $U^T C$ .

$vt$  The  $n$  by  $ncvt$  matrix  $V^T$  whose rows are right singular vectors of  $B$  or (if computing the SVD of  $A$ ), the matrix  $V^T P^T$  whose rows are right singular vectors of  $A$ .

$u$  The  $nru$  by  $n$  matrix  $U$  whose columns are left singular vectors of  $B$  or (if computing the SVD of  $A$ ), the matrix  $QU$  whose columns are right singular vectors of  $A$ .

$c$  Overwritten by the product  $U^T C$ .

$tau$  **REAL** for  $sgeqr$   
**DOUBLE PRECISION** for  $dgeqr$   
**COMPLEX** for  $cgeqr$   
**DOUBLE COMPLEX** for  $zgeqr$ .

Array, **DIMENSION** at least  $\max(1, \min(m, n))$ .

Contains additional information on the matrix  $Q$ .

*info*

INTEGER.

If *info* = 0, the execution is successful.If *info* = *-i*, the *i*th parameter had an illegal value.If *info* = *i*, the algorithm failed to converge;*i* specifies how many off-diagonals did not converge.In this case, *d* and *e* contain on exit the diagonal and off-diagonal elements, respectively, of a bidiagonal matrix orthogonally equivalent to *B*.

### Application Notes

Each singular value and singular vector is computed to high relative accuracy. However, the reduction to bidiagonal form (prior to calling the routine) may decrease the relative accuracy in the small singular values of the original matrix if its singular values vary widely in magnitude.

If  $\sigma_i$  is an exact singular value of *B*, and  $s_i$  is the corresponding computed value, then

$$|s_i - \sigma_i| \leq p(m, n)\epsilon\sigma_i$$

where  $p(m, n)$  is a modestly increasing function of *m* and *n*, and  $\epsilon$  is the machine precision. If only singular values are computed, they are computed more accurately than when some singular vectors are also computed (that is, the function  $p(m, n)$  is smaller).

If  $u_i$  is the corresponding exact left singular vector of *B*, and  $w_i$  is the corresponding computed left singular vector, then the angle  $\theta(u_i, w_i)$  between them is bounded as follows:

$$\theta(u_i, w_i) \leq p(m, n)\epsilon / \min_{i \neq j}(|\sigma_i - \sigma_j|/|\sigma_i + \sigma_j|).$$

Here  $\min_{i \neq j}(|\sigma_i - \sigma_j|/|\sigma_i + \sigma_j|)$  is the *relative gap* between  $\sigma_i$  and the other singular values. A similar error bound holds for the right singular vectors.

The total number of real floating-point operations is roughly proportional to  $n^2$  if only the singular values are computed. About  $6n^2 * nr_u$  additional operations ( $12n^2 * nr_u$  for complex flavors) are required to compute the left singular vectors and about  $6n^2 * ncvt$  operations ( $12n^2 * ncvt$  for complex flavors) to compute the right singular vectors.

## ?bdsdc

*Computes the singular value decomposition  
of a real bidiagonal matrix using a divide  
and conquer method.*

---

```
call sbdsdc ( uplo, compq, n, d, e, u, ldu, vt, ldvt, q, iq, work,
              iwork, info )
call dbdsdc ( uplo, compq, n, d, e, u, ldu, vt, ldvt, q, iq, work,
              iwork, info )
```

### Discussion

This routine computes the [Singular Value Decomposition](#) (SVD) of a real  $n$ -by- $n$  (upper or lower) bidiagonal matrix  $B$ :  $B = U \Sigma V^T$ , using a divide and conquer method, where  $\Sigma$  is a diagonal matrix with non-negative diagonal elements (the singular values of  $B$ ), and  $U$  and  $V$  are orthogonal matrices of left and right singular vectors, respectively. `?bdsdc` can be used to compute all singular values, and optionally, singular vectors or singular vectors in compact form.

### Input Parameters

<code>uplo</code>	CHARACTER*1. Must be ' <code>U</code> ' or ' <code>L</code> '. If <code>uplo</code> = ' <code>U</code> ', $B$ is an upper bidiagonal matrix. If <code>uplo</code> = ' <code>L</code> ', $B$ is a lower bidiagonal matrix.
<code>compq</code>	CHARACTER*1. Must be ' <code>N</code> ', ' <code>P</code> ', or ' <code>I</code> '. If <code>compq</code> = ' <code>N</code> ', compute singular values only. If <code>compq</code> = ' <code>P</code> ', compute singular values and compute singular vectors in compact form. If <code>compq</code> = ' <code>I</code> ', compute singular values and singular vectors.
<code>n</code>	INTEGER. The order of the matrix $B$ ( $n \geq 0$ ).
<code>d, e, work</code>	REAL for <code>sbdsdc</code> DOUBLE PRECISION for <code>dbdsdc</code> . Arrays:

$d(*)$  contains the  $n$  diagonal elements of the bidiagonal matrix  $B$ . The dimension of  $d$  must be at least  $\max(1, n)$ .

$e(*)$  contains the off-diagonal elements of the bidiagonal matrix  $B$ . The dimension of  $e$  must be at least  $\max(1, n)$ .

$work(*)$  is a workspace array.

The dimension of  $work$  must be at least:

$\max(1, 4*n)$ , if  $compq = 'N'$ ;

$\max(1, 6*n)$ , if  $compq = 'P'$ ;

$\max(1, 3*n^2 + 2*n)$ , if  $compq = 'I'$ .

$ldu$

INTEGER. The first dimension of the output array  $u$ ;

$ldu \geq 1$ . If singular vectors are desired, then

$ldu \geq \max(1, n)$ .

$ldvt$

INTEGER. The first dimension of the output array  $vt$ ;

$ldvt \geq 1$ . If singular vectors are desired, then

$ldvt \geq \max(1, n)$ .

$iwork$

INTEGER.

Workspace array, dimension at least  $\max(1, 7*n)$ .

## Output Parameters

$d$  If  $info = 0$ , overwritten by the singular values of  $B$ .

$e$  On exit,  $e$  is overwritten.

$u, vt, q$  REAL for `sbdsdc`

DOUBLE PRECISION for `sbdsdc`.

Arrays:  $u(ldu, *)$ ,  $vt(ldvt, *)$ ,  $q(*)$ .

If  $compq = 'I'$ , then on exit  $u$  contains the left singular vectors of the bidiagonal matrix  $B$ , unless  $info \neq 0$  (see  $info$ ). For other values of  $compq$ ,  $u$  is not referenced. The second dimension of  $u$  must be at least  $\max(1, n)$ .

If  $compq = 'I'$ , then on exit  $vt$  contains the right singular vectors of the bidiagonal matrix  $B$ , unless  $info \neq 0$  (see  $info$ ). For other values of  $compq$ ,  $vt$  is not referenced. The second dimension of  $vt$  must be at least  $\max(1, n)$ .

If  $\text{compq} = 'P'$ , then on exit, if  $\text{info} = 0$ ,  $\text{q}$  and  $\text{iq}$  contain the left and right singular vectors in a compact form. Specifically,  $\text{q}$  contains all the **REAL** (for **sbdsdc**) or **DOUBLE PRECISION** (for **dbdsdc**) data for singular vectors. For other values of  $\text{compq}$ ,  $\text{q}$  is not referenced. See *Application notes* for details.

$\text{iq}$

**INTEGER.**

Array:  $\text{iq}(*)$ .

If  $\text{compq} = 'P'$ , then on exit, if  $\text{info} = 0$ ,  $\text{q}$  and  $\text{iq}$  contain the left and right singular vectors in a compact form. Specifically,  $\text{iq}$  contains all the **INTEGER** data for singular vectors. For other values of  $\text{compq}$ ,  $\text{iq}$  is not referenced. See *Application notes* for details.

$\text{info}$

**INTEGER.**

If  $\text{info} = 0$ , the execution is successful.

If  $\text{info} = -i$ , the  $i$ th parameter had an illegal value.

If  $\text{info} = i$ , the algorithm failed to compute a singular value. The update process of divide and conquer failed.

## Symmetric Eigenvalue Problems

*Symmetric eigenvalue problems* are posed as follows: given an  $n$  by  $n$  real symmetric or complex Hermitian matrix  $A$ , find the *eigenvalues*  $\lambda$  and the corresponding *eigenvectors*  $z$  that satisfy the equation

$$Az = \lambda z. \text{ (or, equivalently, } z^H A = \lambda z^H).$$

In such eigenvalue problems, all  $n$  eigenvalues are real not only for real symmetric but also for complex Hermitian matrices  $A$ , and there exists an orthonormal system of  $n$  eigenvectors. If  $A$  is a symmetric or Hermitian positive-definite matrix, all eigenvalues are positive.

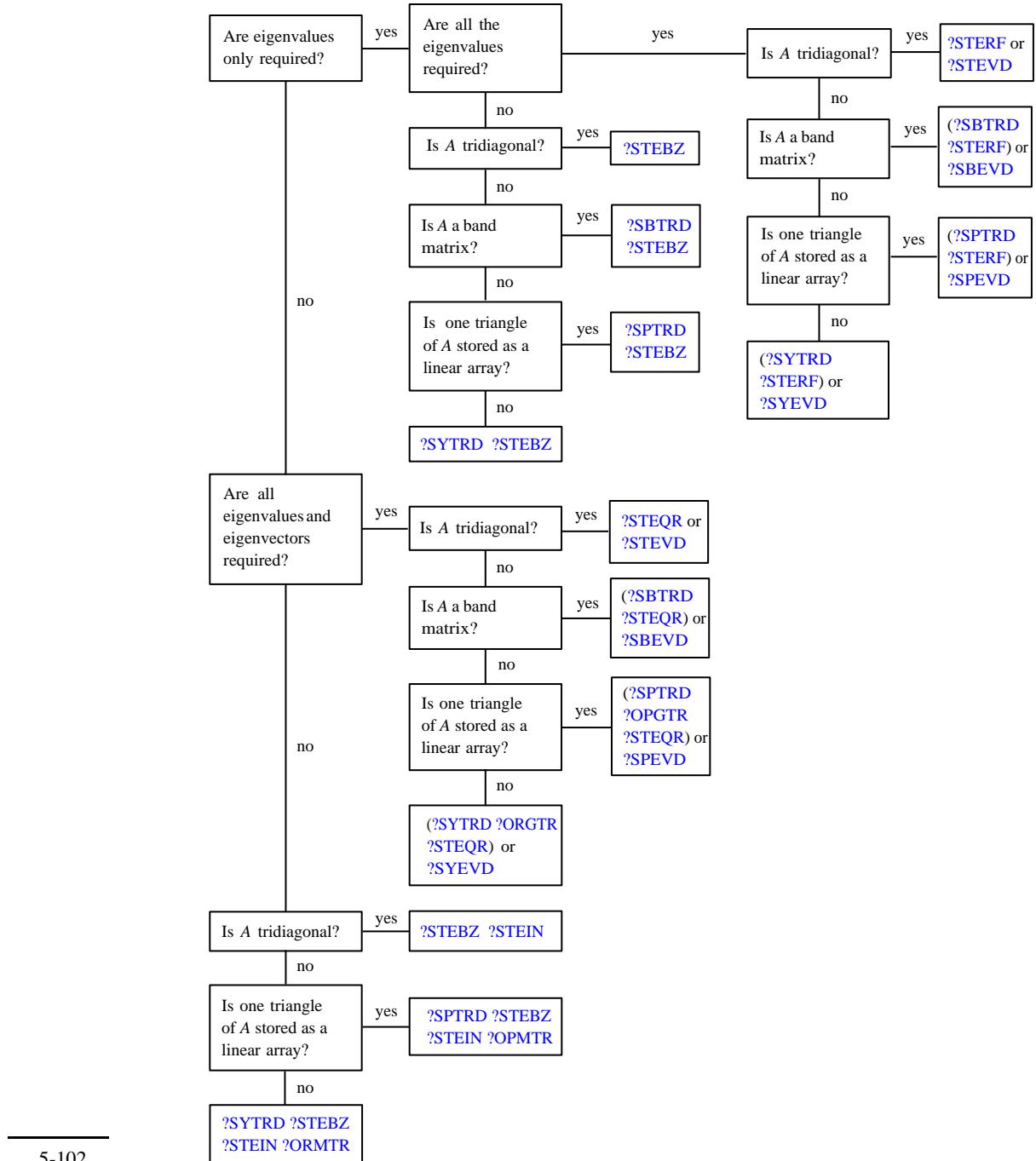
To solve a symmetric eigenvalue problem with LAPACK, you usually need to reduce the matrix to tridiagonal form and then solve the eigenvalue problem with the tridiagonal matrix obtained. LAPACK includes routines for reducing the matrix to a tridiagonal form by an orthogonal (or unitary) similarity transformation  $A = QTQ^H$  as well as for solving tridiagonal symmetric eigenvalue problems. These routines are listed in [Table 5-3](#).

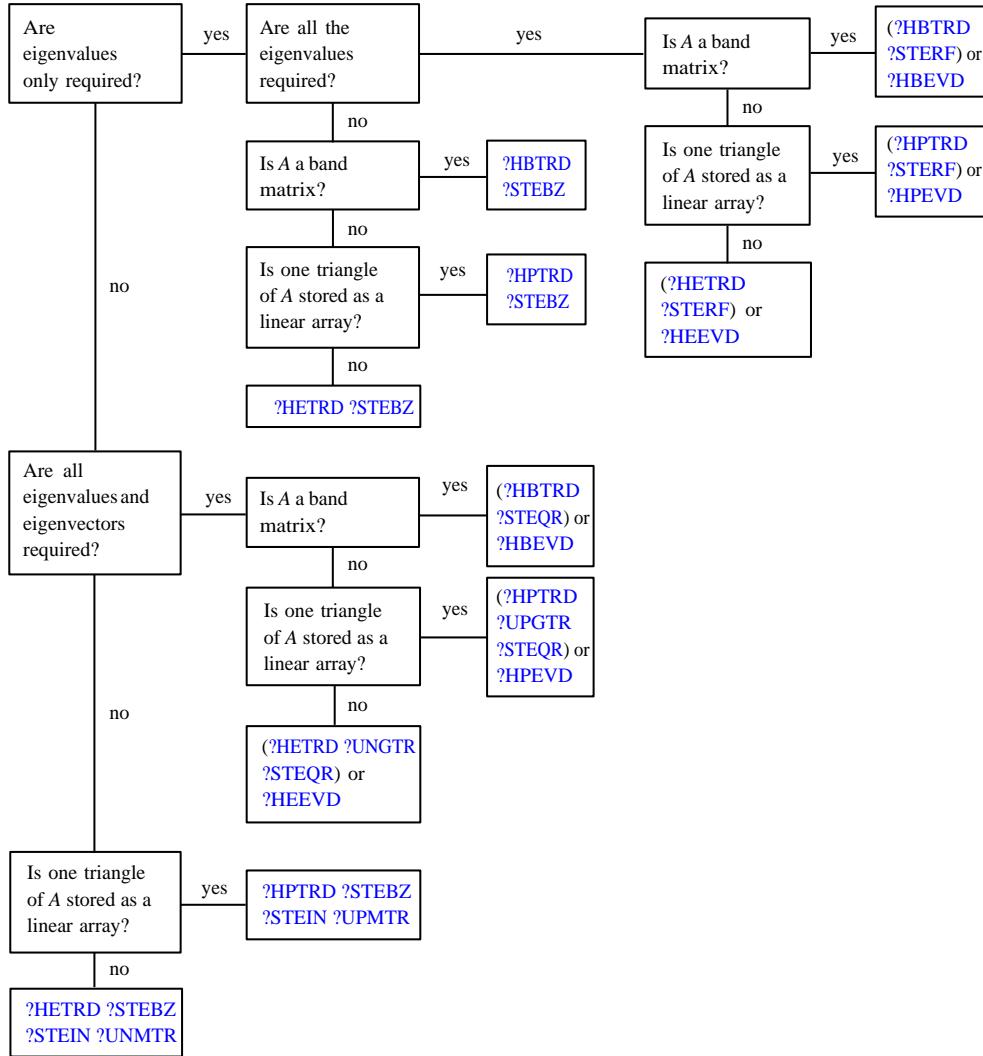
There are different routines for symmetric eigenvalue problems, depending on whether you need all eigenvectors or only some of them or eigenvalues only, whether the matrix  $A$  is positive-definite or not, and so on.

These routines are based on three primary algorithms for computing eigenvalues and eigenvectors of symmetric problems: the divide and conquer algorithm, the QR algorithm, and bisection followed by inverse iteration. The divide and conquer algorithm is generally more efficient and is recommended for computing all eigenvalues and eigenvectors.

Furthermore, to solve an eigenvalue problem using the divide and conquer algorithm, you need to call only one routine. In general, more than one routine has to be called if the QR algorithm or bisection followed by inverse iteration is used.

Decision tree in [Figure 5-2](#) will help you choose the right routine or sequence of routines for eigenvalue problems with real symmetric matrices. A similar decision tree for complex Hermitian matrices is presented in [Figure 5-3](#).

**Figure 5-2 Decision Tree: Real Symmetric Eigenvalue Problems**

**Figure 5-3 Decision Tree: Complex Hermitian Eigenvalue Problems**

**Table 5-3 Computational Routines for Solving Symmetric Eigenvalue Problems**

Operation	Real symmetric matrices	Complex Hermitian matrices
Reduce to tridiagonal form $A = QTQ^H$ (full storage)	<a href="#">?sytrd</a>	<a href="#">?hetrd</a>
Reduce to tridiagonal form $A = QTQ^H$ (packed storage)	<a href="#">?sptrd</a>	<a href="#">?hpstrd</a>
Reduce to tridiagonal form $A = QTQ^H$ (band storage).	<a href="#">?sbtrd</a>	<a href="#">?hbtrd</a>
Generate matrix $Q$ (full storage)	<a href="#">?orgtr</a>	<a href="#">?ungtr</a>
Generate matrix $Q$ (packed storage)	<a href="#">?opgtr</a>	<a href="#">?upgtr</a>
Apply matrix $Q$ (full storage)	<a href="#">?ormtr</a>	<a href="#">?unmtr</a>
Apply matrix $Q$ (packed storage)	<a href="#">?opmtr</a>	<a href="#">?upmtr</a>
Find all eigenvalues of a tridiagonal matrix $T$	<a href="#">?sterf</a>	
Find all eigenvalues and eigenvectors of a tridiagonal matrix $T$	<a href="#">?steqr</a> <a href="#">?stedc</a>	<a href="#">?steqr</a> <a href="#">?stedc</a>
Find all eigenvalues and eigenvectors of a tridiagonal positive-definite matrix $T$ .	<a href="#">?pteqr</a>	<a href="#">?pteqr</a>
Find selected eigenvalues of a tridiagonal matrix $T$	<a href="#">?stebz</a> <a href="#">?steqr</a>	<a href="#">?steqr</a>
Find selected eigenvectors of a tridiagonal matrix $T$	<a href="#">?stein</a> <a href="#">?steqr</a>	<a href="#">?stein</a> <a href="#">?steqr</a>
Compute the reciprocal condition numbers for the eigenvectors	<a href="#">?disna</a>	<a href="#">?disna</a>

## ?sytrd

Reduces a real symmetric matrix to tridiagonal form.

```
call ssytrd ( uplo,n,a,lda,d,e,tau,work,lwork,info )
call dsytrd ( uplo,n,a,lda,d,e,tau,work,lwork,info )
```

### Discussion

This routine reduces a real symmetric matrix  $A$  to symmetric tridiagonal form  $T$  by an orthogonal similarity transformation:  $A = QTQ^T$ . The orthogonal matrix  $Q$  is not formed explicitly but is represented as a product of  $n-1$  elementary reflectors. Routines are provided for working with  $Q$  in this representation. (They are described later in this section.)

### Input Parameters

<i>uplo</i>	CHARACTER*1. Must be ' <b>U</b> ' or ' <b>L</b> '. If <i>uplo</i> = ' <b>U</b> ', <i>a</i> stores the upper triangular part of <i>A</i> . If <i>uplo</i> = ' <b>L</b> ', <i>a</i> stores the lower triangular part of <i>A</i> .
<i>n</i>	INTEGER. The order of the matrix <i>A</i> ( <i>n</i> $\geq 0$ ).
<i>a</i> , <i>work</i>	REAL for <b>ssytrd</b> DOUBLE PRECISION for <b>dsytrd</b> . <i>a</i> ( <i>lda</i> ,*) is an array containing either upper or lower triangular part of the matrix <i>A</i> , as specified by <i>uplo</i> . The second dimension of <i>a</i> must be at least max(1, <i>n</i> ). <i>work(lwork)</i> is a workspace array.
<i>lda</i>	INTEGER. The first dimension of <i>a</i> ; at least max(1, <i>n</i> ).
<i>lwork</i>	INTEGER. The size of the <i>work</i> array ( <i>lwork</i> $\geq n$ ) See <i>Application notes</i> for the suggested value of <i>lwork</i> .

### Output Parameters

<i>a</i>	Overwritten by the tridiagonal matrix <i>T</i> and details of the orthogonal matrix <i>Q</i> , as specified by <i>uplo</i> .
----------	--

<i>d, e, tau</i>	<small>REAL for ssytrd DOUBLE PRECISION for dsytrd.</small>
Arrays:	
<i>d(*)</i>	contains the diagonal elements of the matrix $T$ . The dimension of <i>d</i> must be at least $\max(1, n)$ .
<i>e(*)</i>	contains the off-diagonal elements of $T$ . The dimension of <i>e</i> must be at least $\max(1, n-1)$ .
<i>tau(*)</i>	stores further details of the orthogonal matrix $Q$ . The dimension of <i>tau</i> must be at least $\max(1, n-1)$ .
<i>work(1)</i>	If <i>info</i> =0, on exit <i>work(1)</i> contains the minimum value of <i>lwork</i> required for optimum performance. Use this <i>lwork</i> for subsequent runs.
<i>info</i>	<small>INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = -<i>i</i>, the <i>i</i>th parameter had an illegal value.</small>

## Application Notes

For better performance, try using *lwork* =  $n * \text{blocksize}$ , where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*. If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run. On exit, examine *work(1)* and use this value for subsequent runs.

The computed matrix  $T$  is exactly similar to a matrix  $A + E$ , where  $\|E\|_2 = c(n)\epsilon \|A\|_2$ ,  $c(n)$  is a modestly increasing function of  $n$ , and  $\epsilon$  is the machine precision.

The approximate number of floating-point operations is  $(4/3)n^3$ .

After calling this routine, you can call the following:

<u>?orgtr</u>	to form the computed matrix $Q$ explicitly;
<u>?ormtr</u>	to multiply a real matrix by $Q$ .

The complex counterpart of this routine is ?hetrd.

## ?orgtr

Generates the real orthogonal matrix  $Q$  determined by ?sytrd.

---

```
call sorgtr ( uplo, n, a, lda, tau, work, lwork, info )
call dorgtr ( uplo, n, a, lda, tau, work, lwork, info )
```

### Discussion

The routine explicitly generates the  $n$  by  $n$  orthogonal matrix  $Q$  formed by ?sytrd (see [page 5-105](#)) when reducing a real symmetric matrix  $A$  to tridiagonal form:  $A = QTQ^T$ . Use this routine after a call to ?sytrd.

### Input Parameters

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. Use the same <i>uplo</i> as supplied to ?sytrd.
<i>n</i>	INTEGER. The order of the matrix $Q$ ( $n \geq 0$ ).
<i>a</i> , <i>tau</i> , <i>work</i>	REAL for sorgtr DOUBLE PRECISION for dorgtr. Arrays: <i>a</i> ( <i>lda</i> ,*) is the array <i>a</i> as returned by ?sytrd. The second dimension of <i>a</i> must be at least max(1, <i>n</i> ). <i>tau</i> (*) is the array <i>tau</i> as returned by ?sytrd. The dimension of <i>tau</i> must be at least max(1, <i>n</i> -1). <i>work</i> ( <i>lwork</i> ) is a workspace array.
<i>lda</i>	INTEGER. The first dimension of <i>a</i> ; at least max(1, <i>n</i> ).
<i>lwork</i>	INTEGER. The size of the <i>work</i> array ( <i>lwork</i> $\geq n$ ) See <i>Application notes</i> for the suggested value of <i>lwork</i> .

### Output Parameters

<i>a</i>	Overwritten by the orthogonal matrix $Q$ .
----------	--

*work(1)*      If *info* = 0, on exit *work(1)* contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.

*info*      INTEGER.  
If *info* = 0, the execution is successful.  
If *info* = -*i*, the *i*th parameter had an illegal value.

### Application Notes

For better performance, try using *lwork* = (*n*-1)\**blocksize*, where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run. On exit, examine *work(1)* and use this value for subsequent runs.

The computed matrix *Q* differs from an exactly orthogonal matrix by a matrix *E* such that  $\|E\|_2 = O(\epsilon)$ , where  $\epsilon$  is the machine precision.

The approximate number of floating-point operations is  $(4/3)n^3$ .

The complex counterpart of this routine is [?ungtr](#).

## ?ormtr

*Multiplies a real matrix by the real orthogonal matrix  $Q$  determined by  
?sytrd.*

```
call sormtr ( side, uplo, trans, m, n, a, lda, tau, c, ldc, work, lwork, info )
call dormtr ( side, uplo, trans, m, n, a, lda, tau, c, ldc, work, lwork, info )
```

### Discussion

The routine multiplies a real matrix  $C$  by  $Q$  or  $Q^T$ , where  $Q$  is the orthogonal matrix  $Q$  formed by ?sytrd (see [page 5-105](#)) when reducing a real symmetric matrix  $A$  to tridiagonal form:  $A = QTQ^T$ . Use this routine after a call to ?sytrd.

Depending on the parameters *side* and *trans*, the routine can form one of the matrix products  $QC$ ,  $Q^TC$ ,  $CQ$ , or  $CQ^T$  (overwriting the result on  $C$ ).

### Input Parameters

In the descriptions below, *r* denotes the order of  $Q$ :

If *side* = 'L', *r* = *m*; if *side* = 'R', *r* = *n*.

<i>side</i>	CHARACTER*1. Must be either 'L' or 'R'. If <i>side</i> = 'L', $Q$ or $Q^T$ is applied to $C$ from the left. If <i>side</i> = 'R', $Q$ or $Q^T$ is applied to $C$ from the right.
<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. Use the same <i>uplo</i> as supplied to ?sytrd.
<i>trans</i>	CHARACTER*1. Must be either 'N' or 'T'. If <i>trans</i> = 'N', the routine multiplies $C$ by $Q$ . If <i>trans</i> = 'T', the routine multiplies $C$ by $Q^T$ .
<i>m</i>	INTEGER. The number of rows in the matrix $C$ ( <i>m</i> $\geq$ 0).
<i>n</i>	INTEGER. The number of columns in $C$ ( <i>n</i> $\geq$ 0).
<i>a</i> , <i>work</i> , <i>tau</i> , <i>c</i>	REAL for sormtr DOUBLE PRECISION for dormtr. <i>a</i> ( <i>lda</i> , *) and <i>tau</i> are the arrays returned by ?sytrd.

The second dimension of  $a$  must be at least  $\max(1, r)$ .  
 The dimension of  $tau$  must be at least  $\max(1, r-1)$ .

$c(ldc, *)$  contains the matrix  $C$ .

The second dimension of  $c$  must be at least  $\max(1, n)$ .

$work(lwork)$  is a workspace array.

$lda$  INTEGER. The first dimension of  $a$ ;  $lda \geq \max(1, r)$ .

$ldc$  INTEGER. The first dimension of  $c$ ;  $ldc \geq \max(1, n)$ .

$lwork$  INTEGER. The size of the  $work$  array. Constraints:

$lwork \geq \max(1, n)$  if  $side = 'L'$ ;

$lwork \geq \max(1, m)$  if  $side = 'R'$ .

See *Application notes* for the suggested value of  $lwork$ .

## Output Parameters

$c$  Overwritten by the product  $QC$ ,  $Q^TC$ ,  $CQ$ , or  $CQ^T$  (as specified by  $side$  and  $trans$ ).

$work(1)$  If  $info = 0$ , on exit  $work(1)$  contains the minimum value of  $lwork$  required for optimum performance. Use this  $lwork$  for subsequent runs.

$info$  INTEGER.

If  $info = 0$ , the execution is successful.

If  $info = -i$ , the  $i$ th parameter had an illegal value.

## Application Notes

For better performance, try using  $lwork = n * blocksize$  for  $side = 'L'$ , or  $lwork = m * blocksize$  for  $side = 'R'$ , where  $blocksize$  is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*. If you are in doubt how much workspace to supply, use a generous value of  $lwork$  for the first run. On exit, examine  $work(1)$  and use this value for subsequent runs.

The computed product differs from the exact product by a matrix  $E$  such that  $\|E\|_2 = O(\epsilon) \|C\|_2$ .

The total number of floating-point operations is approximately  $2 * m^2 * n$  if  $side = 'L'$  or  $2 * n^2 * m$  if  $side = 'R'$ .

The complex counterpart of this routine is [?unmtr](#).

## ?hetrd

Reduces a complex Hermitian matrix to tridiagonal form.

```
call chetrd ( uplo,n,a,lda,d,e,tau,work,lwork,info )
call zhetrd ( uplo,n,a,lda,d,e,tau,work,lwork,info )
```

### Discussion

This routine reduces a complex Hermitian matrix  $A$  to symmetric tridiagonal form  $T$  by a unitary similarity transformation:  $A = QTQ^H$ . The unitary matrix  $Q$  is not formed explicitly but is represented as a product of  $n-1$  elementary reflectors. Routines are provided to work with  $Q$  in this representation. (They are described later in this section.)

### Input Parameters

<i>uplo</i>	CHARACTER*1. Must be ' <b>U</b> ' or ' <b>L</b> '. If <i>uplo</i> = ' <b>U</b> ', <i>a</i> stores the upper triangular part of <i>A</i> . If <i>uplo</i> = ' <b>L</b> ', <i>a</i> stores the lower triangular part of <i>A</i> .
<i>n</i>	INTEGER. The order of the matrix <i>A</i> ( <i>n</i> $\geq 0$ ).
<i>a</i> , <i>work</i>	COMPLEX for <b>chetrd</b> DOUBLE COMPLEX for <b>zhetrd</b> . <i>a</i> ( <i>lda</i> ,*) is an array containing either upper or lower triangular part of the matrix <i>A</i> , as specified by <i>uplo</i> . The second dimension of <i>a</i> must be at least max(1, <i>n</i> ). <i>work(lwork)</i> is a workspace array.
<i>lda</i>	INTEGER. The first dimension of <i>a</i> ; at least max(1, <i>n</i> ).
<i>lwork</i>	INTEGER. The size of the <i>work</i> array ( <i>lwork</i> $\geq n$ ) See <i>Application notes</i> for the suggested value of <i>lwork</i> .

### Output Parameters

<i>a</i>	Overwritten by the tridiagonal matrix <i>T</i> and details of the unitary matrix <i>Q</i> , as specified by <i>uplo</i> .
----------	---

<i>d, e</i>	<small>REAL for <code>chetrd</code> DOUBLE PRECISION for <code>zhetrd</code>.</small>
Arrays:	
<i>d(*)</i>	contains the diagonal elements of the matrix $T$ . The dimension of <i>d</i> must be at least $\max(1, n)$ .
<i>e(*)</i>	contains the off-diagonal elements of $T$ . The dimension of <i>e</i> must be at least $\max(1, n-1)$ .
<i>tau</i>	<small>COMPLEX for <code>chetrd</code> DOUBLE COMPLEX for <code>zhetrd</code>.</small> Array, DIMENSION at least $\max(1, n-1)$ . Stores further details of the unitary matrix $Q$ .
<i>work(1)</i>	If <i>info</i> = 0, on exit <i>work(1)</i> contains the minimum value of <i>lwork</i> required for optimum performance. Use this <i>lwork</i> for subsequent runs.
<i>info</i>	<small>INTEGER.</small> If <i>info</i> = 0, the execution is successful. If <i>info</i> = <i>-i</i> , the <i>i</i> th parameter had an illegal value.

## Application Notes

For better performance, try using  $\text{lwork} = n * \text{blocksize}$ , where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*. If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run. On exit, examine *work(1)* and use this value for subsequent runs.

The computed matrix  $T$  is exactly similar to a matrix  $A + E$ , where  $\|E\|_2 = c(n)\epsilon \|A\|_2$ ,  $c(n)$  is a modestly increasing function of  $n$ , and  $\epsilon$  is the machine precision.

The approximate number of floating-point operations is  $(16/3)n^3$ .

After calling this routine, you can call the following:

<u><a href="#">?ungtr</a></u>	to form the computed matrix $Q$ explicitly;
<u><a href="#">?unmtr</a></u>	to multiply a complex matrix by $Q$ .

The real counterpart of this routine is [?sytrd](#).

## ?ungtr

Generates the complex unitary matrix  $Q$  determined by ?hetrd.

---

```
call cungtr ( uplo, n, a, lda, tau, work, lwork, info )
call zungtr ( uplo, n, a, lda, tau, work, lwork, info )
```

### Discussion

The routine explicitly generates the  $n$  by  $n$  unitary matrix  $Q$  formed by ?hetrd (see [page 5-111](#)) when reducing a complex Hermitian matrix  $A$  to tridiagonal form:  $A = QTQ^H$ . Use this routine after a call to ?hetrd.

### Input Parameters

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. Use the same <i>uplo</i> as supplied to ?hetrd.
<i>n</i>	INTEGER. The order of the matrix $Q$ ( $n \geq 0$ ).
<i>a</i> , <i>tau</i> , <i>work</i>	COMPLEX for cungtr DOUBLE COMPLEX for zungtr. Arrays: <i>a</i> ( <i>lda</i> ,*) is the array <i>a</i> as returned by ?hetrd. The second dimension of <i>a</i> must be at least max(1, <i>n</i> ). <i>tau</i> (*) is the array <i>tau</i> as returned by ?hetrd. The dimension of <i>tau</i> must be at least max(1, <i>n</i> -1). <i>work</i> ( <i>lwork</i> ) is a workspace array.
<i>lda</i>	INTEGER. The first dimension of <i>a</i> ; at least max(1, <i>n</i> ).
<i>lwork</i>	INTEGER. The size of the <i>work</i> array ( <i>lwork</i> $\geq n$ ) See <i>Application notes</i> for the suggested value of <i>lwork</i> .

### Output Parameters

<i>a</i>	Overwritten by the unitary matrix $Q$ .
----------	---

*work(1)*      If *info* = 0, on exit *work(1)* contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.

*info*      **INTEGER.**  
If *info* = 0, the execution is successful.  
If *info* = *-i*, the *i*th parameter had an illegal value.

### Application Notes

For better performance, try using *lwork* = (*n*-1)\**blocksize*, where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run. On exit, examine *work(1)* and use this value for subsequent runs.

The computed matrix *Q* differs from an exactly unitary matrix by a matrix *E* such that  $\|E\|_2 = O(\epsilon)$ , where  $\epsilon$  is the machine precision.

The approximate number of floating-point operations is  $(16/3)n^3$ .

The real counterpart of this routine is [?orgtr](#).

## ?unmtr

*Multiplies a complex matrix by the complex unitary matrix  $Q$  determined by ?hetrd.*

```
call cunmtr ( side, uplo, trans, m, n, a, lda, tau, c, ldc, work, lwork, info )
call zunmtr ( side, uplo, trans, m, n, a, lda, tau, c, ldc, work, lwork, info )
```

### Discussion

The routine multiplies a complex matrix  $C$  by  $Q$  or  $Q^H$ , where  $Q$  is the unitary matrix  $Q$  formed by ?hetrd (see [page 5-111](#)) when reducing a complex Hermitian matrix  $A$  to tridiagonal form:  $A = QTQ^H$ . Use this routine after a call to ?hetrd.

Depending on the parameters *side* and *trans*, the routine can form one of the matrix products  $QC$ ,  $Q^HC$ ,  $CQ$ , or  $CQ^H$  (overwriting the result on  $C$ ).

### Input Parameters

In the descriptions below, *r* denotes the order of  $Q$ :

If *side* = 'L', *r* = *m*; if *side* = 'R', *r* = *n*.

<i>side</i>	CHARACTER*1. Must be either 'L' or 'R'. If <i>side</i> = 'L', $Q$ or $Q^H$ is applied to $C$ from the left. If <i>side</i> = 'R', $Q$ or $Q^H$ is applied to $C$ from the right.
<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. Use the same <i>uplo</i> as supplied to ?hetrd.
<i>trans</i>	CHARACTER*1. Must be either 'N' or 'T'. If <i>trans</i> = 'N', the routine multiplies $C$ by $Q$ . If <i>trans</i> = 'T', the routine multiplies $C$ by $Q^H$ .
<i>m</i>	INTEGER. The number of rows in the matrix $C$ ( <i>m</i> $\geq$ 0).
<i>n</i>	INTEGER. The number of columns in $C$ ( <i>n</i> $\geq$ 0).
<i>a</i> , <i>work</i> , <i>tau</i> , <i>c</i>	COMPLEX for cunmtr DOUBLE COMPLEX for zunmtr. <i>a</i> ( <i>lda</i> ,*) and <i>tau</i> are the arrays returned by ?hetrd.

The second dimension of  $a$  must be at least  $\max(1, r)$ .  
 The dimension of  $tau$  must be at least  $\max(1, r-1)$ .

$c(ldc, *)$  contains the matrix  $C$ .

The second dimension of  $c$  must be at least  $\max(1, n)$ .

$work(lwork)$  is a workspace array.

$lda$  INTEGER. The first dimension of  $a$ ;  $lda \geq \max(1, r)$ .

$ldc$  INTEGER. The first dimension of  $c$ ;  $ldc \geq \max(1, n)$ .

$lwork$  INTEGER. The size of the  $work$  array. Constraints:

$lwork \geq \max(1, n)$  if  $side = 'L'$ ;

$lwork \geq \max(1, m)$  if  $side = 'R'$ .

See *Application notes* for the suggested value of  $lwork$ .

## Output Parameters

$c$  Overwritten by the product  $QC$ ,  $Q^H C$ ,  $CQ$ , or  $CQ^H$  (as specified by  $side$  and  $trans$ ).

$work(1)$  If  $info = 0$ , on exit  $work(1)$  contains the minimum value of  $lwork$  required for optimum performance. Use this  $lwork$  for subsequent runs.

$info$  INTEGER.

If  $info = 0$ , the execution is successful.

If  $info = -i$ , the  $i$ th parameter had an illegal value.

## Application Notes

For better performance, try using  $lwork = n * blocksize$  (for  $side = 'L'$ ) or  $lwork = m * blocksize$  (for  $side = 'R'$ ) where  $blocksize$  is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*. If you are in doubt how much workspace to supply, use a generous value of  $lwork$  for the first run. On exit, examine  $work(1)$  and use this value for subsequent runs.

The computed product differs from the exact product by a matrix  $E$  such that  $\|E\|_2 = O(\epsilon) \|C\|_2$ , where  $\epsilon$  is the machine precision.

The total number of floating-point operations is approximately  $8 * m^2 * n$  if  $side = 'L'$  or  $8 * n^2 * m$  if  $side = 'R'$ .

The real counterpart of this routine is [?ormtr](#).

## ?sptrd

Reduces a real symmetric matrix to tridiagonal form using packed storage.

```
call ssptrd ( uplo,n,ap,d,e,tau,info )
call dsptrd ( uplo,n,ap,d,e,tau,info )
```

### Discussion

This routine reduces a packed real symmetric matrix  $A$  to symmetric tridiagonal form  $T$  by an orthogonal similarity transformation:  $A = QTQ^T$ . The orthogonal matrix  $Q$  is not formed explicitly but is represented as a product of  $n-1$  elementary reflectors. Routines are provided for working with  $Q$  in this representation. (They are described later in this section.)

### Input Parameters

<i>uplo</i>	CHARACTER*1. Must be ' <b>U</b> ' or ' <b>L</b> '. If <i>uplo</i> = ' <b>U</b> ', <i>ap</i> stores the packed upper triangle of $A$ . If <i>uplo</i> = ' <b>L</b> ', <i>ap</i> stores the packed lower triangle of $A$ .
<i>n</i>	INTEGER. The order of the matrix $A$ ( <i>n</i> $\geq$ 0).
<i>ap</i>	REAL for <b>ssptrd</b> DOUBLE PRECISION for <b>dsptrd</b> . Array, DIMENSION at least max(1, <i>n</i> ( <i>n</i> +1)/2). Contains either upper or lower triangle of $A$ (as specified by <i>uplo</i> ) in packed form.

### Output Parameters

<i>ap</i>	Overwritten by the tridiagonal matrix $T$ and details of the orthogonal matrix $Q$ , as specified by <i>uplo</i> .
<i>d, e, tau</i>	REAL for <b>ssptrd</b> DOUBLE PRECISION for <b>dsptrd</b> . Arrays: <i>d(*)</i> contains the diagonal elements of the matrix $T$ . The dimension of <i>d</i> must be at least max(1, <i>n</i> ).

$e(*)$  contains the off-diagonal elements of  $T$ .

The dimension of  $e$  must be at least  $\max(1, n-1)$ .

$tau(*)$  stores further details of the matrix  $Q$ .

The dimension of  $tau$  must be at least  $\max(1, n-1)$ .

$info$

INTEGER.

If  $info = 0$ , the execution is successful.

If  $info = -i$ , the  $i$ th parameter had an illegal value.

## Application Notes

The computed matrix  $T$  is exactly similar to a matrix  $A + E$ , where  $\|E\|_2 = c(n)\epsilon \|A\|_2$ ,  $c(n)$  is a modestly increasing function of  $n$ , and  $\epsilon$  is the machine precision.

The approximate number of floating-point operations is  $(4/3)n^3$ .

After calling this routine, you can call the following:

[?opgtr](#) to form the computed matrix  $Q$  explicitly;

[?opmtr](#) to multiply a real matrix by  $Q$ .

The complex counterpart of this routine is [?hptrd](#).

## ?opgtr

Generates the real orthogonal matrix  $Q$   
determined by ?sptrd.

---

```
call sopgtr ( uplo, n, ap, tau, q, ldq, work, info )
call dopgtr ( uplo, n, ap, tau, q, ldq, work, info )
```

### Discussion

The routine explicitly generates the  $n$  by  $n$  orthogonal matrix  $Q$  formed by ?sptrd (see [page 5-117](#)) when reducing a packed real symmetric matrix  $A$  to tridiagonal form:  $A = QTQ^T$ . Use this routine after a call to ?sptrd.

### Input Parameters

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. Use the same <i>uplo</i> as supplied to ?sptrd.
<i>n</i>	INTEGER. The order of the matrix $Q$ ( $n \geq 0$ ).
<i>ap</i> , <i>tau</i>	REAL for sopgtr DOUBLE PRECISION for dopgtr. Arrays <i>ap</i> and <i>tau</i> , as returned by ?sptrd. The dimension of <i>ap</i> must be at least $\max(1, n(n+1)/2)$ . The dimension of <i>tau</i> must be at least $\max(1, n-1)$ .
<i>ldq</i>	INTEGER. The first dimension of the output array <i>q</i> : at least $\max(1, n)$ .
<i>work</i>	REAL for sopgtr DOUBLE PRECISION for dopgtr. Workspace array, DIMENSION at least $\max(1, n-1)$ .

### Output Parameters

<i>q</i>	REAL for sopgtr DOUBLE PRECISION for dopgtr. Array, DIMENSION ( <i>ldq</i> , *). Contains the computed matrix $Q$ . The second dimension of <i>q</i> must be at least $\max(1, n)$ .
----------	--

*info*                    INTEGER.  
 If *info* = 0, the execution is successful.  
 If *info* = -*i*, the *i*th parameter had an illegal value.

### Application Notes

The computed matrix  $Q$  differs from an exactly orthogonal matrix by a matrix  $E$  such that  $\|E\|_2 = O(\epsilon)$ , where  $\epsilon$  is the machine precision.

The approximate number of floating-point operations is  $(4/3)n^3$ .

The complex counterpart of this routine is [?upgtr](#).

---

## ?opmtr

*Multiples a real matrix by the real orthogonal matrix Q determined by ?sptrd.*

---

```
call sopmtr (side, uplo, trans, m, n, ap, tau, c, ldc, work, info)
call dopmtr (side, uplo, trans, m, n, ap, tau, c, ldc, work, info)
```

### Discussion

The routine multiplies a real matrix  $C$  by  $Q$  or  $Q^T$ , where  $Q$  is the orthogonal matrix  $Q$  formed by [?sptrd](#) (see [page 5-117](#)) when reducing a packed real symmetric matrix  $A$  to tridiagonal form:  $A = QTQ^T$ . Use this routine after a call to [?sptrd](#).

Depending on the parameters *side* and *trans*, the routine can form one of the matrix products  $QC$ ,  $Q^TC$ ,  $CQ$ , or  $CQ^T$  (overwriting the result on  $C$ ).

### Input Parameters

In the descriptions below, *r* denotes the order of  $Q$ :

If *side* = 'L', *r* = *m*; if *side* = 'R', *r* = *n*.

<i>side</i>	CHARACTER*1. Must be either 'L' or 'R'. If <i>side</i> = 'L', $Q$ or $Q^T$ is applied to $C$ from the left. If <i>side</i> = 'R', $Q$ or $Q^T$ is applied to $C$ from the right.
-------------	--

---

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. Use the same <i>uplo</i> as supplied to ? <b>sptrd</b> .
<i>trans</i>	CHARACTER*1. Must be either 'N' or 'T'. If <i>trans</i> = 'N', the routine multiplies $C$ by $Q$ . If <i>trans</i> = 'T', the routine multiplies $C$ by $Q^T$ .
<i>m</i>	INTEGER. The number of rows in the matrix $C$ ( $m \geq 0$ ).
<i>n</i>	INTEGER. The number of columns in $C$ ( $n \geq 0$ ).
<i>ap, work, tau, c</i>	REAL for <b>sopmtr</b> DOUBLE PRECISION for <b>dopmtr</b> . <i>ap</i> and <i>tau</i> are the arrays returned by ? <b>sptrd</b> . The dimension of <i>ap</i> must be at least $\max(1, r(r+1)/2)$ . The dimension of <i>tau</i> must be at least $\max(1, r-1)$ . <i>c</i> ( <i>ldc</i> , *) contains the matrix $C$ . The second dimension of <i>c</i> must be at least $\max(1, n)$ <i>work</i> ( *) is a workspace array. The dimension of <i>work</i> must be at least $\max(1, n)$ if <i>side</i> = 'L'; $\max(1, m)$ if <i>side</i> = 'R'.
<i>ldc</i>	INTEGER. The first dimension of <i>c</i> ; <i>ldc</i> $\geq \max(1, n)$ .

### Output Parameters

<i>c</i>	Overwritten by the product $QC$ , $Q^TC$ , $CQ$ , or $CQ^T$ (as specified by <i>side</i> and <i>trans</i> ).
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = <i>-i</i> , the <i>i</i> th parameter had an illegal value.

### Application Notes

The computed product differs from the exact product by a matrix  $E$  such that  $\|E\|_2 = O(\epsilon) \|C\|_2$ , where  $\epsilon$  is the machine precision.

The total number of floating-point operations is approximately  $2*m^2*n$  if *side* = 'L' or  $2*n^2*m$  if *side* = 'R'.

The complex counterpart of this routine is [?upmtr](#).

## ?hptrd

*Reduces a complex Hermitian matrix to tridiagonal form using packed storage.*

---

```
call chptrd ( uplo,n,ap,d,e,tau,info )
call zhptrd ( uplo,n,ap,d,e,tau,info )
```

### Discussion

This routine reduces a packed complex Hermitian matrix  $A$  to symmetric tridiagonal form  $T$  by a unitary similarity transformation:  $A = QTQ^H$ . The unitary matrix  $Q$  is not formed explicitly but is represented as a product of  $n-1$  elementary reflectors. Routines are provided for working with  $Q$  in this representation. (They are described later in this section.)

### Input Parameters

<i>uplo</i>	CHARACTER*1. Must be ' <b>U</b> ' or ' <b>L</b> '. If <i>uplo</i> ='U', <i>ap</i> stores the packed upper triangle of $A$ . If <i>uplo</i> ='L', <i>ap</i> stores the packed lower triangle of $A$ .
<i>n</i>	INTEGER. The order of the matrix $A$ ( <i>n</i> $\geq$ 0).
<i>ap</i>	COMPLEX for <b>chptrd</b> DOUBLE COMPLEX for <b>zhptrd</b> . Array, DIMENSION at least max(1, <i>n(n+1)/2</i> ). Contains either upper or lower triangle of $A$ (as specified by <i>uplo</i> ) in packed form.

### Output Parameters

<i>ap</i>	Overwritten by the tridiagonal matrix $T$ and details of the orthogonal matrix $Q$ , as specified by <i>uplo</i> .
<i>d, e</i>	REAL for <b>chptrd</b> DOUBLE PRECISION for <b>zhptrd</b> . Arrays: <i>d(*)</i> contains the diagonal elements of the matrix $T$ . The dimension of <i>d</i> must be at least max(1, <i>n</i> ).

`e(*)` contains the off-diagonal elements of  $T$ .

The dimension of `e` must be at least  $\max(1, n-1)$ .

`tau`

`COMPLEX` for `chptrd`

`DOUBLE COMPLEX` for `zhptrd`.

Arrays, DIMENSION at least  $\max(1, n-1)$ .

Contains further details of the orthogonal matrix  $Q$ .

`info`

`INTEGER`.

If `info` = 0, the execution is successful.

If `info` =  $-i$ , the  $i$ th parameter had an illegal value.

## Application Notes

The computed matrix  $T$  is exactly similar to a matrix  $A + E$ , where  $\|E\|_2 = c(n)\epsilon \|A\|_2$ ,  $c(n)$  is a modestly increasing function of  $n$ , and  $\epsilon$  is the machine precision.

The approximate number of floating-point operations is  $(16/3)n^3$ .

After calling this routine, you can call the following:

`?upgtr` to form the computed matrix  $Q$  explicitly;

`?upmtr` to multiply a complex matrix by  $Q$ .

The real counterpart of this routine is `?sptrd`.

---

## ?upgtr

Generates the complex unitary matrix  $Q$  determined by ?hptrd.

---

```
call cupgtr ( uplo, n, ap, tau, q, ldq, work, info )
call zugptr ( uplo, n, ap, tau, q, ldq, work, info )
```

### Discussion

The routine explicitly generates the  $n$  by  $n$  unitary matrix  $Q$  formed by ?hptrd (see [page 5-122](#)) when reducing a packed complex Hermitian matrix  $A$  to tridiagonal form:  $A = QTQ^H$ . Use this routine after a call to ?hptrd.

### Input Parameters

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. Use the same <i>uplo</i> as supplied to ?sptrd.
<i>n</i>	INTEGER. The order of the matrix $Q$ ( $n \geq 0$ ).
<i>ap</i> , <i>tau</i>	COMPLEX for cupgtr DOUBLE COMPLEX for zugptr. Arrays <i>ap</i> and <i>tau</i> , as returned by ?hptrd. The dimension of <i>ap</i> must be at least $\max(1, n(n+1)/2)$ . The dimension of <i>tau</i> must be at least $\max(1, n-1)$ .
<i>ldq</i>	INTEGER. The first dimension of the output array <i>q</i> ; at least $\max(1, n)$ .
<i>work</i>	COMPLEX for cupgtr DOUBLE COMPLEX for zugptr. Workspace array, DIMENSION at least $\max(1, n-1)$ .

## Output Parameters

*q*                    COMPLEX for `cupgtr`  
                   DOUBLE COMPLEX for `zupgtr`.  
                   Array, DIMENSION (*ldq*, \*).  
                   Contains the computed matrix  $Q$ .  
                   The second dimension of *q* must be at least  $\max(1, n)$ .

*info*                INTEGER.  
                   If *info* = 0, the execution is successful.  
                   If *info* = *-i*, the *i*th parameter had an illegal value.

## Application Notes

The computed matrix  $Q$  differs from an exactly orthogonal matrix by a matrix  $E$  such that  $\|E\|_2 = O(\epsilon)$ , where  $\epsilon$  is the machine precision.

The approximate number of floating-point operations is  $(16/3)n^3$ .

The real counterpart of this routine is [?opgtr](#).

## ?upmtr

*Multiples a complex matrix by the unitary matrix Q determined by ?ptrd.*

```
call cupmtr (side, uplo, trans, m, n, ap, tau, c, ldc, work, info)
call zupmtr (side, uplo, trans, m, n, ap, tau, c, ldc, work, info)
```

## Discussion

The routine multiplies a complex matrix  $C$  by  $Q$  or  $Q^H$ , where  $Q$  is the unitary matrix  $Q$  formed by `?ptrd` (see [page 5-122](#)) when reducing a packed complex Hermitian matrix  $A$  to tridiagonal form:  $A = QTQ^H$ . Use this routine after a call to `?ptrd`.

Depending on the parameters *side* and *trans*, the routine can form one of the matrix products  $QC$ ,  $Q^HC$ ,  $CQ$ , or  $CQ^H$  (overwriting the result on  $C$ ).

## Input Parameters

In the descriptions below,  $r$  denotes the order of  $Q$ :

If  $\text{side} = \text{'L'}$ ,  $r = m$ ; if  $\text{side} = \text{'R'}$ ,  $r = n$ .

$\text{side}$	CHARACTER*1. Must be either ' $\text{L}$ ' or ' $\text{R}$ '.
	If $\text{side} = \text{'L'}$ , $Q$ or $Q^H$ is applied to $C$ from the left.
	If $\text{side} = \text{'R'}$ , $Q$ or $Q^H$ is applied to $C$ from the right.
$\text{uplo}$	CHARACTER*1. Must be ' $\text{U}$ ' or ' $\text{L}$ '.
	Use the same $\text{uplo}$ as supplied to ?hptrd.
$\text{trans}$	CHARACTER*1. Must be either ' $\text{N}$ ' or ' $\text{T}$ '.
	If $\text{trans} = \text{'N'}$ , the routine multiplies $C$ by $Q$ .
	If $\text{trans} = \text{'T'}$ , the routine multiplies $C$ by $Q^H$ .
$m$	INTEGER. The number of rows in the matrix $C$ ( $m \geq 0$ ).
$n$	INTEGER. The number of columns in $C$ ( $n \geq 0$ ).
$\text{ap}, \text{tau}, \text{c}, \text{work}$	COMPLEX for cupmtr DOUBLE COMPLEX for zupmtr. $\text{ap}$ and $\text{tau}$ are the arrays returned by ?hptrd.
	The dimension of $\text{ap}$ must be at least $\max(1, r(r+1)/2)$ . The dimension of $\text{tau}$ must be at least $\max(1, r-1)$ .
	$\text{c}(\text{ldc}, *)$ contains the matrix $C$ .
	The second dimension of $\text{c}$ must be at least $\max(1, n)$ .
	$\text{work}(*)$ is a workspace array.
	The dimension of $\text{work}$ must be at least $\max(1, n)$ if $\text{side} = \text{'L'}$ ; $\max(1, m)$ if $\text{side} = \text{'R'}$ .
$\text{ldc}$	INTEGER. The first dimension of $\text{c}$ ; $\text{ldc} \geq \max(1, n)$ .

## Output Parameters

$\text{c}$	Overwritten by the product $QC$ , $Q^H C$ , $CQ$ , or $CQ^H$ (as specified by $\text{side}$ and $\text{trans}$ ).
$\text{info}$	INTEGER. If $\text{info} = 0$ , the execution is successful. If $\text{info} = -i$ , the $i$ th parameter had an illegal value.

### Application Notes

The computed product differs from the exact product by a matrix  $E$  such that  $\|E\|_2 = O(\epsilon) \|C\|_2$ , where  $\epsilon$  is the machine precision.

The total number of floating-point operations is approximately  $8*m^2*n$  if  $side = 'L'$  or  $8*n^2*m$  if  $side = 'R'$ .

The real counterpart of this routine is [?opmtr](#).

## ?sbtrd

*Reduces a real symmetric band matrix to tridiagonal form.*

---

```
call ssbtrd (vect, uplo, n, kd, ab, ldab, d, e, q, ldq, work, info)
call dsbtrd (vect, uplo, n, kd, ab, ldab, d, e, q, ldq, work, info)
```

### Discussion

This routine reduces a real symmetric band matrix  $A$  to symmetric tridiagonal form  $T$  by an orthogonal similarity transformation:  $A = QTQ^T$ . The orthogonal matrix  $Q$  is determined as a product of Givens rotations. If required, the routine can also form the matrix  $Q$  explicitly.

### Input Parameters

<i>vect</i>	CHARACTER*1. Must be ' <b>V</b> ' or ' <b>N</b> '. If <i>vect</i> = ' <b>V</b> ', the routine returns the explicit matrix $Q$ . If <i>vect</i> = ' <b>N</b> ', the routine does not return $Q$ .
<i>uplo</i>	CHARACTER*1. Must be ' <b>U</b> ' or ' <b>L</b> '. If <i>uplo</i> = ' <b>U</b> ', <i>ab</i> stores the upper triangular part of $A$ . If <i>uplo</i> = ' <b>L</b> ', <i>ab</i> stores the lower triangular part of $A$ .
<i>n</i>	INTEGER. The order of the matrix $A$ ( <i>n</i> $\geq$ 0).
<i>kd</i>	INTEGER. The number of super- or sub-diagonals in $A$ ( <i>kd</i> $\geq$ 0).
<i>ab</i> , <i>work</i>	REAL for ssbtrd DOUBLE PRECISION for dsbtrd. <i>ab</i> ( <i>ldab</i> , *) is an array containing either upper or lower triangular part of the matrix $A$ (as specified by <i>uplo</i> ) in band storage format. The second dimension of <i>ab</i> must be at least max(1, <i>n</i> ). <i>work</i> (*) is a workspace array. The dimension of <i>work</i> must be at least max(1, <i>n</i> ).
<i>ldab</i>	INTEGER. The first dimension of <i>ab</i> , at least <i>kd</i> +1.

*ldq*                    **INTEGER.** The first dimension of *q*. Constraints:  
 $ldq \geq \max(1, n)$  if *vect* = 'V';  
 $ldq \geq 1$  if *vect* = 'N'.

### Output Parameters

*ab*                    On exit, the array *ab* is overwritten.

*d, e, q*                **REAL** for *ssbtrd*  
**DOUBLE PRECISION** for *dsbtrd*.

Arrays:

*d*(\*) contains the diagonal elements of the matrix *T*.

The dimension of *d* must be at least  $\max(1, n)$ .

*e*(\*) contains the off-diagonal elements of *T*.

The dimension of *e* must be at least  $\max(1, n-1)$ .

*q*(*ldq*, \*) is not referenced if *vect* = 'N'.

If *vect* = 'V', *q* contains the *n* by *n* matrix *Q*.

The second dimension of *q* must be:

at least  $\max(1, n)$  if *vect* = 'V';

at least 1 if *vect* = 'N'.

*info*                    **INTEGER.**

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*th parameter had an illegal value.

### Application Notes

The computed matrix *T* is exactly similar to a matrix *A* + *E*, where  $\|E\|_2 = c(n)\epsilon \|A\|_2$ ,  $c(n)$  is a modestly increasing function of *n*, and  $\epsilon$  is the machine precision. The computed matrix *Q* differs from an exactly orthogonal matrix by a matrix *E* such that  $\|E\|_2 = O(\epsilon)$ .

The total number of floating-point operations is approximately  $6n^2 * kd$  if *vect* = 'N', with  $3n^3 * (kd-1)/kd$  additional operations if *vect* = 'V'.

The complex counterpart of this routine is [?hbtrd](#).

## ?hbtrd

*Reduces a complex Hermitian band matrix  
to tridiagonal form.*

---

```
call chbtrd (vect, uplo, n, kd, ab, ldab, d, e, q, ldq, work, info)
call zhbtrd (vect, uplo, n, kd, ab, ldab, d, e, q, ldq, work, info)
```

### Discussion

This routine reduces a complex Hermitian band matrix  $A$  to symmetric tridiagonal form  $T$  by a unitary similarity transformation:  $A = QTQ^H$ . The unitary matrix  $Q$  is determined as a product of Givens rotations. If required, the routine can also form the matrix  $Q$  explicitly.

### Input Parameters

<i>vect</i>	CHARACTER*1. Must be ' <b>V</b> ' or ' <b>N</b> '. If <i>vect</i> = ' <b>V</b> ', the routine returns the explicit matrix $Q$ . If <i>vect</i> = ' <b>N</b> ', the routine does not return $Q$ .
<i>uplo</i>	CHARACTER*1. Must be ' <b>U</b> ' or ' <b>L</b> '. If <i>uplo</i> = ' <b>U</b> ', <i>ab</i> stores the upper triangular part of $A$ . If <i>uplo</i> = ' <b>L</b> ', <i>ab</i> stores the lower triangular part of $A$ .
<i>n</i>	INTEGER. The order of the matrix $A$ ( <i>n</i> $\geq$ 0).
<i>kd</i>	INTEGER. The number of super- or sub-diagonals in $A$ ( <i>kd</i> $\geq$ 0).
<i>ab</i> , <i>work</i>	COMPLEX for <b>chbtrd</b> DOUBLE COMPLEX for <b>zhbtrd</b> . <i>ab</i> ( <i>ldab</i> , *) is an array containing either upper or lower triangular part of the matrix $A$ (as specified by <i>uplo</i> ) in band storage format. The second dimension of <i>ab</i> must be at least $\max(1, n)$ . <i>work</i> (*) is a workspace array. The dimension of <i>work</i> must be at least $\max(1, n)$ .
<i>ldab</i>	INTEGER. The first dimension of <i>ab</i> ; at least <i>kd</i> +1.

*ldq*                    **INTEGER.** The first dimension of *q*. Constraints:  
 $ldq \geq \max(1, n)$  if *vect* = 'V';  
 $ldq \geq 1$  if *vect* = 'N'.

### Output Parameters

*ab*                    On exit, the array *ab* is overwritten.

*d, e*                    **REAL** for *chbtrd*  
**DOUBLE PRECISION** for *zhbtrd*.

Arrays:

*d(\*)* contains the diagonal elements of the matrix *T*.  
The dimension of *d* must be at least  $\max(1, n)$ .

*e(\*)* contains the off-diagonal elements of *T*.  
The dimension of *e* must be at least  $\max(1, n-1)$ .

*q*                    **COMPLEX** for *chbtrd*  
**DOUBLE COMPLEX** for *zhbtrd*.

Array, **DIMENSION** (*ldq,\**).

If *vect* = 'N', *q* is not referenced.

If *vect* = 'V', *q* contains the *n* by *n* matrix *Q*.

The second dimension of *q* must be:

at least  $\max(1, n)$  if *vect* = 'V';

at least 1 if *vect* = 'N'.

*info*                    **INTEGER.**

If *info* = 0, the execution is successful.

If *info* = *-i*, the *i*th parameter had an illegal value.

### Application Notes

The computed matrix *T* is exactly similar to a matrix *A* + *E*, where  $\|E\|_2 = c(n)\epsilon \|A\|_2$ , *c(n)* is a modestly increasing function of *n*, and  $\epsilon$  is the machine precision. The computed matrix *Q* differs from an exactly unitary matrix by a matrix *E* such that  $\|E\|_2 = O(\epsilon)$ .

The total number of floating-point operations is approximately  $20n^2 * kd$  if *vect* = 'N', with  $10n^3 * (kd-1)/kd$  additional operations if *vect* = 'V'.

The real counterpart of this routine is [?sbtrd](#).

---

## ?sterf

*Computes all eigenvalues of a real symmetric tridiagonal matrix using QR algorithm.*

---

```
call ssterf ( n, d, e, info )
call dsterf ( n, d, e, info )
```

### Discussion

This routine computes all the eigenvalues of a real symmetric tridiagonal matrix  $T$  (which can be obtained by reducing a symmetric or Hermitian matrix to tridiagonal form). The routine uses a square-root-free variant of the  $QR$  algorithm.

If you need not only the eigenvalues but also the eigenvectors, call [?steqr](#) ([page 5-134](#)).

### Input Parameters

*n*                    INTEGER. The order of the matrix  $T$  ( $n \geq 0$ ).

*d, e*                REAL for **ssterf**  
                        DOUBLE PRECISION for **dsterf**.

Arrays:

*d(\*)* contains the diagonal elements of  $T$ .  
The dimension of *d* must be at least  $\max(1, n)$ .  
*e(\*)* contains the off-diagonal elements of  $T$ .  
The dimension of *e* must be at least  $\max(1, n-1)$ .

### Output Parameters

*d*                    The *n* eigenvalues in ascending order, unless *info* > 0.  
See also *info*.

*e*                    On exit, the array is overwritten; see *info*.

*info*

INTEGER.

If *info* = 0, the execution is successful.

If *info* = *i*, the algorithm failed to find all the eigenvalues after  $30n$  iterations: *i* off-diagonal elements have not converged to zero. On exit, *d* and *e* contain, respectively, the diagonal and off-diagonal elements of a tridiagonal matrix orthogonally similar to *T*.

If *info* = *-i*, the *i*th parameter had an illegal value.

### Application Notes

The computed eigenvalues and eigenvectors are exact for a matrix  $T + E$  such that  $\|E\|_2 = O(\epsilon) \|T\|_2$ , where  $\epsilon$  is the machine precision.

If  $\lambda_i$  is an exact eigenvalue, and  $\mu_i$  is the corresponding computed value, then

$$|\mu_i - \lambda_i| \leq c(n)\epsilon \|T\|_2$$

where  $c(n)$  is a modestly increasing function of  $n$ .

The total number of floating-point operations depends on how rapidly the algorithm converges. Typically, it is about  $14n^2$ .

---

## ?steqr

*Computes all eigenvalues and eigenvectors of a symmetric or Hermitian matrix reduced to tridiagonal form (QR algorithm).*

---

```
call ssteqr ( compz, n, d, e, z, ldz, work, info )
call dsteqr ( compz, n, d, e, z, ldz, work, info )
call csteqr ( compz, n, d, e, z, ldz, work, info )
call zsteqr ( compz, n, d, e, z, ldz, work, info )
```

### Discussion

This routine computes all the eigenvalues and (optionally) all the eigenvectors of a real symmetric tridiagonal matrix  $T$ . In other words, the routine can compute the spectral factorization:  $T = Z\Lambda Z^T$ .

Here  $\Lambda$  is a diagonal matrix whose diagonal elements are the eigenvalues  $\lambda_i$ ;  $Z$  is an orthogonal matrix whose columns are eigenvectors. Thus,

$$Tz_i = \lambda_i z_i \text{ for } i = 1, 2, \dots, n.$$

(The routine normalizes the eigenvectors so that  $\|z_i\|_2 = 1$ .)

You can also use the routine for computing the eigenvalues and eigenvectors of an arbitrary real symmetric (or complex Hermitian) matrix  $A$  reduced to tridiagonal form  $T$ :  $A = QTQ^H$ . In this case, the spectral factorization is as follows:  $A = QTQ^H = (QZ)\Lambda(QZ)^H$ . Before calling `?steqr`, you must reduce  $A$  to tridiagonal form and generate the explicit matrix  $Q$  by calling the following routines:

	for real matrices:	for complex matrices:
full storage	<code>?sytrd</code> , <code>?orgtr</code>	<code>?hetrd</code> , <code>?ungtr</code>
packed storage	<code>?ptrd</code> , <code>?opgtr</code>	<code>?ptrd</code> , <code>?upgtr</code>
band storage	<code>?sbtrd</code> ( <code>vect='V'</code> )	<code>?hbtrd</code> ( <code>vect='V'</code> )

If you need eigenvalues only, it's more efficient to call `?sterf` ([page 5-132](#)). If  $T$  is positive-definite, `?pteqr` ([page 5-146](#)) can compute small eigenvalues more accurately than `?steqr`.

To solve the problem by a single call, use one of the divide and conquer routines `?stevd`, `?syevd`, `?spevd`, or `?sbevd` for real symmetric matrices or `?heevd`, `?hpevd`, or `?hbevd` for complex Hermitian matrices.

## Input Parameters

<i>compz</i>	CHARACTER*1. Must be ' <b>N</b> ' or ' <b>I</b> ' or ' <b>V</b> '. If <i>compz</i> = ' <b>N</b> ', the routine computes eigenvalues only. If <i>compz</i> = ' <b>I</b> ', the routine computes the eigenvalues and eigenvectors of the tridiagonal matrix $T$ . If <i>compz</i> = ' <b>V</b> ', the routine computes the eigenvalues and eigenvectors of $A$ (and the array <i>z</i> must contain the matrix $Q$ on entry).
<i>n</i>	INTEGER. The order of the matrix $T$ ( <i>n</i> $\geq$ 0).
<i>d, e, work</i>	REAL for single-precision flavors DOUBLE PRECISION for double-precision flavors. Arrays: <i>d</i> (*) contains the diagonal elements of $T$ . The dimension of <i>d</i> must be at least max(1, <i>n</i> ). <i>e</i> (*) contains the off-diagonal elements of $T$ . The dimension of <i>e</i> must be at least max(1, <i>n</i> -1). <i>work</i> (*) is a workspace array. The dimension of <i>work</i> must be: at least 1 if <i>compz</i> = ' <b>N</b> '; at least max(1, 2*i <i>n</i> -2) if <i>compz</i> = ' <b>V</b> ' or ' <b>I</b> '.
<i>z</i>	REAL for <b>ssteqr</b> DOUBLE PRECISION for <b>dsteqr</b> COMPLEX for <b>csteqr</b> DOUBLE COMPLEX for <b>zsteqr</b> . Array, DIMENSION ( <i>ldz</i> , *) If <i>compz</i> = ' <b>N</b> ' or ' <b>I</b> ', <i>z</i> need not be set. If <i>vect</i> = ' <b>V</b> ', <i>z</i> must contain the <i>n</i> by <i>n</i> matrix $Q$ . The second dimension of <i>z</i> must be: at least 1 if <i>compz</i> = ' <b>N</b> '; at least max(1, <i>n</i> ) if <i>compz</i> = ' <b>V</b> ' or ' <b>I</b> '. <i>work</i> ( <i>lwork</i> ) is a workspace array.
<i>ldz</i>	INTEGER. The first dimension of <i>z</i> . Constraints: <i>ldz</i> $\geq$ 1 if <i>compz</i> = ' <b>N</b> '; <i>ldz</i> $\geq$ max(1, <i>n</i> ) if <i>compz</i> = ' <b>V</b> ' or ' <b>I</b> '.

## Output Parameters

<i>d</i>	The <i>n</i> eigenvalues in ascending order, unless <i>info</i> > 0. See also <i>info</i> .
<i>e</i>	On exit, the array is overwritten; see <i>info</i> .
<i>z</i>	If <i>info</i> = 0, contains the <i>n</i> orthonormal eigenvectors, stored by columns. (The <i>i</i> th column corresponds to the <i>i</i> th eigenvalue.)
<i>info</i>	<b>INTEGER.</b> If <i>info</i> = 0, the execution is successful. If <i>info</i> = <i>i</i> , the algorithm failed to find all the eigenvalues after $30n$ iterations: <i>i</i> off-diagonal elements have not converged to zero. On exit, <i>d</i> and <i>e</i> contain, respectively, the diagonal and off-diagonal elements of a tridiagonal matrix orthogonally similar to <i>T</i> . If <i>info</i> = - <i>i</i> , the <i>i</i> th parameter had an illegal value.

## Application Notes

The computed eigenvalues and eigenvectors are exact for a matrix  $T + E$  such that  $\|E\|_2 = O(\epsilon) \|T\|_2$ , where  $\epsilon$  is the machine precision.

If  $\lambda_i$  is an exact eigenvalue, and  $\mu_i$  is the corresponding computed value, then

$$|\mu_i - \lambda_i| \leq c(n)\epsilon \|T\|_2$$

where  $c(n)$  is a modestly increasing function of *n*.

If  $z_i$  is the corresponding exact eigenvector, and  $w_i$  is the corresponding computed vector, then the angle  $\theta(z_i, w_i)$  between them is bounded as follows:

$$\theta(z_i, w_i) \leq c(n)\epsilon \|T\|_2 / \min_{i \neq j} |\lambda_i - \lambda_j|.$$

The total number of floating-point operations depends on how rapidly the algorithm converges. Typically, it is about

$$24n^2 \text{ if } \text{compz} = 'N';$$

$$7n^3 \text{ (for complex flavors, } 14n^3 \text{) if } \text{compz} = 'V' \text{ or } 'I'.$$

## ?stedc

*Computes all eigenvalues and eigenvectors of a symmetric tridiagonal matrix using the divide and conquer method.*

```
call sstedc(compz, n, d, e, z, ldz, work, lwork, iwork, liwork, info)
call dstedc(compz, n, d, e, z, ldz, work, lwork, iwork, liwork, info)
call cstedc(compz, n, d, e, z, ldz, work, lwork, rwork, lrwork,
            iwork, liwork, info)
call zstedc(compz, n, d, e, z, ldz, work, lwork, rwork, lrwork,
            iwork, liwork, info)
```

### Discussion

This routine computes all the eigenvalues and (optionally) all the eigenvectors of a symmetric tridiagonal matrix using the divide and conquer method.

The eigenvectors of a full or band real symmetric or complex Hermitian matrix can also be found if `?sytrd/?hetrd` or `?sptrd/?hptrd` or `?sbtrd/?hbtrd` has been used to reduce this matrix to tridiagonal form.

### Input Parameters

<code>compz</code>	<code>CHARACTER*1</code> . Must be ' <code>N</code> ' or ' <code>I</code> ' or ' <code>V</code> '. If <code>compz = 'N'</code> , the routine computes eigenvalues only. If <code>compz = 'I'</code> , the routine computes the eigenvalues and eigenvectors of the tridiagonal matrix. If <code>compz = 'V'</code> , the routine computes the eigenvalues and eigenvectors of original symmetric/Hermitian matrix. On entry, the array <code>z</code> must contain the orthogonal/unitary matrix used to reduce the original matrix to tridiagonal form.
<code>n</code>	<code>INTEGER</code> . The order of the symmetric tridiagonal matrix ( <code>n ≥ 0</code> ).

<i>d, e, rwork</i>	<b>REAL</b> for single-precision flavors <b>DOUBLE PRECISION</b> for double-precision flavors. Arrays: <i>d(*)</i> contains the diagonal elements of the tridiagonal matrix. The dimension of <i>d</i> must be at least $\max(1, n)$ . <i>e(*)</i> contains the subdiagonal elements of the tridiagonal matrix. The dimension of <i>e</i> must be at least $\max(1, n-1)$ . <i>rwork(lrwork)</i> is a workspace array used in complex flavors only.
<i>z, work</i>	<b>REAL</b> for <i>sstedc</i> <b>DOUBLE PRECISION</b> for <i>dstedc</i> <b>COMPLEX</b> for <i>cstedc</i> <b>DOUBLE COMPLEX</b> for <i>zstedc</i> . Arrays: <i>z(ldz, *)</i> , <i>work(*)</i> . If <i>compz</i> = 'V', then, on entry, <i>z</i> must contain the orthogonal/unitary matrix used to reduce the original matrix to tridiagonal form. The second dimension of <i>z</i> must be at least $\max(1, n)$ . <i>work(lwork)</i> is a workspace array.
<i>ldz</i>	<b>INTEGER</b> . The first dimension of <i>z</i> . Constraints: $ldz \geq 1$ if <i>compz</i> = 'N'; $ldz \geq \max(1, n)$ if <i>compz</i> = 'V' or 'I'.
<i>lwork</i>	<b>INTEGER</b> . The dimension of the array <i>work</i> . See <i>Application Notes</i> for the required value of <i>lwork</i> .
<i>lrwork</i>	<b>INTEGER</b> . The dimension of the array <i>rwork</i> (used for complex flavors only). See <i>Application Notes</i> for the required value of <i>lrwork</i> .
<i>iwork</i>	<b>INTEGER</b> . Workspace array, <b>DIMENSION</b> ( <i>liwork</i> ).
<i>liwork</i>	<b>INTEGER</b> . The dimension of the array <i>iwork</i> . See <i>Application Notes</i> for the required value of <i>liwork</i> .

## Output Parameters

<i>d</i>	The <i>n</i> eigenvalues in ascending order, unless <i>info</i> ≠ 0. See also <i>info</i> .
<i>e</i>	On exit, the array is overwritten; see <i>info</i> .
<i>z</i>	If <i>info</i> = 0, then if <i>compz</i> = 'V', <i>z</i> contains the orthonormal eigenvectors of the original symmetric/Hermitian matrix, and if <i>compz</i> = 'I', <i>z</i> contains the orthonormal eigenvectors of the symmetric tridiagonal matrix. If <i>compz</i> = 'N', <i>z</i> is not referenced.
<i>work(1)</i>	On exit, if <i>info</i> = 0, then <i>work(1)</i> returns the optimal <i>lwork</i> .
<i>rwork(1)</i>	On exit, if <i>info</i> = 0, then <i>rwork(1)</i> returns the optimal <i>lrwork</i> (for complex flavors only).
<i>iwork(1)</i>	On exit, if <i>info</i> = 0, then <i>iwork(1)</i> returns the optimal <i>liwork</i> .
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> th parameter had an illegal value. If <i>info</i> = <i>i</i> , the algorithm failed to compute an eigenvalue while working on the submatrix lying in rows and columns <i>i</i> /( <i>n</i> +1) through mod( <i>i</i> , <i>n</i> +1).

## Application Notes

The required size of workspace arrays must be as follows.

For *sstedc/dstedc*:

- If *compz* = 'N' or *n* ≤ 1 then *lwork* must be at least 1.
- If *compz* = 'V' and *n* > 1 then *lwork* must be at least  $(1 + 3n + 2n \lg n + 3n^2)$ , where  $\lg(n)$  = smallest integer *k* such that  $2^k \geq n$ .
- If *compz* = 'I' and *n* > 1 then *lwork* must be at least  $(1 + 4n + n^2)$ .
- If *compz* = 'N' or *n* ≤ 1 then *liwork* must be at least 1.
- If *compz* = 'V' and *n* > 1 then *liwork* must be at least  $(6 + 6n + 5n \lg n)$ .
- If *compz* = 'I' and *n* > 1 then *liwork* must be at least  $(3 + 5n)$ .

For *cstedc/zstedc*:

If  $\text{compz} = 'N'$  or  $'I'$ , or  $n \leq 1$ ,  $\text{lwork}$  must be at least 1.

If  $\text{compz} = 'V'$  and  $n > 1$ ,  $\text{lwork}$  must be at least  $n^2$ .

If  $\text{compz} = 'N'$  or  $n \leq 1$ ,  $\text{lrwork}$  must be at least 1.

If  $\text{compz} = 'V'$  and  $n > 1$ ,  $\text{lrwork}$  must be at least  
 $(1 + 3n + 2n \cdot \lg n + 3n^2)$ , where  $\lg(n) =$  smallest integer  $k$  such that  $2^k \geq n$ .

If  $\text{compz} = 'I'$  and  $n > 1$ ,  $\text{lrwork}$  must be at least  $(1 + 4n + 2n^2)$ .

The required value of  $\text{liwork}$  for complex flavors is the same as for real flavors.

## ?steqr

*Computes selected eigenvalues and eigenvectors of a real symmetric tridiagonal matrix.*

```
call ssteqr (jobz, range, n, d, e, vl, vu, il, iu, abstol, m, w, z,
            ldz, isuppz, work, lwork, iwork, liwork, info)
call dstegr (jobz, range, n, d, e, vl, vu, il, iu, abstol, m, w, z,
            ldz, isuppz, work, lwork, iwork, liwork, info)
call csteqr (jobz, range, n, d, e, vl, vu, il, iu, abstol, m, w, z,
            ldz, isuppz, work, lwork, iwork, liwork, info)
call zsteqr (jobz, range, n, d, e, vl, vu, il, iu, abstol, m, w, z,
            ldz, isuppz, work, lwork, iwork, liwork, info)
```

### Discussion

This routine computes selected eigenvalues and, optionally, eigenvectors of a real symmetric tridiagonal matrix  $T$ . Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues. The eigenvalues are computed by the *dqds* algorithm, while orthogonal eigenvectors are computed from various "good"  $LDL^T$  representations (also known as Relatively Robust Representations). Gram-Schmidt orthogonalization is avoided as far as possible. More specifically, the various steps of the algorithm are as follows. For the  $i$ -th unreduced block of  $T$ ,

- (a) Compute  $T - \sigma_i = L_i D_i L_i^T$ , such that  $L_i D_i L_i^T$  is a relatively robust representation;
- (b) Compute the eigenvalues,  $\lambda_j$ , of  $L_i D_i L_i^T$  to high relative accuracy by the *dqds* algorithm;
- (c) If there is a cluster of close eigenvalues, "choose"  $\sigma_i$  close to the cluster, and go to step (a);
- (d) Given the approximate eigenvalue  $\lambda_j$  of  $L_i D_i L_i^T$ , compute the corresponding eigenvector by forming a rank-revealing twisted factorization.

The desired accuracy of the output can be specified by the input parameter *abstol*.

### Input Parameters

<i>jobz</i>	CHARACTER*1. Must be ' <b>N</b> ' or ' <b>V</b> '. If <i>job</i> = ' <b>N</b> ', then only eigenvalues are computed. If <i>job</i> = ' <b>V</b> ', then eigenvalues and eigenvectors are computed.
<i>range</i>	CHARACTER*1. Must be ' <b>A</b> ' or ' <b>V</b> ' or ' <b>I</b> '. If <i>range</i> = ' <b>A</b> ', the routine computes all eigenvalues. If <i>range</i> = ' <b>V</b> ', the routine computes eigenvalues $\lambda_i$ in the half-open interval: <i>vl</i> < $\lambda_i \leq$ <i vu="">. If <i>range</i> = '<b>I</b>', the routine computes eigenvalues with indices <i>il</i> to <i>iu</i>.</i>
<i>n</i>	INTEGER. The order of the matrix <i>T</i> ( <i>n</i> $\geq$ 0).
<i>d, e, work</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Arrays: <i>d(*)</i> contains the diagonal elements of <i>T</i> . The dimension of <i>d</i> must be at least max(1, <i>n</i> ). <i>e(*)</i> contains the subdiagonal elements of <i>T</i> in elements 1 to <i>n</i> -1; <i>e(n)</i> need not be set. The dimension of <i>e</i> must be at least max(1, <i>n</i> ). <i>work(lwork)</i> is a workspace array.
<i>vl, vu</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. If <i>range</i> = ' <b>V</b> ', the lower and upper bounds of the interval to be searched for eigenvalues. Constraint: <i>vl</i> < <i vu="">.</i> If <i>range</i> = ' <b>A</b> ' or ' <b>I</b> ', <i>vl</i> and <i i="" vu<=""> are not referenced.</i>
<i>il, iu</i>	INTEGER. If <i>range</i> = ' <b>I</b> ', the indices in ascending order of the smallest and largest eigenvalues to be returned. Constraint: $1 \leq il \leq iu \leq n$ , if <i>n</i> > 0; <i>il</i> =1 and <i>iu</i> =0 if <i>n</i> = 0.

If `range = 'A'` or `'V'`, `il` and `iu` are not referenced.

`abstol`

`REAL` for single precision flavors

`DOUBLE PRECISION` for double precision flavors.

The absolute tolerance to which each eigenvalue/eigenvector is required.

If `jobz = 'V'`, the eigenvalues and eigenvectors output have residual norms bounded by `abstol`, and the dot products between different eigenvectors are bounded by `abstol`. If  $abstol < n\epsilon\|T\|_1$ , then  $n\epsilon\|T\|_1$  will be used in its place, where  $\epsilon$  is the machine precision. The eigenvalues are computed to an accuracy of  $\epsilon\|T\|_1$  irrespective of `abstol`. If high relative accuracy is important, set `abstol` to `?lamch` ('Safe minimum').

`ldz`

`INTEGER`. The leading dimension of the output array `z`.

Constraints:

$ldz \geq 1$  if `jobz = 'N'`;

$ldz \geq \max(1, n)$  if `jobz = 'V'`.

`lwork`

`INTEGER`. The dimension of the array `work`,

$lwork \geq \max(1, 18n)$ .

`iwork`

`INTEGER`.

Workspace array, `DIMENSION` (`liwork`).

`liwork`

`INTEGER`. The dimension of the array `iwork`,

$lwork \geq \max(1, 10n)$ .

## Output Parameters

`d, e`

On exit, `d` and `e` are overwritten.

`m`

`INTEGER`. The total number of eigenvalues found,

$0 \leq m \leq n$ . If `range = 'A'`,  $m = n$ , and if `range = 'I'`,  $m = iu - il + 1$ .

`w`

`REAL` for single precision flavors

`DOUBLE PRECISION` for double precision flavors.

Array, `DIMENSION` at least  $\max(1, n)$ .

The selected eigenvalues in ascending order, stored in `w(1)` to `w(m)`.

<i>z</i>	<small>REAL for <b>ssteqr</b> DOUBLE PRECISION for <b>dsteqr</b> COMPLEX for <b>csteqr</b> DOUBLE COMPLEX for <b>zsteqr</b>.</small> Array <i>z</i> ( <i>ldz</i> , *), the second dimension of <i>z</i> must be at least $\max(1, m)$ . If <i>jobz</i> = 'V', then if <i>info</i> = 0, the first <i>m</i> columns of <i>z</i> contain the orthonormal eigenvectors of the matrix <i>T</i> corresponding to the selected eigenvalues, with the <i>i</i> -th column of <i>z</i> holding the eigenvector associated with <i>w(i)</i> . If <i>jobz</i> = 'N', then <i>z</i> is not referenced. Note: you must ensure that at least $\max(1, m)$ columns are supplied in the array <i>z</i> ; if <i>range</i> = 'V', the exact value of <i>m</i> is not known in advance and an upper bound must be used.
<i>isuppz</i>	<small>INTEGER. Array, DIMENSION at least <math>2 * \max(1, m)</math>. The support of the eigenvectors in <i>z</i>, i.e., the indices indicating the nonzero elements in <i>z</i>. The <i>i</i>-th eigenvector is nonzero only in elements <i>isuppz</i>(<math>2i - 1</math>) through <i>isuppz</i>(<math>2i</math>).</small>
<i>work(1)</i>	<small>On exit, if <i>info</i> = 0, then <i>work(1)</i> returns the required minimal size of <i>lwork</i>.</small>
<i>iwork(1)</i>	<small>On exit, if <i>info</i> = 0, then <i>iwork(1)</i> returns the required minimal size of <i>liwork</i>.</small>
<i>info</i>	<small>INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = <i>-i</i>, the <i>i</i>th parameter had an illegal value. If <i>info</i> = 1, internal error in <b>slarre</b> occurred, If <i>info</i> = 2, internal error in <b>?larry</b> occurred.</small>

### Application Notes

Currently **?steqr** is only set up to find *all* the *n* eigenvalues and eigenvectors of *T* in  $O(n^2)$  time, that is, only *range* = 'A' is supported.

Currently the routine `?stein` is called when an appropriate  $\sigma_i$  cannot be chosen in step (c) above. `?stein` invokes modified Gram-Schmidt when eigenvalues are close.

`?steqr` works only on machines which follow IEEE-754 floating-point standard in their handling of infinities and NaNs. Normal execution of `?steqr` may create NaNs and infinities and hence may abort due to a floating point exception in environments which do not conform to the IEEE-754 standard.

---

**?pteqr**

*Computes all eigenvalues and (optionally) all eigenvectors of a real symmetric positive-definite tridiagonal matrix.*

---

```
call spteqr ( compz, n, d, e, z, ldz, work, info )
call dpteqr ( compz, n, d, e, z, ldz, work, info )
call cpteqr ( compz, n, d, e, z, ldz, work, info )
call zpteqr ( compz, n, d, e, z, ldz, work, info )
```

**Discussion**

This routine computes all the eigenvalues and (optionally) all the eigenvectors of a real symmetric positive-definite tridiagonal matrix  $T$ . In other words, the routine can compute the spectral factorization:  $T = Z\Lambda Z^T$ . Here  $\Lambda$  is a diagonal matrix whose diagonal elements are the eigenvalues  $\lambda_i$ ;  $Z$  is an orthogonal matrix whose columns are eigenvectors. Thus,

$$Tz_i = \lambda_i z_i \text{ for } i = 1, 2, \dots, n.$$

(The routine normalizes the eigenvectors so that  $\|z_i\|_2 = 1$ .)

You can also use the routine for computing the eigenvalues and eigenvectors of real symmetric (or complex Hermitian) positive-definite matrices  $A$  reduced to tridiagonal form  $T$ :  $A = QTQ^H$ . In this case, the spectral factorization is as follows:  $A = QTQ^H = (QZ)\Lambda(QZ)^H$ . Before calling **?pteqr**, you must reduce  $A$  to tridiagonal form and generate the explicit matrix  $Q$  by calling the following routines:

	for real matrices:	for complex matrices:
full storage	<b>?sytrd</b> , <b>?orgtr</b>	<b>?hetrd</b> , <b>?ungtr</b>
packed storage	<b>?sptrd</b> , <b>?opgtr</b>	<b>?hptrd</b> , <b>?upgtr</b>
band storage	<b>?sbtrd</b> ( <i>vect='V'</i> )	<b>?hbtrd</b> ( <i>vect='V'</i> )

The routine first factorizes  $T$  as  $LDL^H$  where  $L$  is a unit lower bidiagonal matrix, and  $D$  is a diagonal matrix. Then it forms the bidiagonal matrix  $B = LD^{1/2}$  and calls **?bdsqr** to compute the singular values of  $B$ , which are the same as the eigenvalues of  $T$ .

## Input Parameters

<i>compz</i>	CHARACTER*1. Must be ' <b>N</b> ' or ' <b>I</b> ' or ' <b>V</b> '. If <i>compz</i> = ' <b>N</b> ', the routine computes eigenvalues only. If <i>compz</i> = ' <b>I</b> ', the routine computes the eigenvalues and eigenvectors of the tridiagonal matrix $T$ . If <i>compz</i> = ' <b>V</b> ', the routine computes the eigenvalues and eigenvectors of $A$ (and the array <i>z</i> must contain the matrix $Q$ on entry).
<i>n</i>	INTEGER. The order of the matrix $T$ ( <i>n</i> $\geq$ 0).
<i>d, e, work</i>	REAL for single-precision flavors DOUBLE PRECISION for double-precision flavors. Arrays: <i>d</i> (*) contains the diagonal elements of $T$ . The dimension of <i>d</i> must be at least max(1, <i>n</i> ). <i>e</i> (*) contains the off-diagonal elements of $T$ . The dimension of <i>e</i> must be at least max(1, <i>n</i> -1). <i>work</i> (*) is a workspace array. The dimension of <i>work</i> must be: at least 1 if <i>compz</i> = ' <b>N</b> '; at least max(1, 4* <i>n</i> -4) if <i>compz</i> = ' <b>V</b> ' or ' <b>I</b> '.
<i>z</i>	REAL for <b>spteqr</b> DOUBLE PRECISION for <b>dpteqr</b> COMPLEX for <b>cpteqr</b> DOUBLE COMPLEX for <b>zpteqr</b> . Array, DIMENSION( <i>ldz</i> , *) If <i>compz</i> = ' <b>N</b> ' or ' <b>I</b> ', <i>z</i> need not be set. If <i>vect</i> = ' <b>V</b> ', <i>z</i> must contain the <i>n</i> by <i>n</i> matrix $Q$ . The second dimension of <i>z</i> must be: at least 1 if <i>compz</i> = ' <b>N</b> '; at least max(1, <i>n</i> ) if <i>compz</i> = ' <b>V</b> ' or ' <b>I</b> '.
<i>ldz</i>	INTEGER. The first dimension of <i>z</i> . Constraints: <i>ldz</i> $\geq$ 1 if <i>compz</i> = ' <b>N</b> '; <i>ldz</i> $\geq$ max(1, <i>n</i> ) if <i>compz</i> = ' <b>V</b> ' or ' <b>I</b> '.

## Output Parameters

<i>d</i>	The <i>n</i> eigenvalues in descending order, unless <i>info</i> > 0. See also <i>info</i> .
<i>e</i>	On exit, the array is overwritten.
<i>z</i>	If <i>info</i> = 0, contains the <i>n</i> orthonormal eigenvectors, stored by columns. (The <i>i</i> th column corresponds to the <i>i</i> th eigenvalue.)
<i>info</i>	<b>INTEGER.</b> If <i>info</i> = 0, the execution is successful. If <i>info</i> = <i>i</i> , the leading minor of order <i>i</i> (and hence <i>T</i> itself) is not positive-definite. If <i>info</i> = <i>n + i</i> , the algorithm for computing singular values failed to converge; <i>i</i> off-diagonal elements have not converged to zero. If <i>info</i> = <i>-i</i> , the <i>i</i> th parameter had an illegal value.

## Application Notes

If  $\lambda_i$  is an exact eigenvalue, and  $\mu_i$  is the corresponding computed value, then

$$|\mu_i - \lambda_i| \leq c(n)\epsilon K \lambda_i$$

where  $c(n)$  is a modestly increasing function of  $n$ ,  $\epsilon$  is the machine precision, and  $K = \|DTD\|_2 \| (DTD)^{-1} \|_2$ ,  $D$  is diagonal with  $d_{ii} = t_{ii}^{-1/2}$ .

If  $z_i$  is the corresponding exact eigenvector, and  $w_i$  is the corresponding computed vector, then the angle  $\theta(z_i, w_i)$  between them is bounded as follows:

$$\theta(u_i, w_i) \leq c(n)\epsilon K / \min_{i \neq j}(|\lambda_i - \lambda_j|/|\lambda_i + \lambda_j|).$$

Here  $\min_{i \neq j}(|\lambda_i - \lambda_j|/|\lambda_i + \lambda_j|)$  is the *relative gap* between  $\lambda_i$  and the other eigenvalues.

The total number of floating-point operations depends on how rapidly the algorithm converges. Typically, it is about

$30n^2$  if *compz* = 'N';

$6n^3$  (for complex flavors,  $12n^3$ ) if *compz* = 'V' or 'I'.

## ?stebz

*Computes selected eigenvalues of a real symmetric tridiagonal matrix by bisection.*

```
call sstebz (range, order, n, vl, vu, il, iu, abstol,
             d, e, m, nsplitt, w, iblock, isplit, work, iwork, info)
call dstebz (range, order, n, vl, vu, il, iu, abstol,
             d, e, m, nsplitt, w, iblock, isplit, work, iwork, info)
```

### Discussion

This routine computes some (or all) of the eigenvalues of a real symmetric tridiagonal matrix  $T$  by bisection. The routine searches for zero or negligible off-diagonal elements to see if  $T$  splits into block-diagonal form  $T = \text{diag}(T_1, T_2, \dots)$ . Then it performs bisection on each of the blocks  $T_i$  and returns the block index of each computed eigenvalue, so that a subsequent call to ?stein can also take advantage of the block structure.

### Input Parameters

<i>range</i>	CHARACTER*1. Must be 'A' or 'V' or 'I'. If <i>range</i> = 'A', the routine computes all eigenvalues. If <i>range</i> = 'V', the routine computes eigenvalues $\lambda_i$ in the half-open interval: $vl < \lambda_i \leq vu$ . If <i>range</i> = 'I', the routine computes eigenvalues with indices <i>il</i> to <i>iu</i> .
<i>order</i>	CHARACTER*1. Must be 'B' or 'E'. If <i>order</i> = 'B', the eigenvalues are to be ordered from smallest to largest within each split-off block. If <i>order</i> = 'E', the eigenvalues for the entire matrix are to be ordered from smallest to largest.
<i>n</i>	INTEGER. The order of the matrix $T$ ( $n \geq 0$ ).

<i>vl</i> , <i>vu</i>	<b>REAL</b> for <b>sstebz</b> <b>DOUBLE PRECISION</b> for <b>dstebz</b> . If <i>range</i> = 'V', the routine computes eigenvalues $\lambda_i$ in the half-open interval: $vl < \lambda_i \leq vu$ . If <i>range</i> = 'A' or 'I', <i>vl</i> and <i>vu</i> are not referenced.
<i>il</i> , <i>iu</i>	<b>INTEGER</b> . Constraint: $1 \leq il \leq iu \leq n$ . If <i>range</i> = 'I', the routine computes eigenvalues $\lambda_i$ such that $il \leq i \leq iu$ (assuming that the eigenvalues $\lambda_i$ are in ascending order). If <i>range</i> = 'A' or 'V', <i>il</i> and <i>iu</i> are not referenced.
<i>abstol</i>	<b>REAL</b> for <b>sstebz</b> <b>DOUBLE PRECISION</b> for <b>dstebz</b> . The absolute tolerance to which each eigenvalue is required. An eigenvalue (or cluster) is considered to have converged if it lies in an interval of width <i>abstol</i> . If <i>abstol</i> $\leq 0.0$ , then the tolerance is taken as $\epsilon \ T\ _1$ , where $\epsilon$ is the machine precision.
<i>d</i> , <i>e</i>	<b>REAL</b> for <b>sstebz</b> <b>DOUBLE PRECISION</b> for <b>dstebz</b> . Arrays: <i>d(*)</i> contains the diagonal elements of $T$ . The dimension of <i>d</i> must be at least $\max(1, n)$ . <i>e(*)</i> contains the off-diagonal elements of $T$ . The dimension of <i>e</i> must be at least $\max(1, n-1)$ .
<i>iwork</i>	<b>INTEGER</b> . Workspace. Array, <b>DIMENSION</b> at least $\max(1, 3n)$ .

## Output Parameters

<i>m</i>	<b>INTEGER</b> . The actual number of eigenvalues found.
<i>nsplit</i>	<b>INTEGER</b> . The number of diagonal blocks detected in $T$ .
<i>w</i>	<b>REAL</b> for <b>sstebz</b> <b>DOUBLE PRECISION</b> for <b>dstebz</b> . Array, <b>DIMENSION</b> at least $\max(1, n)$ . The computed eigenvalues, stored in <i>w(1)</i> to <i>w(m)</i> .

*iblock, isplit* INTEGER.

Arrays, DIMENSION at least max(1, *n*).

A positive value *iblock(i)* is the block number of the eigenvalue stored in *w(i)* (see also *info*).

The leading *nsplit* elements of *isplit* contain points at which *T* splits into blocks *T<sub>i</sub>* as follows: the block *T<sub>1</sub>* contains rows/columns 1 to *isplit(1)*; the block *T<sub>2</sub>* contains rows/columns *isplit(1)+1* to *isplit(2)*, and so on.

*info*

INTEGER.

If *info* = 0, the execution is successful.

If *info* = 1, for *range* = 'A' or 'V', the algorithm failed to compute some of the required eigenvalues to the desired accuracy; *iblock(i)<0* indicates that the eigenvalue stored in *w(i)* failed to converge.

If *info* = 2, for *range* = 'I', the algorithm failed to compute some of the required eigenvalues. Try calling the routine again with *range* = 'A'.

If *info* = 3:

for *range* = 'A' or 'V', same as *info* = 1;

for *range* = 'I', same as *info* = 2.

If *info* = 4, no eigenvalues have been computed. The floating-point arithmetic on the computer is not behaving as expected.

If *info* = -*i*, the *i*th parameter had an illegal value.

## Application Notes

The eigenvalues of *T* are computed to high relative accuracy which means that if they vary widely in magnitude, then any small eigenvalues will be computed more accurately than, for example, with the standard *QR* method. However, the reduction to tridiagonal form (prior to calling the routine) may exclude the possibility of obtaining high relative accuracy in the small eigenvalues of the original matrix if its eigenvalues vary widely in magnitude.

---

## ?stein

*Computes the eigenvectors corresponding to specified eigenvalues of a real symmetric tridiagonal matrix.*

---

```
call sstein ( n, d, e, m, w, iblock, isplit, z, ldz,
              work, iwork, ifailv, info )
call dstein ( n, d, e, m, w, iblock, isplit, z, ldz,
              work, iwork, ifailv, info )
call cstein ( n, d, e, m, w, iblock, isplit, z, ldz,
              work, iwork, ifailv, info )
call zstein ( n, d, e, m, w, iblock, isplit, z, ldz,
              work, iwork, ifailv, info )
```

### Discussion

This routine computes the eigenvectors of a real symmetric tridiagonal matrix  $T$  corresponding to specified eigenvalues, by inverse iteration. It is designed to be used in particular after the specified eigenvalues have been computed by [?stebz](#) with  $order = 'B'$ , but may also be used when the eigenvalues have been computed by other routines. If you use this routine after [?stebz](#), it can take advantage of the block structure by performing inverse iteration on each block  $T_i$  separately, which is more efficient than using the whole matrix  $T$ .

If  $T$  has been formed by reduction of a full symmetric or Hermitian matrix  $A$  to tridiagonal form, you can transform eigenvectors of  $T$  to eigenvectors of  $A$  by calling [?ormtr](#) or [?opmtr](#) (for real flavors) or by calling [?unmtr](#) or [?upmtr](#) (for complex flavors).

### Input Parameters

- |     |   |
|-----|---|
| $n$ | <b>INTEGER.</b> The order of the matrix $T$ ( $n \geq 0$ ). |
| $m$ | <b>INTEGER.</b> The number of eigenvectors to be returned.  |

---

<i>d, e, w</i>	<b>REAL</b> for single-precision flavors <b>DOUBLE PRECISION</b> for double-precision flavors. Arrays: <i>d(*)</i> contains the diagonal elements of $T$ . The dimension of <i>d</i> must be at least $\max(1, n)$ . <i>e(*)</i> contains the off-diagonal elements of $T$ . The dimension of <i>e</i> must be at least $\max(1, n-1)$ . <i>w(*)</i> contains the eigenvalues of $T$ , stored in <i>w(1)</i> to <i>w(m)</i> (as returned by <b>?stebz</b> , see <a href="#">page 5-149</a> ). Eigenvalues of $T_1$ must be supplied first, in non-decreasing order; then those of $T_2$ , again in non-decreasing order, and so on. Constraint: if <i>iblock(i) = iblock(i+1)</i> , $w(i) \leq w(i+1)$ . The dimension of <i>w</i> must be at least $\max(1, n)$ .
<i>iblock, isplit</i>	<b>INTEGER</b> . Arrays, <b>DIMENSION</b> at least $\max(1, n)$ . The arrays <i>iblock</i> and <i>isplit</i> , as returned by <b>?stebz</b> with <i>order = 'B'</i> . If you did not call <b>?stebz</b> with <i>order = 'B'</i> , set all elements of <i>iblock</i> to 1, and <i>isplit(1)</i> to <i>n</i> .
<i>ldz</i>	<b>INTEGER</b> . The first dimension of the output array <i>z</i> ; $ldz \geq \max(1, n)$ .
<i>work</i>	<b>REAL</b> for single-precision flavors <b>DOUBLE PRECISION</b> for double-precision flavors. Workspace array, <b>DIMENSION</b> at least $\max(1, 5n)$ .
<i>iwork</i>	<b>INTEGER</b> . Workspace array, <b>DIMENSION</b> at least $\max(1, n)$ .

## Output Parameters

<i>z</i>	<b>REAL</b> for <b>sstein</b> <b>DOUBLE PRECISION</b> for <b>dstein</b> <b>COMPLEX</b> for <b>cstein</b> <b>DOUBLE COMPLEX</b> for <b>zstein</b> . Array, <b>DIMENSION</b> ( <i>ldz, *</i> ).
----------	---

If  $\text{info} = 0$ ,  $\text{z}$  contains the  $m$  orthonormal eigenvectors, stored by columns. (The  $i$ th column corresponds to the  $i$ th specified eigenvalue.)

*ifailv*

**INTEGER.** Array, **DIMENSION** at least  $\max(1, m)$ .

If  $\text{info} = i > 0$ , the first  $i$  elements of *ifailv* contain the indices of any eigenvectors that failed to converge.

*info*

**INTEGER.**

If  $\text{info} = 0$ , the execution is successful.

If  $\text{info} = i$ , then  $i$  eigenvectors (as indicated by the parameter *ifailv*) each failed to converge in 5 iterations. The current iterates are stored in the corresponding columns of the array *z*.

If  $\text{info} = -i$ , the  $i$ th parameter had an illegal value.

## Application Notes

Each computed eigenvector  $z_i$  is an exact eigenvector of a matrix  $T + E_i$ , where  $\|E_i\|_2 = O(\epsilon) \|T\|_2$ . However, a set of eigenvectors computed by this routine may not be orthogonal to so high a degree of accuracy as those computed by [?steqr](#).

---

## ?disna

*Computes the reciprocal condition numbers for the eigenvectors of a symmetric/ Hermitian matrix or for the left or right singular vectors of a general matrix.*

---

```
call sdisna (job, m, n, d, sep, info)
call ddisna (job, m, n, d, sep, info)
```

## Discussion

This routine computes the reciprocal condition numbers for the eigenvectors of a real symmetric or complex Hermitian matrix or for the left or right singular vectors of a general  $m$ -by- $n$  matrix.

The reciprocal condition number is the 'gap' between the corresponding eigenvalue or singular value and the nearest other one.

The bound on the error, measured by angle in radians, in the  $i$ -th computed vector is given by

```
slamch( 'E' ) * ( anorm / sep(i) )
```

where  $\text{anorm} = \|A\|_2 = \max(|d(j)|)$ .  $\text{sep}(i)$  is not allowed to be smaller than  $\text{slamch}( 'E' ) * \text{anorm}$  in order to limit the size of the error bound.

?disna may also be used to compute error bounds for eigenvectors of the generalized symmetric definite eigenproblem.

### Input Parameters

*job*            CHARACTER\*1. Must be 'E', 'L', or 'R'.

Specifies for which problem the reciprocal condition numbers should be computed:

*job*='E': for the eigenvectors of a symmetric/Hermitian matrix ;

*job*='L': for the left singular vectors of a general matrix;

*job*='R': for the right singular vectors of a general matrix .

*m*            INTEGER. The number of rows of the matrix ( $m \geq 0$ ).

*n*            INTEGER. If *job*='L', or 'R', the number of columns of the matrix ( $n \geq 0$ ). Ignored if *job*='E'.

*d*            REAL for sdisna

DOUBLE PRECISION for ddisna.

Array, dimension at least  $\max(1, m)$  if *job*='E', and at least  $\max(1, \min(m, n))$  if *job*='L' or 'R'.

This array must contain the eigenvalues (if *job*='E') or singular values (if *job*='L' or 'R') of the matrix, in either increasing or decreasing order. If singular values, they must be non-negative.

## Output Parameters

*sep*            REAL for `sdisna`  
DOUBLE PRECISION for `ddisna`.  
Array, dimension at least  $\max(1, m)$  if  $job = 'E'$ , and at  
least  $\max(1, \min(m, n))$  if  $job = 'L'$  or  $'R'$ .  
The reciprocal condition numbers of the vectors.

*info*            INTEGER.  
If  $info = 0$ , the execution is successful.  
If  $info = -i$ , the  $i$ th parameter had an illegal value.

## Generalized Symmetric-Definite Eigenvalue Problems

*Generalized symmetric-definite eigenvalue problems* are as follows: find the eigenvalues  $\lambda$  and the corresponding eigenvectors  $z$  that satisfy one of these equations:

$$Az = \lambda Bz, \quad ABz = \lambda z, \quad \text{or} \quad BAz = \lambda z$$

where  $A$  is an  $n$  by  $n$  symmetric or Hermitian matrix, and  $B$  is an  $n$  by  $n$  symmetric positive-definite or Hermitian positive-definite matrix.

In these problems, there exist  $n$  real eigenvectors corresponding to real eigenvalues (even for complex Hermitian matrices  $A$  and  $B$ ).

Routines described in this section allow you to reduce the above generalized problems to standard symmetric eigenvalue problem  $Cy = \lambda y$ , which you can solve by calling LAPACK routines described earlier in this chapter (see [page 5-101](#)).

Different routines allow the matrices to be stored either conventionally or in packed storage. Prior to reduction, the positive-definite matrix  $B$  must first be factorized using either [?potrf](#) or [?pptrf](#).

The reduction routine for the banded matrices  $A$  and  $B$  uses a split Cholesky factorization for which a specific routine [?pbstf](#) is provided. This refinement halves the amount of work required to form matrix  $C$ .

**Table 5-4 Computational Routines for Reducing Generalized Eigenproblems to Standard Problems**

Matrix type	Reduce to standard problems (full storage)	Reduce to standard problems (packed storage)	Reduce to standard problems (band matrices)	Factorize band matrix
real symmetric matrices	<a href="#">?sygst</a>	<a href="#">?spgst</a>	<a href="#">?sbgst</a>	<a href="#">?pbstf</a>
complex Hermitian matrices	<a href="#">?hegst</a> / <a href="#">?hpbst</a>	<a href="#">?hpgst</a>	<a href="#">?hbgst</a>	<a href="#">?pbstf</a>

---

## ?sygst

*Reduces a real symmetric-definite generalized eigenvalue problem to the standard form.*

---

```
call ssygst ( itype, uplo, n, a, lda, b, ldb, info )
call dsygst ( itype, uplo, n, a, lda, b, ldb, info )
```

### Discussion

This routine reduces real symmetric-definite generalized eigenproblems

$$Az = \lambda Bz, \quad ABz = \lambda z, \quad \text{or} \quad BAz = \lambda z$$

to the standard form  $Cy = \lambda y$ . Here  $A$  is a real symmetric matrix, and  $B$  is a real symmetric positive-definite matrix. Before calling this routine, call [?potrf](#) to compute the Cholesky factorization:  $B = U^T U$  or  $B = LL^T$  (see [page 4-14](#)).

### Input Parameters

*itype*

**INTEGER.** Must be 1 or 2 or 3.

If *itype* = 1, the generalized eigenproblem is  $Az = \lambda Bz$ ;  
for *uplo* = 'U':  $C = U^{-T}AU^{-1}$ ,  $z = U^{-1}y$ ;  
for *uplo* = 'L':  $C = L^{-1}AL^{-T}$ ,  $z = L^{-T}y$ .

If *itype* = 2, the generalized eigenproblem is  $ABz = \lambda z$ ;  
for *uplo* = 'U':  $C = UAU^T$ ,  $z = U^{-1}y$ ;  
for *uplo* = 'L':  $C = L^TAL$ ,  $z = L^{-T}y$ .

If *itype* = 3, the generalized eigenproblem is  $BAz = \lambda z$ ;  
for *uplo* = 'U':  $C = UAU^T$ ,  $z = U^Ty$ ;  
for *uplo* = 'L':  $C = L^TAL$ ,  $z = Ly$ .

*uplo*

**CHARACTER\*1.** Must be 'U' or 'L'.

If *uplo* = 'U', the array *a* stores the upper triangle of  $A$ ;  
you must supply  $B$  in the factored form  $B = U^T U$ .

If *uplo* = 'L', the array *a* stores the lower triangle of  $A$ ;  
you must supply  $B$  in the factored form  $B = LL^T$ .

*n*

**INTEGER.** The order of the matrices  $A$  and  $B$  (*n*  $\geq 0$ ).

*a, b*      REAL for ssygst  
DOUBLE PRECISION for dsygst.  
 Arrays:  
*a( lda, \* )* contains the upper or lower triangle of  $A$ .  
 The second dimension of *a* must be at least  $\max(1, n)$ .  
*b( ldb, \* )* contains the Cholesky-factored matrix  $B$ :  
 $B = U^T U$  or  $B = LL^T$  (as returned by ?potrf).  
 The second dimension of *b* must be at least  $\max(1, n)$ .  
*lda*      INTEGER. The first dimension of *a*; at least  $\max(1, n)$ .  
*ldb*      INTEGER. The first dimension of *b*; at least  $\max(1, n)$ .

## Output Parameters

*a*      The upper or lower triangle of  $A$  is overwritten by the upper or lower triangle of  $C$ , as specified by the arguments *itype* and *uplo*.  
*info*      INTEGER.  
 If *info* = 0, the execution is successful.  
 If *info* =  $-i$ , the *i*th parameter had an illegal value.

## Application Notes

Forming the reduced matrix  $C$  is a stable procedure. However, it involves implicit multiplication by  $B^{-1}$  (if *itype* = 1) or  $B$  (if *itype* = 2 or 3). When the routine is used as a step in the computation of eigenvalues and eigenvectors of the original problem, there may be a significant loss of accuracy if  $B$  is ill-conditioned with respect to inversion.

The approximate number of floating-point operations is  $n^3$ .

## ?hegst

*Reduces a complex Hermitian-definite generalized eigenvalue problem to the standard form.*

---

```
call chegst ( itype, uplo, n, a, lda, b, ldb, info )
call zhegst ( itype, uplo, n, a, lda, b, ldb, info )
```

### Discussion

This routine reduces complex Hermitian-definite generalized eigenvalue problems

$$Az = \lambda Bz, \quad ABz = \lambda z, \quad \text{or} \quad BAz = \lambda z$$

to the standard form  $Cy = \lambda y$ . Here the matrix  $A$  is complex Hermitian, and  $B$  is complex Hermitian positive-definite. Before calling this routine, you must call [?potrf](#) to compute the Cholesky factorization:  $B = U^H U$  or  $B = LL^H$  (see [page 4-14](#)).

### Input Parameters

*itype*

INTEGER. Must be 1 or 2 or 3.

If *itype* = 1, the generalized eigenproblem is  $Az = \lambda Bz$ ;  
for *uplo* = 'U':  $C = U^{-H}AU^{-1}$ ,  $z = U^{-1}y$ ;  
for *uplo* = 'L':  $C = L^{-1}AL^{-H}$ ,  $z = L^{-H}y$ .

If *itype* = 2, the generalized eigenproblem is  $ABz = \lambda z$ ;  
for *uplo* = 'U':  $C = UAU^H$ ,  $z = U^{-1}y$ ;  
for *uplo* = 'L':  $C = L^HAL$ ,  $z = L^{-H}y$ .

If *itype* = 3, the generalized eigenproblem is  $BAz = \lambda z$ ;  
for *uplo* = 'U':  $C = UAU^H$ ,  $z = U^H y$ ;  
for *uplo* = 'L':  $C = L^HAL$ ,  $z = Ly$ .

*uplo*

CHARACTER\*1. Must be 'U' or 'L'.

If *uplo* = 'U', the array *a* stores the upper triangle of  $A$ ;  
you must supply  $B$  in the factored form  $B = U^H U$ .

If *uplo* = 'L', the array *a* stores the lower triangle of  $A$ ;  
you must supply  $B$  in the factored form  $B = LL^H$ .

---

<i>n</i>	<code>INTEGER</code> . The order of the matrices <i>A</i> and <i>B</i> ( <i>n</i> $\geq 0$ ).
<i>a, b</i>	<code>COMPLEX</code> for <code>chegst</code> <code>DOUBLE COMPLEX</code> for <code>zhegst</code> .
	Arrays: <i>a</i> ( <i>lda</i> ,*) contains the upper or lower triangle of <i>A</i> . The second dimension of <i>a</i> must be at least $\max(1, n)$ .
	<i>b</i> ( <i>ldb</i> ,*) contains the Cholesky-factored matrix <i>B</i> : $B = U^H U$ or $B = LL^H$ (as returned by <code>?potrf</code> ). The second dimension of <i>b</i> must be at least $\max(1, n)$ .
<i>lda</i>	<code>INTEGER</code> . The first dimension of <i>a</i> ; at least $\max(1, n)$ .
<i>ldb</i>	<code>INTEGER</code> . The first dimension of <i>b</i> ; at least $\max(1, n)$ .

## Output Parameters

<i>a</i>	The upper or lower triangle of <i>A</i> is overwritten by the upper or lower triangle of <i>C</i> , as specified by the arguments <i>itype</i> and <i>uplo</i> .
<i>info</i>	<code>INTEGER</code> . If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> th parameter had an illegal value.

## Application Notes

Forming the reduced matrix *C* is a stable procedure. However, it involves implicit multiplication by  $B^{-1}$  (if *itype* = 1) or *B* (if *itype* = 2 or 3). When the routine is used as a step in the computation of eigenvalues and eigenvectors of the original problem, there may be a significant loss of accuracy if *B* is ill-conditioned with respect to inversion.

The approximate number of floating-point operations is  $n^3$ .

---

## ?spgst

*Reduces a real symmetric-definite generalized eigenvalue problem to the standard form using packed storage.*

---

```
call sspgst ( itype, uplo, n, ap, bp, info )
call dspgst ( itype, uplo, n, ap, bp, info )
```

### Discussion

This routine reduces real symmetric-definite generalized eigenproblems

$$Az = \lambda Bz, \quad ABz = \lambda z, \quad \text{or} \quad BAz = \lambda z$$

to the standard form  $Cy = \lambda y$ , using packed matrix storage. Here  $A$  is a real symmetric matrix, and  $B$  is a real symmetric positive-definite matrix.

Before calling this routine, call [?pptrf](#) to compute the Cholesky factorization:  $B = U^T U$  or  $B = LL^T$  (see [page 4-16](#)).

### Input Parameters

*itype*

**INTEGER.** Must be 1 or 2 or 3.

If *itype* = 1, the generalized eigenproblem is  $Az = \lambda Bz$ ;  
for *uplo* = 'U':  $C = U^{-T}AU^{-1}$ ,  $z = U^{-1}y$ ;  
for *uplo* = 'L':  $C = L^{-1}AL^{-T}$ ,  $z = L^{-T}y$ .

If *itype* = 2, the generalized eigenproblem is  $ABz = \lambda z$ ;  
for *uplo* = 'U':  $C = UAU^T$ ,  $z = U^{-1}y$ ;  
for *uplo* = 'L':  $C = L^TAL$ ,  $z = L^{-T}y$ .

If *itype* = 3, the generalized eigenproblem is  $BAz = \lambda z$ ;  
for *uplo* = 'U':  $C = UAU^T$ ,  $z = U^Ty$ ;  
for *uplo* = 'L':  $C = L^TAL$ ,  $z = Ly$ .

*uplo*

**CHARACTER\*1.** Must be 'U' or 'L'.

If *uplo* = 'U', *ap* stores the packed upper triangle of  $A$ ;  
you must supply  $B$  in the factored form  $B = U^T U$ .

If *uplo* = 'L', *ap* stores the packed lower triangle of  $A$ ;  
you must supply  $B$  in the factored form  $B = LL^T$ .

*n*

**INTEGER.** The order of the matrices  $A$  and  $B$  ( $n \geq 0$ ).

*ap, bp*      **REAL** for **sspgst**  
**DOUBLE PRECISION** for **dspgst**.  
 Arrays:  
*ap*(\*) contains the packed upper or lower triangle of  $A$ .  
 The dimension of *ap* must be at least  $\max(1, n*(n+1)/2)$ .  
*bp*(\*) contains the packed Cholesky factor of  $B$   
 (as returned by **?ptrf** with the same *uplo* value).  
 The dimension of *bp* must be at least  $\max(1, n*(n+1)/2)$ .

### Output Parameters

*ap*      The upper or lower triangle of  $A$  is overwritten by the  
 upper or lower triangle of  $C$ , as specified by the  
 arguments *itype* and *uplo*.  
*info*      **INTEGER**.  
 If *info* = 0, the execution is successful.  
 If *info* = *-i*, the *i*th parameter had an illegal value.

### Application Notes

Forming the reduced matrix  $C$  is a stable procedure. However, it involves implicit multiplication by  $B^{-1}$  (if *itype* = 1) or  $B$  (if *itype* = 2 or 3). When the routine is used as a step in the computation of eigenvalues and eigenvectors of the original problem, there may be a significant loss of accuracy if  $B$  is ill-conditioned with respect to inversion.

The approximate number of floating-point operations is  $n^3$ .

## ?hpgst

*Reduces a complex Hermitian-definite generalized eigenvalue problem to the standard form using packed storage.*

---

```
call chpgst ( itype, uplo, n, ap, bp, info )
call zhpgst ( itype, uplo, n, ap, bp, info )
```

### Discussion

This routine reduces real symmetric-definite generalized eigenproblems

$$Az = \lambda Bz, \quad ABz = \lambda z, \quad \text{or} \quad BAz = \lambda z$$

to the standard form  $Cy = \lambda y$ , using packed matrix storage. Here  $A$  is a real symmetric matrix, and  $B$  is a real symmetric positive-definite matrix.

Before calling this routine, you must call ?[pptrf](#) to compute the Cholesky factorization:  $B = U^H U$  or  $B = LL^H$  (see [page 4-16](#)).

### Input Parameters

*itype*

INTEGER. Must be 1 or 2 or 3.

If *itype* = 1, the generalized eigenproblem is  $Az = \lambda Bz$ ;  
for *uplo* = 'U':  $C = U^{-H}AU^{-1}$ ,  $z = U^{-1}y$ ;  
for *uplo* = 'L':  $C = L^{-1}AL^{-H}$ ,  $z = L^{-H}y$ .

If *itype* = 2, the generalized eigenproblem is  $ABz = \lambda z$ ;  
for *uplo* = 'U':  $C = UAU^H$ ,  $z = U^{-1}y$ ;  
for *uplo* = 'L':  $C = L^HAL$ ,  $z = L^{-H}y$ .

If *itype* = 3, the generalized eigenproblem is  $BAz = \lambda z$ ;  
for *uplo* = 'U':  $C = UAU^H$ ,  $z = U^Hy$ ;  
for *uplo* = 'L':  $C = L^HAL$ ,  $z = Ly$ .

*uplo*

CHARACTER\*1. Must be 'U' or 'L'.

If *uplo* = 'U', *ap* stores the packed upper triangle of  $A$ ;  
you must supply  $B$  in the factored form  $B = U^H U$ .

If *uplo* = 'L', *ap* stores the packed lower triangle of  $A$ ;  
you must supply  $B$  in the factored form  $B = LL^H$ .

*n*

INTEGER. The order of the matrices  $A$  and  $B$  ( $n \geq 0$ ).

*ap, bp*      COMPLEX for `chpgst`  
                 DOUBLE COMPLEX for `zhpgst`.  
 Arrays:  
*ap(\*)* contains the packed upper or lower triangle of  $A$ .  
 The dimension of *a* must be at least  $\max(1, n^*(n+1)/2)$ .  
*bp(\*)* contains the packed Cholesky factor of  $B$   
 (as returned by `?ptrf` with the same *uplo* value).  
 The dimension of *b* must be at least  $\max(1, n^*(n+1)/2)$ .

## Output Parameters

*ap*      The upper or lower triangle of  $A$  is overwritten by the  
                 upper or lower triangle of  $C$ , as specified by the  
                 arguments *itype* and *uplo*.  
*info*      INTEGER.  
                 If *info* = 0, the execution is successful.  
                 If *info* = *-i*, the *i*th parameter had an illegal value.

## Application Notes

Forming the reduced matrix  $C$  is a stable procedure. However, it involves implicit multiplication by  $B^{-1}$  (if *itype* = 1) or  $B$  (if *itype* = 2 or 3). When the routine is used as a step in the computation of eigenvalues and eigenvectors of the original problem, there may be a significant loss of accuracy if  $B$  is ill-conditioned with respect to inversion.

The approximate number of floating-point operations is  $n^3$ .

## ?sbgst

*Reduces a real symmetric-definite generalized eigenproblem for banded matrices to the standard form using the factorization performed by ?pbstf.*

```
call ssbgst ( vect, uplo, n, ka, kb, ab, ldab, bb, ldbb, x, ldx,
              work, info )
call dsbgst ( vect, uplo, n, ka, kb, ab, ldab, bb, ldbb, x, ldx,
              work, info )
```

### Discussion

To reduce the real symmetric-definite generalized eigenproblem  $Az = \lambda Bz$  to the standard form  $Cy = \lambda y$ , where  $A$ ,  $B$  and  $C$  are banded, this routine must be preceded by a call to [spbstf/dpbstf](#), which computes the split Cholesky factorization of the positive-definite matrix  $B$ :  $B = S^T S$ . The split Cholesky factorization, compared with the ordinary Cholesky factorization, allows the work to be approximately halved.

This routine overwrites  $A$  with  $C = X^T A X$ , where  $X = S^{-1} Q$  and  $Q$  is an orthogonal matrix chosen (implicitly) to preserve the bandwidth of  $A$ .

The routine also has an option to allow the accumulation of  $X$ , and then, if  $z$  is an eigenvector of  $C$ ,  $Xz$  is an eigenvector of the original system.

### Input Parameters

<i>vect</i>	CHARACTER*1. Must be ' <b>N</b> ' or ' <b>V</b> '. If <i>vect</i> = ' <b>N</b> ', then matrix $X$ is not returned; If <i>vect</i> = ' <b>V</b> ', then matrix $X$ is returned.
<i>uplo</i>	CHARACTER*1. Must be ' <b>U</b> ' or ' <b>L</b> '. If <i>uplo</i> = ' <b>U</b> ', <i>ab</i> stores the upper triangular part of $A$ . If <i>uplo</i> = ' <b>L</b> ', <i>ab</i> stores the lower triangular part of $A$ .
<i>n</i>	INTEGER. The order of the matrices $A$ and $B$ ( <i>n</i> ≥ 0).
<i>ka</i>	INTEGER. The number of super- or sub-diagonals in $A$ ( <i>ka</i> ≥ 0).

---

<i>kb</i>	<code>INTEGER</code> . The number of super- or sub-diagonals in $B$ ( $ka \geq kb \geq 0$ ).
<i>ab, bb, work</i>	<code>REAL</code> for <code>ssbgst</code> <code>DOUBLE PRECISION</code> for <code>dsbgst</code> <i>ab</i> ( <i>ldab</i> , *) is an array containing either upper or lower triangular part of the symmetric matrix $A$ (as specified by <i>uplo</i> ) in band storage format. The second dimension of the array <i>ab</i> must be at least $\max(1, n)$ . <i>bb</i> ( <i>ldbb</i> , *) is an array containing the banded split Cholesky factor of $B$ as specified by <i>uplo</i> , <i>n</i> and <i>kb</i> and returned by <code>s pbstf/dpbstf</code> . The second dimension of the array <i>bb</i> must be at least $\max(1, n)$ . <i>work</i> (*) is a workspace array, <code>DIMENSION</code> at least $\max(1, 2*n)$
<i>ldab</i>	<code>INTEGER</code> . The first dimension of the array <i>ab</i> ; must be at least <i>ka</i> +1.
<i>ldbb</i>	<code>INTEGER</code> . The first dimension of the array <i>bb</i> ; must be at least <i>kb</i> +1.
<i>lidx</i>	The first dimension of the output array <i>x</i> . Constraints: if <i>vect</i> = 'N' , then <i>lidx</i> $\geq 1$ ; if <i>vect</i> = 'V' , then <i>lidx</i> $\geq \max(1, n)$ .

## Output Parameters

<i>ab</i>	On exit, this array is overwritten by the upper or lower triangle of $C$ as specified by <i>uplo</i> .
<i>x</i>	<code>REAL</code> for <code>ssbgst</code> <code>DOUBLE PRECISION</code> for <code>dsbgst</code> Array. If <i>vect</i> = 'V' , then <i>x</i> ( <i>lidx</i> , *) contains the <i>n</i> by <i>n</i> matrix $X = S^{-1}Q$ . If <i>vect</i> = 'N' , then <i>x</i> is not referenced. The second dimension of <i>x</i> must be: at least $\max(1, n)$ , if <i>vect</i> = 'V'; at least 1, if <i>vect</i> = 'N'.

*info*

INTEGER.

If *info* = 0, the execution is successful.If *info* = -*i*, the *i*th parameter had an illegal value.

### Application Notes

Forming the reduced matrix  $C$  involves implicit multiplication by  $B^{-1}$ . When the routine is used as a step in the computation of eigenvalues and eigenvectors of the original problem, there may be a significant loss of accuracy if  $B$  is ill-conditioned with respect to inversion.

The total number of floating-point operations is approximately  $6n^2 * kb$ , when *vect* = 'N'. Additional  $(3/2)n^3 * (kb/ka)$  operations are required when *vect* = 'V'. All these estimates assume that both *ka* and *kb* are much less than *n*.

## ?hbgst

*Reduces a complex Hermitian-definite generalized eigenproblem for banded matrices to the standard form using the factorization performed by ?pbstf.*

```
call chbgst ( vect, uplo, n, ka, kb, ab, ldab, bb, ldbb, x, ldx,
              work, rwork, info )
call zhbgst ( vect, uplo, n, ka, kb, ab, ldab, bb, ldbb, x, ldx,
              work, rwork, info )
```

### Discussion

To reduce the complex Hermitian-definite generalized eigenproblem  $Az = \lambda Bz$  to the standard form  $Cy = \lambda y$ , where  $A$ ,  $B$  and  $C$  are banded, this routine must be preceded by a call to [cpbstf/zpbstf](#), which computes the split Cholesky factorization of the positive-definite matrix  $B$ :  $B = S^H S$ . The split Cholesky factorization, compared with the ordinary Cholesky factorization, allows the work to be approximately halved.

This routine overwrites  $A$  with  $C = X^H A X$ , where  $X = S^{-1} Q$  and  $Q$  is a unitary matrix chosen (implicitly) to preserve the bandwidth of  $A$ .

The routine also has an option to allow the accumulation of  $X$ , and then, if  $z$  is an eigenvector of  $C$ ,  $Xz$  is an eigenvector of the original system.

### Input Parameters

<i>vect</i>	CHARACTER*1. Must be ' <b>N</b> ' or ' <b>V</b> '. If <i>vect</i> = ' <b>N</b> ', then matrix $X$ is not returned; If <i>vect</i> = ' <b>V</b> ', then matrix $X$ is returned.
<i>uplo</i>	CHARACTER*1. Must be ' <b>U</b> ' or ' <b>L</b> '. If <i>uplo</i> = ' <b>U</b> ', <i>ab</i> stores the upper triangular part of $A$ . If <i>uplo</i> = ' <b>L</b> ', <i>ab</i> stores the lower triangular part of $A$ .
<i>n</i>	INTEGER. The order of the matrices $A$ and $B$ ( <i>n</i> $\geq 0$ ).
<i>ka</i>	INTEGER. The number of super- or sub-diagonals in $A$ ( <i>ka</i> $\geq 0$ ).

<i>kb</i>	<i>INTEGER</i> . The number of super- or sub-diagonals in <i>B</i> ( $ka \geq kb \geq 0$ ).
<i>ab, bb, work</i>	<i>COMPLEX</i> for <i>chbgst</i> <i>DOUBLE COMPLEX</i> for <i>zhbgst</i> <i>ab</i> ( <i>ldab</i> , *) is an array containing either upper or lower triangular part of the Hermitian matrix <i>A</i> (as specified by <i>uplo</i> ) in band storage format. The second dimension of the array <i>ab</i> must be at least $\max(1, n)$ . <i>bb</i> ( <i>ldbb</i> , *) is an array containing the banded split Cholesky factor of <i>B</i> as specified by <i>uplo</i> , <i>n</i> and <i>kb</i> and returned by <i>cpbstf/zpbstf</i> . The second dimension of the array <i>bb</i> must be at least $\max(1, n)$ . <i>work</i> (*) is a workspace array, <i>DIMENSION</i> at least $\max(1, n)$
<i>ldab</i>	<i>INTEGER</i> . The first dimension of the array <i>ab</i> ; must be at least <i>ka</i> +1.
<i>ldbb</i>	<i>INTEGER</i> . The first dimension of the array <i>bb</i> ; must be at least <i>kb</i> +1.
<i>lidx</i>	The first dimension of the output array <i>x</i> . Constraints: if <i>vect</i> = 'N' , then <i>lidx</i> $\geq 1$ ; if <i>vect</i> = 'V' , then <i>lidx</i> $\geq \max(1, n)$ .
<i>rwork</i>	<i>REAL</i> for <i>chbgst</i> <i>DOUBLE PRECISION</i> for <i>zhbgst</i> Workspace array, <i>DIMENSION</i> at least $\max(1, n)$

## Output Parameters

<i>ab</i>	On exit, this array is overwritten by the upper or lower triangle of <i>C</i> as specified by <i>uplo</i> .
<i>x</i>	<i>COMPLEX</i> for <i>chbgst</i> <i>DOUBLE COMPLEX</i> for <i>zhbgst</i> Array. If <i>vect</i> = 'V' , then <i>x</i> ( <i>lidx</i> , *) contains the <i>n</i> by <i>n</i> matrix $X = S^{-1}Q$ . If <i>vect</i> = 'N' , then <i>x</i> is not referenced.

The second dimension of  $x$  must be:

at least  $\max(1, n)$ , if  $\text{vect} = 'V'$ ;

at least 1, if  $\text{vect} = 'N'$ .

*info*

INTEGER.

If  $\text{info} = 0$ , the execution is successful.

If  $\text{info} = -i$ , the  $i$ th parameter had an illegal value.

## Application Notes

Forming the reduced matrix  $C$  involves implicit multiplication by  $B^{-1}$ . When the routine is used as a step in the computation of eigenvalues and eigenvectors of the original problem, there may be a significant loss of accuracy if  $B$  is ill-conditioned with respect to inversion.

The total number of floating-point operations is approximately  $20n^2 * kb$ , when  $\text{vect} = 'N'$ . Additional  $5n^3 * (kb/ka)$  operations are required when  $\text{vect} = 'V'$ . All these estimates assume that both  $ka$  and  $kb$  are much less than  $n$ .

## ?pbstf

*Computes a split Cholesky factorization  
of a real symmetric or complex  
Hermitian positive-definite banded  
matrix used in ?sbgst/?hbgst .*

---

```
call spbstf ( uplo, n, kb, bb, ldbb, info )
call dpbstf ( uplo, n, kb, bb, ldbb, info )
call cpbstf ( uplo, n, kb, bb, ldbb, info )
call zpbstf ( uplo, n, kb, bb, ldbb, info )
```

### Discussion

This routine computes a split Cholesky factorization of a real symmetric or complex Hermitian positive-definite band matrix  $B$ . It is to be used in conjunction with ?sbgst/?hbgst.

The factorization has the form  $B = S^T S$  (or  $B = S^H S$  for complex flavors), where  $S$  is a band matrix of the same bandwidth as  $B$  and the following structure:  $S$  is upper triangular in the first  $(n+kb)/2$  rows and lower triangular in the remaining rows.

### Input Parameters

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. If <i>uplo</i> = 'U', <i>bb</i> stores the upper triangular part of $B$ . If <i>uplo</i> = 'L', <i>bb</i> stores the lower triangular part of $B$ .
<i>n</i>	INTEGER. The order of the matrix $B$ ( $n \geq 0$ ).
<i>kb</i>	INTEGER. The number of super- or sub-diagonals in $B$ ( $kb \geq 0$ ).
<i>bb</i>	REAL for <b>spbstf</b> DOUBLE PRECISION for <b>dpbstf</b> COMPLEX for <b>cpbstf</b> DOUBLE COMPLEX for <b>zpbstf</b> . <i>bb</i> ( <i>ldb</i> , *) is an array containing either upper or lower triangular part of the matrix $B$ (as specified by

`uplo`) in band storage format.

The second dimension of the array `bb` must be at least  $\max(1, n)$ .

`ldb` **INTEGER.** The first dimension of `bb`; must be at least  $kb+1$ .

## Output Parameters

`bb` On exit, this array is overwritten by the elements of the split Cholesky factor  $S$ .

`info` **INTEGER.**  
 If `info` = 0, the execution is successful.  
 If `info` =  $i$ , then the factorization could not be completed, because the updated element  $b_{ii}$  would be the square root of a negative number; hence the matrix  $B$  is not positive-definite.  
 If `info` =  $-i$ , the  $i$ th parameter had an illegal value.

## Application Notes

The computed factor  $S$  is the exact factor of a perturbed matrix  $B + E$ , where

$$|E| \leq c(kb + 1)\epsilon|S^H||S|, \quad |e_{ij}| \leq c(kb + 1)\epsilon\sqrt{b_{ii}b_{jj}}$$

$c(n)$  is a modest linear function of  $n$ , and  $\epsilon$  is the machine precision.

The total number of floating-point operations for real flavors is approximately  $n(kb+1)^2$ . The number of operations for complex flavors is 4 times greater. All these estimates assume that  $kb$  is much less than  $n$ .

After calling this routine, you can call [?sbgst/?hbgst](#) to solve the generalized eigenproblem  $Az = \lambda Bz$ , where  $A$  and  $B$  are banded and  $B$  is positive-definite.

## Nonsymmetric Eigenvalue Problems

This section describes LAPACK routines for solving nonsymmetric eigenvalue problems, computing the Schur factorization of general matrices, as well as performing a number of related computational tasks.

A *nonsymmetric eigenvalue problem* is as follows: given a nonsymmetric (or non-Hermitian) matrix  $A$ , find the *eigenvalues*  $\lambda$  and the corresponding *eigenvectors*  $z$  that satisfy the equation

$$Az = \lambda z \text{ (right eigenvectors } z\text{)}$$

or the equation

$$z^H A = \lambda z^H \text{ (left eigenvectors } z\text{).}$$

Nonsymmetric eigenvalue problems have the following properties:

- The number of eigenvectors may be less than the matrix order (but is not less than the number of *distinct eigenvalues* of  $A$ ).
- Eigenvalues may be complex even for a real matrix  $A$ .
- If a real nonsymmetric matrix has a complex eigenvalue  $a+bi$  corresponding to an eigenvector  $z$ , then  $a-bi$  is also an eigenvalue. The eigenvalue  $a-bi$  corresponds to the eigenvector whose elements are complex conjugate to the elements of  $z$ .

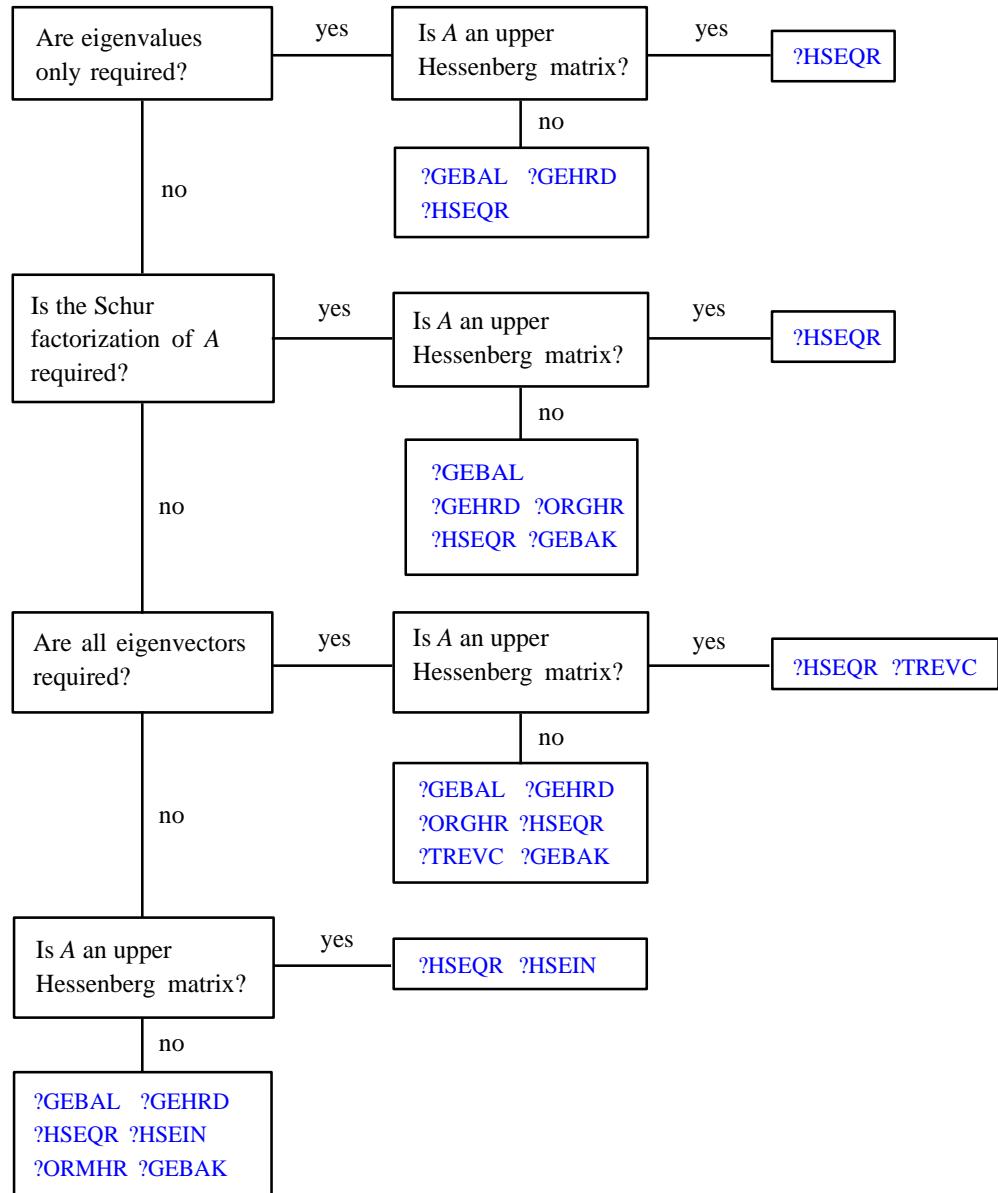
To solve a nonsymmetric eigenvalue problem with LAPACK, you usually need to reduce the matrix to the upper Hessenberg form and then solve the eigenvalue problem with the Hessenberg matrix obtained. [Table 5-5](#) lists LAPACK routines for reducing the matrix to the upper Hessenberg form by an orthogonal (or unitary) similarity transformation  $A = QHQ^H$  as well as routines for solving eigenvalue problems with Hessenberg matrices, forming the Schur factorization of such matrices and computing the corresponding condition numbers.

Decision tree in [Figure 5-4](#) helps you choose the right routine or sequence of routines for an eigenvalue problem with a real nonsymmetric matrix.

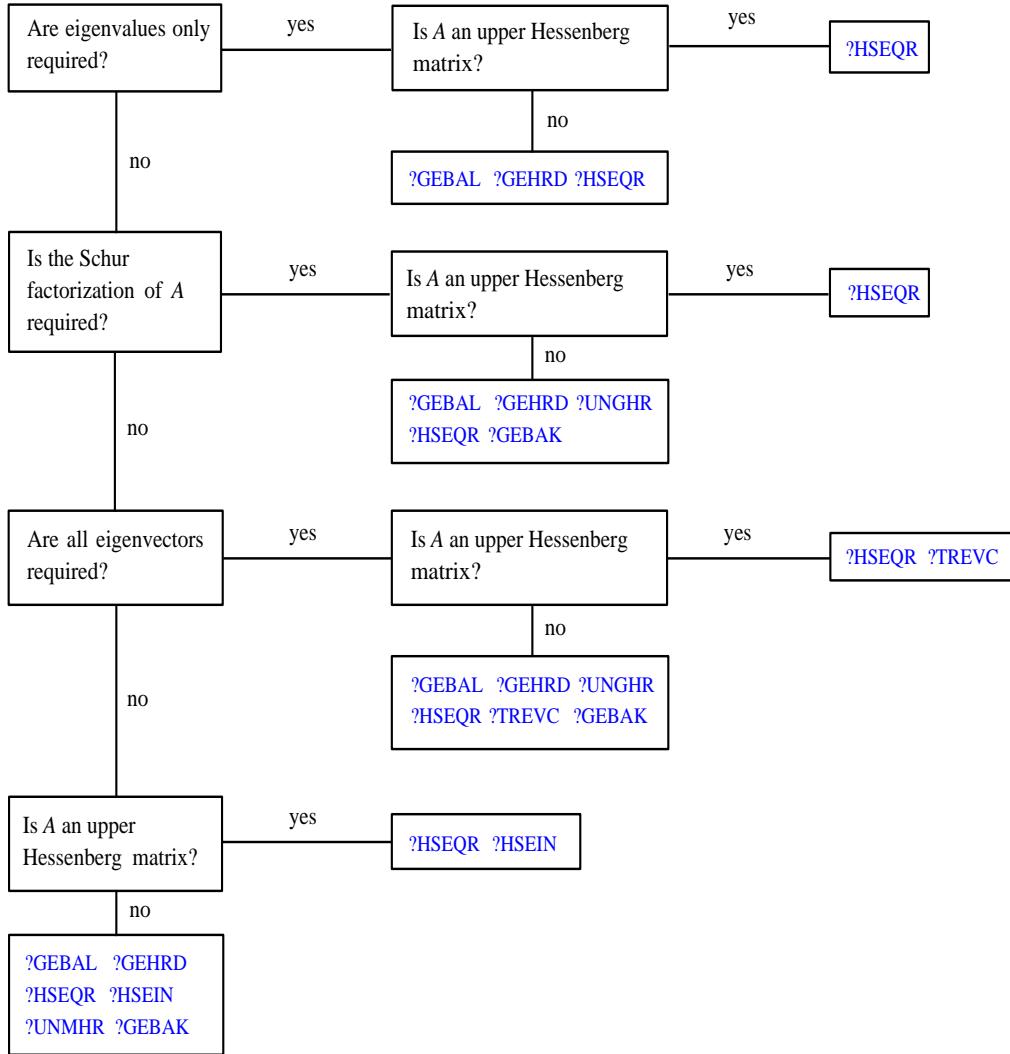
If you need to solve an eigenvalue problem with a complex non-Hermitian matrix, use the decision tree shown in [Figure 5-5](#).

**Table 5-5 Computational Routines for Solving Nonsymmetric Eigenvalue Problems**

Operation performed	Routines for real matrices	Routines for complex matrices
Reduce to Hessenberg form $A = QHQ^H$	<a href="#">?gehrd</a> ,	<a href="#">?gehrd</a>
Generate the matrix $Q$	<a href="#">?orghr</a>	<a href="#">?unghr</a>
Apply the matrix $Q$	<a href="#">?ormhr</a>	<a href="#">?unmhr</a>
Balance matrix	<a href="#">?gebal</a>	<a href="#">?gebal</a>
Transform eigenvectors of balanced matrix to those of the original matrix	<a href="#">?gebak</a>	<a href="#">?gebak</a>
Find eigenvalues and Schur factorization (QR algorithm)	<a href="#">?hseqr</a>	<a href="#">?hseqr</a>
Find eigenvectors from Hessenberg form (inverse iteration)	<a href="#">?hsein</a>	<a href="#">?hsein</a>
Find eigenvectors from Schur factorization	<a href="#">?trevc</a>	<a href="#">?trevc</a>
Estimate sensitivities of eigenvalues and eigenvectors	<a href="#">?trsna</a>	<a href="#">?trsna</a>
Reorder Schur factorization	<a href="#">?trexc</a>	<a href="#">?trexc</a>
Reorder Schur factorization, find the invariant subspace and estimate sensitivities	<a href="#">?trsen</a>	<a href="#">?trsen</a>
Solves Sylvester's equation.	<a href="#">?trsyl</a>	<a href="#">?trsyl</a>

**Figure 5-4 Decision Tree: Real Nonsymmetric Eigenvalue Problems**

**Figure 5-5 Decision Tree: Complex Non-Hermitian Eigenvalue Problems**



## ?gehrd

*Reduces a general matrix to upper Hessenberg form.*

---

```
call sgehrd ( n, ilo, ihi, a, lda, tau, work, lwork, info )
call dgehrd ( n, ilo, ihi, a, lda, tau, work, lwork, info )
call cgehrd ( n, ilo, ihi, a, lda, tau, work, lwork, info )
call zgehrd ( n, ilo, ihi, a, lda, tau, work, lwork, info )
```

### Discussion

The routine reduces a general matrix  $A$  to upper Hessenberg form  $H$  by an orthogonal or unitary similarity transformation  $A = QHQ^H$ . Here  $H$  has real subdiagonal elements.

The routine does not form the matrix  $Q$  explicitly. Instead,  $Q$  is represented as a product of *elementary reflectors*. Routines are provided to work with  $Q$  in this representation.

### Input Parameters

<i>n</i>	INTEGER. The order of the matrix $A$ ( $n \geq 0$ ).
<i>ilo, ihi</i>	INTEGER. If $A$ has been output by ?gebal, then <i>ilo</i> and <i>ihi</i> must contain the values returned by that routine. Otherwise <i>ilo</i> = 1 and <i>ihi</i> = <i>n</i> . (If <i>n</i> > 0, then 1 ≤ <i>ilo</i> ≤ <i>ihi</i> ≤ <i>n</i> ; if <i>n</i> = 0, <i>ilo</i> = 1 and <i>ihi</i> = 0.)
<i>a, work</i>	REAL for sgehrd DOUBLE PRECISION for dgehrd COMPLEX for cgehrd DOUBLE COMPLEX for zgehrd. Arrays: <i>a</i> ( <i>lda</i> , *) contains the matrix $A$ . The second dimension of <i>a</i> must be at least max(1, <i>n</i> ). <i>work</i> ( <i>lwork</i> ) is a workspace array.
<i>lda</i>	INTEGER. The first dimension of <i>a</i> ; at least max(1, <i>m</i> ).

*lwork*            **INTEGER.** The size of the *work* array; at least  $\max(1, n)$ .  
 See *Application notes* for the suggested value of *lwork*.

## Output Parameters

<i>a</i>	Overwritten by the upper Hessenberg matrix <i>H</i> and details of the matrix <i>Q</i> . The subdiagonal elements of <i>H</i> are real.
<i>tau</i>	<b>REAL</b> for <i>sgehrd</i> <b>DOUBLE PRECISION</b> for <i>dgehrd</i> <b>COMPLEX</b> for <i>cgehrd</i> <b>DOUBLE COMPLEX</b> for <i>zgehrd</i> . Array, <b>DIMENSION</b> at least $\max(1, n-1)$ . Contains additional information on the matrix <i>Q</i> .
<i>work(1)</i>	If <i>info</i> = 0, on exit <i>work(1)</i> contains the minimum value of <i>lwork</i> required for optimum performance. Use this <i>lwork</i> for subsequent runs.
<i>info</i>	<b>INTEGER.</b> If <i>info</i> = 0, the execution is successful. If <i>info</i> = <i>-i</i> , the <i>i</i> th parameter had an illegal value.

## Application Notes

For better performance, try using *lwork* =  $n * \text{blocksize}$ , where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*. If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run. On exit, examine *work(1)* and use this value for subsequent runs.

The computed Hessenberg matrix *H* is exactly similar to a nearby matrix *A + E*, where  $\|E\|_2 < c(n)\epsilon\|A\|_2$ ,  $c(n)$  is a modestly increasing function of *n*, and  $\epsilon$  is the machine precision.

The approximate number of floating-point operations for real flavors is  $(2/3)(\text{ih}i - \text{ilo})^2(2\text{ih}i + 2\text{ilo} + 3n)$ ; for complex flavors it is 4 times greater.

---

## ?orghr

*Generates the real orthogonal matrix  $Q$  determined by ?gehrd.*

---

```
call sorghr ( n, ilo, ihi, a, lda, tau, work, lwork, info )
call dorghr ( n, ilo, ihi, a, lda, tau, work, lwork, info )
```

### Discussion

This routine explicitly generates the orthogonal matrix  $Q$  that has been determined by a preceding call to `sgehrd/dgehrd`. (The routine `?gehrd` reduces a real general matrix  $A$  to upper Hessenberg form  $H$  by an orthogonal similarity transformation,  $A = QHQ^T$ , and represents the matrix  $Q$  as a product of  $ihi - ilo$  elementary reflectors. Here `ilo` and `ihi` are values determined by `sgebal/dgebal` when balancing the matrix; if the matrix has not been balanced, `ilo = 1` and `ihi = n`.)

The matrix  $Q$  generated by `?orghr` has the structure:

$$Q = \begin{bmatrix} I & 0 & 0 \\ 0 & Q_{22} & 0 \\ 0 & 0 & I \end{bmatrix}$$

where  $Q_{22}$  occupies rows and columns `ilo` to `ihi`.

### Input Parameters

<code>n</code>	<code>INTEGER</code> . The order of the matrix $Q$ ( $n \geq 0$ ).
<code>ilo, ihi</code>	<code>INTEGER</code> . These must be the same parameters <code>ilo</code> and <code>ihi</code> , respectively, as supplied to <code>?gehrd</code> . (If $n > 0$ , then $1 \leq ilo \leq ihi \leq n$ ; if $n = 0$ , <code>ilo = 1</code> and <code>ihi = 0</code> .)
<code>a, tau, work</code>	<code>REAL</code> for <code>sorghr</code> <code>DOUBLE PRECISION</code> for <code>dorghr</code> Arrays:

`a(lda, *)` contains details of the vectors which define the elementary reflectors, as returned by [?gehrd](#).

The second dimension of `a` must be at least  $\max(1, n)$ .

`tau(*)` contains further details of the elementary reflectors, as returned by [?gehrd](#).

The dimension of `tau` must be at least  $\max(1, n-1)$ .

`work(lwork)` is a workspace array.

`lda` INTEGER. The first dimension of `a`; at least  $\max(1, n)$ .

`lwork` INTEGER. The size of the `work` array;

$lwork \geq \max(1, ihi - ilo)$ .

See *Application notes* for the suggested value of `lwork`.

## Output Parameters

`a` Overwritten by the  $n$  by  $n$  orthogonal matrix  $Q$ .

`work(1)` If `info` = 0, on exit `work(1)` contains the minimum value of `lwork` required for optimum performance. Use this `lwork` for subsequent runs.

`info` INTEGER.

If `info` = 0, the execution is successful.

If `info` =  $-i$ , the  $i$ th parameter had an illegal value.

## Application Notes

For better performance, try using `lwork = (ihi - ilo) * blocksize`, where `blocksize` is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*. If you are in doubt how much workspace to supply, use a generous value of `lwork` for the first run. On exit, examine `work(1)` and use this value for subsequent runs.

The computed matrix  $Q$  differs from the exact result by a matrix  $E$  such that  $\|E\|_2 = O(\epsilon)$ , where  $\epsilon$  is the machine precision.

The approximate number of floating-point operations is  $(4/3)(ihi - ilo)^3$ .

The complex counterpart of this routine is [?unghr](#).

---

## ?ormhr

*Multiplies an arbitrary real matrix C by the real orthogonal matrix Q determined by ?gehrd.*

---

```
call sormhr ( side, trans, m, n, ilo, ihi, a, lda, tau, c, ldc,
              work, lwork, info )
call dormhr ( side, trans, m, n, ilo, ihi, a, lda, tau, c, ldc,
              work, lwork, info )
```

### Discussion

This routine multiplies a matrix  $C$  by the orthogonal matrix  $Q$  that has been determined by a preceding call to `sgehrd/dgehrd`. (The routine `?gehrd` reduces a real general matrix  $A$  to upper Hessenberg form  $H$  by an orthogonal similarity transformation,  $A = QHQ^T$ , and represents the matrix  $Q$  as a product of  $ihi - ilo$  elementary reflectors. Here `ilo` and `ihi` are values determined by `sgebal/dgebal` when balancing the matrix; if the matrix has not been balanced, `ilo` = 1 and `ihi` =  $n$ .)

With `?ormhr`, you can form one of the matrix products  $QC$ ,  $Q^TC$ ,  $CQ$ , or  $CQ^T$ , overwriting the result on  $C$  (which may be any real rectangular matrix).

A common application of `?ormhr` is to transform a matrix  $V$  of eigenvectors of  $H$  to the matrix  $QV$  of eigenvectors of  $A$ .

### Input Parameters

<code>side</code>	CHARACTER*1. Must be ' <code>L</code> ' or ' <code>R</code> '. If <code>side</code> = ' <code>L</code> ', then the routine forms $QC$ or $Q^TC$ . If <code>side</code> = ' <code>R</code> ', then the routine forms $CQ$ or $CQ^T$ .
<code>trans</code>	CHARACTER*1. Must be ' <code>N</code> ' or ' <code>T</code> '. If <code>trans</code> = ' <code>N</code> ', then $Q$ is applied to $C$ . If <code>trans</code> = ' <code>T</code> ', then $Q^T$ is applied to $C$ .
<code>m</code>	INTEGER. The number of rows in $C$ ( $m \geq 0$ ).
<code>n</code>	INTEGER. The number of columns in $C$ ( $n \geq 0$ ).

*ilo, ihi* INTEGER. These must be the same parameters *ilo* and *ihi*, respectively, as supplied to ?gehrd.  
 If *m* > 0 and *side* = 'L', then  $1 \leq ilo \leq ihi \leq m$ .  
 If *m* = 0 and *side* = 'L', then *ilo* = 1 and *ihi* = 0.  
 If *n* > 0 and *side* = 'R', then  $1 \leq ilo \leq ihi \leq n$ .  
 If *n* = 0 and *side* = 'R', then *ilo* = 1 and *ihi* = 0.

*a, tau, c, work* REAL for sormhr  
 DOUBLE PRECISION for dormhr  
 Arrays:  
*a( lda, \* )* contains details of the vectors which define the *elementary reflectors*, as returned by ?gehrd.  
 The second dimension of *a* must be at least max(1, *m*) if *side* = 'L' and at least max(1, *n*) if *side* = 'R'.  
*tau( \* )* contains further details of the *elementary reflectors*, as returned by ?gehrd.  
 The dimension of *tau* must be at least max (1, *m*-1) if *side* = 'L' and at least max (1, *n*-1) if *side* = 'R'.  
*c( ldc, \* )* contains the *m* by *n* matrix *C*.  
 The second dimension of *c* must be at least max(1, *n*).  
*work ( lwork )* is a workspace array.

*lda* INTEGER. The first dimension of *a*; at least max(1, *m*) if *side* = 'L' and at least max (1, *n*) if *side* = 'R'.  
*ldc* INTEGER. The first dimension of *c*; at least max(1, *m*).  
*lwork* INTEGER. The size of the *work* array.  
 If *side* = 'L', *lwork*  $\geq \max(1,n)$ .  
 If *side* = 'R', *lwork*  $\geq \max(1,m)$ .  
 See Application notes for the suggested value of *lwork*.

## Output Parameters

*c* *C* is overwritten by  $QC$  or  $Q^T C$  or  $CQ^T$  or  $CQ$  as specified by *side* and *trans*.  
*work(1)* If *info* = 0, on exit *work(1)* contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.

*info*            INTEGER.  
If *info* = 0, the execution is successful.  
If *info* = -*i*, the *i*th parameter had an illegal value.

### Application Notes

For better performance, *lwork* should be at least *n*\**blocksize* if *side* = 'L' and at least *m*\**blocksize* if *side* = 'R', where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*. If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run. On exit, examine *work(1)* and use this value for subsequent runs.

The computed matrix *Q* differs from the exact result by a matrix *E* such that  $\|E\|_2 = O(\epsilon)\|C\|_2$ , where  $\epsilon$  is the machine precision.

The approximate number of floating-point operations is  
 $2n(ihi-ilo)^2$  if *side* = 'L';  
 $2m(ihi-ilo)^2$  if *side* = 'R'.

The complex counterpart of this routine is [?unmhr](#).

## ?unghr

Generates the complex unitary matrix  $Q$  determined by ?gehrd.

```
call cunghr ( n, ilo, ihi, a, lda, tau, work, lwork, info )
call zunghr ( n, ilo, ihi, a, lda, tau, work, lwork, info )
```

### Discussion

This routine is intended to be used following a call to `cgehrd/zgehrd`, which reduces a complex matrix  $A$  to upper Hessenberg form  $H$  by a unitary similarity transformation:  $A = QHQ^H$ . ?gehrd represents the matrix  $Q$  as a product of  $ihi - ilo$  elementary reflectors. Here `ilo` and `ihi` are values determined by `cgebal/zgebal` when balancing the matrix; if the matrix has not been balanced, `ilo` = 1 and `ihi` = `n`.

Use the routine ?unghr to generate  $Q$  explicitly as a square matrix. The matrix  $Q$  has the structure:

$$Q = \begin{bmatrix} I & 0 & 0 \\ 0 & Q_{22} & 0 \\ 0 & 0 & I \end{bmatrix}$$

where  $Q_{22}$  occupies rows and columns `ilo` to `ihi`.

### Input Parameters

<code>n</code>	INTEGER. The order of the matrix $Q$ ( $n \geq 0$ ).
<code>ilo, ihi</code>	INTEGER. These must be the same parameters <code>ilo</code> and <code>ihi</code> , respectively, as supplied to ?gehrd. (If <code>n</code> > 0, then $1 \leq ilo \leq ihi \leq n$ . If <code>n</code> = 0, then <code>ilo</code> = 1 and <code>ihi</code> = 0.)
<code>a, tau, work</code>	COMPLEX for cunghr DOUBLE COMPLEX for zunghr. Arrays:

$a(\text{lda}, *)$  contains details of the vectors which define the *elementary reflectors*, as returned by [?gehrd](#).  
The second dimension of  $a$  must be at least  $\max(1, n)$ .

$\tau(\text{*})$  contains further details of the *elementary reflectors*, as returned by [?gehrd](#).

The dimension of  $\tau$  must be at least  $\max(1, n-1)$ .

$work(\text{lwork})$  is a workspace array.

$\text{lda}$  INTEGER. The first dimension of  $a$ ; at least  $\max(1, n)$ .

$\text{lwork}$  INTEGER. The size of the  $work$  array;  
 $\text{lwork} \geq \max(1, ihi - ilo)$ .

See *Application notes* for the suggested value of  $\text{lwork}$ .

## Output Parameters

$a$  Overwritten by the  $n$  by  $n$  unitary matrix  $Q$ .

$work(1)$  If  $\text{info} = 0$ , on exit  $work(1)$  contains the minimum value of  $\text{lwork}$  required for optimum performance. Use this  $\text{lwork}$  for subsequent runs.

$\text{info}$  INTEGER.

If  $\text{info} = 0$ , the execution is successful.

If  $\text{info} = -i$ , the  $i$ th parameter had an illegal value.

## Application Notes

For better performance, try using  $\text{lwork} = (ihi - ilo) * blocksize$ , where  $blocksize$  is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*. If you are in doubt how much workspace to supply, use a generous value of  $\text{lwork}$  for the first run. On exit, examine  $work(1)$  and use this value for subsequent runs.

The computed matrix  $Q$  differs from the exact result by a matrix  $E$  such that  $\|E\|_2 = O(\epsilon)$ , where  $\epsilon$  is the machine precision.

The approximate number of real floating-point operations is  $(16/3)(ihi - ilo)^3$ .

The real counterpart of this routine is [?orghr](#).

## ?unmhr

*Multiples an arbitrary complex matrix*

*C by the complex unitary matrix Q*

*determined by ?gehrd.*

```
call cunmhr ( side, trans, m, n, ilo, ihi, a, lda, tau, c, ldc,
              work, lwork, info )
call zunmhr ( side, trans, m, n, ilo, ihi, a, lda, tau, c, ldc,
              work, lwork, info )
```

### Discussion

This routine multiplies a matrix  $C$  by the unitary matrix  $Q$  that has been determined by a preceding call to `cgehrd/zgehrd`. (The routine `?gehrd` reduces a real general matrix  $A$  to upper Hessenberg form  $H$  by an orthogonal similarity transformation,  $A = QHQ^H$ , and represents the matrix  $Q$  as a product of  $ihi - ilo$  elementary reflectors. Here `ilo` and `ihi` are values determined by `cgebal/zgebal` when balancing the matrix; if the matrix has not been balanced, `ilo` = 1 and `ihi` =  $n$ .)

With `?unmhr`, you can form one of the matrix products  $QC$ ,  $Q^HC$ ,  $CQ$ , or  $CQ^H$ , overwriting the result on  $C$  (which may be any complex rectangular matrix). A common application of this routine is to transform a matrix  $V$  of eigenvectors of  $H$  to the matrix  $QV$  of eigenvectors of  $A$ .

### Input Parameters

<code>side</code>	CHARACTER*1. Must be ' <code>L</code> ' or ' <code>R</code> '. If <code>side</code> = ' <code>L</code> ', then the routine forms $QC$ or $Q^HC$ . If <code>side</code> = ' <code>R</code> ', then the routine forms $CQ$ or $CQ^H$ .
<code>trans</code>	CHARACTER*1. Must be ' <code>N</code> ' or ' <code>C</code> '. If <code>trans</code> = ' <code>N</code> ', then $Q$ is applied to $C$ . If <code>trans</code> = ' <code>T</code> ', then $Q^H$ is applied to $C$ .
<code>m</code>	INTEGER. The number of rows in $C$ ( $m \geq 0$ ).
<code>n</code>	INTEGER. The number of columns in $C$ ( $n \geq 0$ ).

<i>ilo, ihi</i>	<b>INTEGER.</b> These must be the same parameters <i>ilo</i> and <i>ihi</i> , respectively, as supplied to <a href="#">?gehrd</a> . If <i>m</i> > 0 and <i>side</i> = 'L', then $1 \leq ilo \leq ihi \leq m$ . If <i>m</i> = 0 and <i>side</i> = 'L', then <i>ilo</i> = 1 and <i>ihi</i> = 0. If <i>n</i> > 0 and <i>side</i> = 'R', then $1 \leq ilo \leq ihi \leq n$ . If <i>n</i> = 0 and <i>side</i> = 'R', then <i>ilo</i> = 1 and <i>ihi</i> = 0.
<i>a, tau, c, work</i>	<b>COMPLEX</b> for <a href="#">cunmhr</a> <b>DOUBLE COMPLEX</b> for <a href="#">zunmhr</a> .
<i>Arrays:</i>	
<i>a</i> ( <i>lda</i> , *)	contains details of the vectors which define the elementary reflectors, as returned by <a href="#">?gehrd</a> . The second dimension of <i>a</i> must be at least $\max(1, m)$ if <i>side</i> = 'L' and at least $\max(1, n)$ if <i>side</i> = 'R'.
<i>tau</i> (*)	contains further details of the elementary reflectors, as returned by <a href="#">?gehrd</a> . The dimension of <i>tau</i> must be at least $\max(1, m-1)$ if <i>side</i> = 'L' and at least $\max(1, n-1)$ if <i>side</i> = 'R'.
<i>c</i> ( <i>ldc</i> , *)	contains the <i>m</i> by <i>n</i> matrix <i>C</i> . The second dimension of <i>c</i> must be at least $\max(1, n)$ .
<i>work</i> ( <i>lwork</i> )	is a workspace array.
<i>lda</i>	<b>INTEGER.</b> The first dimension of <i>a</i> ; at least $\max(1, m)$ if <i>side</i> = 'L' and at least $\max(1, n)$ if <i>side</i> = 'R'.
<i>ldc</i>	<b>INTEGER.</b> The first dimension of <i>c</i> ; at least $\max(1, m)$ .
<i>lwork</i>	<b>INTEGER.</b> The size of the <i>work</i> array. If <i>side</i> = 'L', <i>lwork</i> $\geq \max(1, n)$ . If <i>side</i> = 'R', <i>lwork</i> $\geq \max(1, m)$ . See <i>Application notes</i> for the suggested value of <i>lwork</i> .

## Output Parameters

<i>c</i>	<i>C</i> is overwritten by $QC$ or $Q^H C$ or $CQ^H$ or $CQ$ as specified by <i>side</i> and <i>trans</i> .
<i>work</i> (1)	If <i>info</i> = 0, on exit <i>work</i> (1) contains the minimum value of <i>lwork</i> required for optimum performance. Use this <i>lwork</i> for subsequent runs.

*info*

INTEGER.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*th parameter had an illegal value.

## Application Notes

For better performance, *lwork* should be at least *n\*blocksize* if *side* = 'L' and at least *m\*blocksize* if *side* = 'R', where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*. If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run. On exit, examine *work(1)* and use this value for subsequent runs.

The computed matrix *Q* differs from the exact result by a matrix *E* such that  $\|E\|_2 = O(\epsilon) \|C\|_2$ , where  $\epsilon$  is the machine precision.

The approximate number of floating-point operations is

$8n(ihi-ilo)^2$  if *side* = 'L';  
 $8m(ihi-ilo)^2$  if *side* = 'R'.

The real counterpart of this routine is [?ormhr](#).

## ?gebal

Balances a general matrix to improve the accuracy of computed eigenvalues and eigenvectors.

---

```
call sgebal ( job, n, a, lda, ilo, ihi, scale, info )
call dgebal ( job, n, a, lda, ilo, ihi, scale, info )
call cgebal ( job, n, a, lda, ilo, ihi, scale, info )
call zgebal ( job, n, a, lda, ilo, ihi, scale, info )
```

### Discussion

This routine *balances* a matrix  $A$  by performing either or both of the following two similarity transformations:

- (1) The routine first attempts to permute  $A$  to block upper triangular form:

$$PAP^T = A' = \begin{bmatrix} A'_{11} & A'_{12} & A'_{13} \\ 0 & A'_{22} & A'_{23} \\ 0 & 0 & A'_{33} \end{bmatrix}$$

where  $P$  is a permutation matrix, and  $A'_{11}$  and  $A'_{33}$  are upper triangular. The diagonal elements of  $A'_{11}$  and  $A'_{33}$  are eigenvalues of  $A$ . The rest of the eigenvalues of  $A$  are the eigenvalues of the central diagonal block  $A'_{22}$ , in rows and columns *ilo* to *ihi*. Subsequent operations to compute the eigenvalues of  $A$  (or its Schur factorization) need only be applied to these rows and columns; this can save a significant amount of work if *ilo* > 1 and *ihi* < *n*. If no suitable permutation exists (as is often the case), the routine sets *ilo* = 1 and *ihi* = *n*, and  $A'_{22}$  is the whole of  $A$ .

- (2) The routine applies a diagonal similarity transformation to  $A'$ , to make the rows and columns of  $A'_{22}$  as close in norm as possible:

$$A'' = DA'D^{-1} = \begin{bmatrix} I & 0 & 0 \\ 0 & D_{22} & 0 \\ 0 & 0 & I \end{bmatrix} \times \begin{bmatrix} A'_{11} & A'_{12} & A'_{13} \\ 0 & A'_{22} & A'_{23} \\ 0 & 0 & A'_{33} \end{bmatrix} \times \begin{bmatrix} I & 0 & 0 \\ 0 & D_{22}^{-1} & 0 \\ 0 & 0 & I \end{bmatrix}$$

This scaling can reduce the norm of the matrix (that is,  $\|A''_{22}\| < \|A'_{22}\|$ ), and hence reduce the effect of rounding errors on the accuracy of computed eigenvalues and eigenvectors.

### Input Parameters

<i>job</i>	CHARACTER*1. Must be ' <b>N</b> ' or ' <b>P</b> ' or ' <b>S</b> ' or ' <b>B</b> '. If <i>job</i> = ' <b>N</b> ', then <i>A</i> is neither permuted nor scaled (but <i>ilo</i> , <i>ih</i> , and <i>scale</i> get their values). If <i>job</i> = ' <b>P</b> ', then <i>A</i> is permuted but not scaled. If <i>job</i> = ' <b>S</b> ', then <i>A</i> is scaled but not permuted. If <i>job</i> = ' <b>B</b> ', then <i>A</i> is both scaled and permuted.
<i>n</i>	INTEGER. The order of the matrix <i>A</i> ( <i>n</i> ≥ 0).
<i>a</i>	REAL for <b>sgebal</b> DOUBLE PRECISION for <b>dgebal</b> COMPLEX for <b>cgebal</b> DOUBLE COMPLEX for <b>zgebal</b> . Arrays: <i>a</i> ( <i>lda</i> , *) contains the matrix <i>A</i> . The second dimension of <i>a</i> must be at least max(1, <i>n</i> ). <i>a</i> is not referenced if <i>job</i> = ' <b>N</b> '.
<i>lda</i>	INTEGER. The first dimension of <i>a</i> ; at least max(1, <i>n</i> ).

### Output Parameters

<i>a</i>	Overwritten by the balanced matrix ( <i>a</i> is not referenced if <i>job</i> = ' <b>N</b> ').
<i>ilo</i> , <i>ih</i>	INTEGER. The values <i>ilo</i> and <i>ih</i> such that on exit <i>a</i> ( <i>i</i> , <i>j</i> ) is zero if <i>i</i> > <i>j</i> and $1 \leq j < ilo$ or $ih < i \leq n$ . If <i>job</i> = ' <b>N</b> ' or ' <b>S</b> ', then <i>ilo</i> = 1 and <i>ih</i> = <i>n</i> .
<i>scale</i>	REAL for single-precision flavors DOUBLE PRECISION for double-precision flavors Array, DIMENSION at least max(1, <i>n</i> ). Contains details of the permutations and scaling factors.

More precisely, if  $p_j$  is the index of the row and column interchanged with row and column  $j$ , and  $d_j$  is the scaling factor used to balance row and column  $j$ , then  
 $\text{scale}(j) = p_j$  for  $j = 1, 2, \dots, \text{ilo}-1, \text{ghi}+1, \dots, n$ ;  
 $\text{scale}(j) = d_j$  for  $j = \text{ilo}, \text{ilo}+1, \dots, \text{ghi}$ .  
The order in which the interchanges are made is  $n$  to  $\text{ghi}+1$ , then 1 to  $\text{ilo}-1$ .

*info*

INTEGER.

If *info* = 0, the execution is successful.

If *info* =  $-i$ , the *i*th parameter had an illegal value.

## Application Notes

The errors are negligible, compared with those in subsequent computations.

If the matrix  $A$  is balanced by this routine, then any eigenvectors computed subsequently are eigenvectors of the matrix  $A''$  and hence you must call [?gebak](#) (see [page 5-193](#)) to transform them back to eigenvectors of  $A$ .

If the Schur vectors of  $A$  are required, do not call this routine with *job* = 'S' or 'B', because then the balancing transformation is not orthogonal (not unitary for complex flavors). If you call this routine with *job* = 'P', then any Schur vectors computed subsequently are Schur vectors of the matrix  $A''$ , and you'll need to call [?gebak](#) (with *side* = 'R') to transform them back to Schur vectors of  $A$ .

The total number of floating-point operations is proportional to  $n^2$ .

## ?gebak

*Transforms eigenvectors of a balanced matrix to those of the original nonsymmetric matrix.*

```
call sgebak ( job,side,n,ilo,ih,i,scale,m,v,ldv,info )
call dgebak ( job,side,n,ilo,ih,i,scale,m,v,ldv,info )
call cgebak ( job,side,n,ilo,ih,i,scale,m,v,ldv,info )
call zgebak ( job,side,n,ilo,ih,i,scale,m,v,ldv,info )
```

### Discussion

This routine is intended to be used after a matrix  $A$  has been balanced by a call to ?[gebal](#), and eigenvectors of the balanced matrix  $A''_{22}$  have subsequently been computed.

For a description of balancing, see ?[gebal](#) ([page 5-190](#)). The balanced matrix  $A''$  is obtained as  $A'' = DPAP^T D^{-1}$ , where  $P$  is a permutation matrix and  $D$  is a diagonal scaling matrix. This routine transforms the eigenvectors as follows:

if  $x$  is a right eigenvector of  $A''$ , then  $P^T D^{-1} x$  is a right eigenvector of  $A$ ;  
if  $x$  is a left eigenvector of  $A''$ , then  $P^T D y$  is a left eigenvector of  $A$ .

### Input Parameters

<i>job</i>	CHARACTER*1. Must be ' <b>N</b> ' or ' <b>P</b> ' or ' <b>S</b> ' or ' <b>B</b> '. The same parameter <i>job</i> as supplied to ? <a href="#">gebal</a> .
<i>side</i>	CHARACTER*1. Must be ' <b>L</b> ' or ' <b>R</b> '. If <i>side</i> = ' <b>L</b> ', then left eigenvectors are transformed. If <i>side</i> = ' <b>R</b> ', then right eigenvectors are transformed.
<i>n</i>	INTEGER. The number of rows of the matrix of eigenvectors ( <i>n</i> $\geq 0$ ).
<i>ilo, ihi</i>	INTEGER. The values <i>ilo</i> and <i>ihi</i> , as returned by ? <a href="#">gebal</a> . (If <i>n</i> > 0, then $1 \leq ilo \leq ihi \leq n$ ; if <i>n</i> = 0, then <i>ilo</i> = 1 and <i>ihi</i> = 0.)

<i>scale</i>	<i>REAL</i> for single-precision flavors <i>DOUBLE PRECISION</i> for double-precision flavors Array, <i>DIMENSION</i> at least $\max(1, n)$ . Contains details of the permutations and/or the scaling factors used to balance the original general matrix, as returned by <i>?gebal</i> .
<i>m</i>	<i>INTEGER</i> . The number of columns of the matrix of eigenvectors ( $m \geq 0$ ).
<i>v</i>	<i>REAL</i> for <i>sgebak</i> <i>DOUBLE PRECISION</i> for <i>dgebak</i> <i>COMPLEX</i> for <i>cgebak</i> <i>DOUBLE COMPLEX</i> for <i>zgebak</i> . Arrays: <i>v</i> ( <i>ldv</i> , *) contains the matrix of left or right eigenvectors to be transformed. The second dimension of <i>v</i> must be at least $\max(1, m)$ .
<i>ldv</i>	<i>INTEGER</i> . The first dimension of <i>v</i> ; at least $\max(1, n)$ .

## Output Parameters

<i>v</i>	Overwritten by the transformed eigenvectors.
<i>info</i>	<i>INTEGER</i> . If <i>info</i> = 0, the execution is successful. If <i>info</i> = <i>-i</i> , the <i>i</i> th parameter had an illegal value.

## Application Notes

The errors in this routine are negligible.

The approximate number of floating-point operations is approximately proportional to  $m^2 n$ .

## ?hseqr

*Computes all eigenvalues and  
(optionally) the Schur factorization of a  
matrix reduced to Hessenberg form.*

```
call shseqr ( job, compz, n, ilo, ihi, h, ldh, wr, wi, z, ldz, work, lwork, info)
call dhseqr ( job, compz, n, ilo, ihi, h, ldh, wr, wi, z, ldz, work, lwork, info)
call chseqr ( job, compz, n, ilo, ihi, h, ldh, w, z, ldz, work, lwork, info)
call zhseqr ( job, compz, n, ilo, ihi, h, ldh, w, z, ldz, work, lwork, info)
```

### Discussion

This routine computes all the eigenvalues, and optionally the Schur factorization, of an upper Hessenberg matrix  $H$ :  $H = ZTZ^H$ , where  $T$  is an upper triangular (or, for real flavors, quasi-triangular) matrix (the Schur form of  $H$ ), and  $Z$  is the unitary or orthogonal matrix whose columns are the Schur vectors  $z_i$ .

You can also use this routine to compute the Schur factorization of a general matrix  $A$  which has been reduced to upper Hessenberg form  $H$ :

$$\begin{aligned} A &= QHQ^H, \text{ where } Q \text{ is unitary (orthogonal for real flavors);} \\ A &= (QZ)T(QZ)^H. \end{aligned}$$

In this case, after reducing  $A$  to Hessenberg form by [?gehrd](#) ([page 5-178](#)), call [?orgqr](#) to form  $Q$  explicitly ([page 5-180](#)) and then pass  $Q$  to [?hseqr](#) with  $\text{compz} = 'V'$ .

You can also call [?gebal](#) ([page 5-190](#)) to balance the original matrix before reducing it to Hessenberg form by [?hseqr](#), so that the Hessenberg matrix  $H$  will have the structure:

$$\begin{bmatrix} H_{11} & H_{12} & H_{13} \\ 0 & H_{22} & H_{23} \\ 0 & 0 & H_{33} \end{bmatrix}$$

where  $H_{11}$  and  $H_{33}$  are upper triangular.

If so, only the central diagonal block  $H_{22}$  (in rows and columns  $\text{ilo}$  to  $\text{ih}$ ) needs to be further reduced to Schur form (the blocks  $H_{12}$  and  $H_{23}$  are also affected). Therefore the values of  $\text{ilo}$  and  $\text{ih}$  can be supplied to [?hseqr](#) directly. Also, after calling this routine you must call [?gebak](#) ([page 5-193](#)) to permute the Schur vectors of the balanced matrix to those of the original matrix.

If [?gebal](#) has not been called, however, then  $\text{ilo}$  must be set to 1 and  $\text{ih}$  to  $n$ . Note that if the Schur factorization of  $A$  is required, [?gebal](#) must not be called with  $\text{job} = 'S'$  or ' $B$ ', because the balancing transformation is not unitary (for real flavors, it is not orthogonal).

[?hseqr](#) uses a multishift form of the upper Hessenberg  $QR$  algorithm. The Schur vectors are normalized so that  $\|z_i\|_2 = 1$ , but are determined only to within a complex factor of absolute value 1 (for the real flavors, to within a factor  $\pm 1$ ).

## Input Parameters

<i>job</i>	CHARACTER*1. Must be ' $E$ ' or ' $S$ '. If $\text{job} = 'E'$ , then eigenvalues only are required. If $\text{job} = 'S'$ , then the Schur form $T$ is required.
<i>compz</i>	CHARACTER*1. Must be ' $N$ ' or ' $I$ ' or ' $V$ '. If $\text{compz} = 'N'$ , then no Schur vectors are computed (and the array $\text{z}$ is not referenced). If $\text{compz} = 'I'$ , then the Schur vectors of $H$ are computed (and the array $\text{z}$ is initialized by the routine). If $\text{compz} = 'V'$ , then the Schur vectors of $A$ are computed (and the array $\text{z}$ must contain the matrix $Q$ on entry).
<i>n</i>	INTEGER. The order of the matrix $H$ ( $n \geq 0$ ).
<i>ilo, ih</i>	INTEGER. If $A$ has been balanced by <a href="#">?gebal</a> , then $\text{ilo}$ and $\text{ih}$ must contain the values returned by <a href="#">?gebal</a> . Otherwise, $\text{ilo}$ must be set to 1 and $\text{ih}$ to $n$ .
<i>h, z, work</i>	REAL for <a href="#">shseqr</a> DOUBLE PRECISION for <a href="#">dhseqr</a> COMPLEX for <a href="#">chseqr</a> DOUBLE COMPLEX for <a href="#">zhseqr</a> .

Arrays:

`h( ldh, * )` The `n` by `n` upper Hessenberg matrix  $H$ .

The second dimension of `h` must be at least  $\max(1, n)$ .

`z( ldz, * )`

If `compz = 'V'`, then `z` must contain the matrix  $Q$  from the reduction to Hessenberg form.

If `compz = 'I'`, then `z` need not be set.

If `compz = 'N'`, then `z` is not referenced.

The second dimension of `z` must be

at least  $\max(1, n)$  if `compz = 'V'` or '`I`';

at least 1 if `compz = 'N'`.

`work(lwork)` is a workspace array.

The dimension of `work` must be at least  $\max(1, n)$ .

`ldh`

`INTEGER`. The first dimension of `h`; at least  $\max(1, n)$ .

`ldz`

`INTEGER`. The first dimension of `z`;

If `compz = 'N'`, then `ldz ≥ 1`.

If `compz = 'V'` or '`I`', then `ldz ≥ max(1, n)`.

`lwork`

This parameter is currently redundant.

## Output Parameters

`w`

`COMPLEX` for `chseqr`

`DOUBLE COMPLEX` for `zhseqr`.

Array, `DIMENSION` at least  $\max(1, n)$ .

Contains the computed eigenvalues, unless `info > 0`. The eigenvalues are stored in the same order as on the diagonal of the Schur form  $T$  (if computed).

`wr, wi`

`REAL` for `shseqr`

`DOUBLE PRECISION` for `dhseqr`

Arrays, `DIMENSION` at least  $\max(1, n)$  each.

Contain the real and imaginary parts, respectively, of the computed eigenvalues, unless `info > 0`. Complex conjugate pairs of eigenvalues appear consecutively with the eigenvalue having positive imaginary part first. The eigenvalues are stored in the same order as on the diagonal of the Schur form  $T$  (if computed).

<i>z</i>	If <i>compz</i> = 'V' or 'I', then <i>z</i> contains the unitary (orthogonal) matrix of the required Schur vectors, unless <i>info</i> > 0. If <i>compz</i> = 'N', then <i>z</i> is not referenced.
<i>info</i>	<b>INTEGER.</b> If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> th parameter had an illegal value. If <i>info</i> > 0, the algorithm has failed to find all the eigenvalues after a total $30(ihi-ilo+1)$ iterations. If <i>info</i> = <i>i</i> , elements 1,2, ..., <i>ilo</i> -1 and <i>i</i> +1, <i>i</i> +2, ..., <i>n</i> of <i>wr</i> and <i>wi</i> contain the real and imaginary parts of the eigenvalues which have been found.

## Application Notes

The computed Schur factorization is the exact factorization of a nearby matrix  $H + E$ , where  $\|E\|_2 < O(\epsilon) \|H\|_2/s_i$ , and  $\epsilon$  is the machine precision. If  $\lambda_i$  is an exact eigenvalue, and  $\mu_i$  is the corresponding computed value, then  $|\lambda_i - \mu_i| \leq c(n)\epsilon \|H\|_2/s_i$  where  $c(n)$  is a modestly increasing function of *n*, and  $s_i$  is the reciprocal condition number of  $\lambda_i$ . You can compute the condition numbers  $s_i$  by calling ?trsna (see [page 5-209](#)).

The total number of floating-point operations depends on how rapidly the algorithm converges; typical numbers are as follows.

If only eigenvalues are computed:       $7n^3$  for real flavors  
 $25n^3$  for complex flavors.

If the Schur form is computed:       $10n^3$  for real flavors  
 $35n^3$  for complex flavors.

If the full Schur factorization is computed:  $20n^3$  for real flavors  
 $70n^3$  for complex flavors.

## ?hsein

*Computes selected eigenvectors of an upper Hessenberg matrix that correspond to specified eigenvalues.*

```
call shsein ( job, eigsrc, initv, select, n, h, ldh, wr, wi, vl,
    ldvl, vr, ldvr, mm, m, work, ifaill, ifailr, info )
call dhsein ( job, eigsrc, initv, select, n, h, ldh, wr, wi, vl,
    ldvl, vr, ldvr, mm, m, work, ifaill, ifailr, info )
call chsein ( job, eigsrc, initv, select, n, h, ldh, w, vl,
    ldvl, vr, ldvr, mm, m, work, rwork, ifaill, ifailr, info )
call zhsein ( job, eigsrc, initv, select, n, h, ldh, w, vl,
    ldvl, vr, ldvr, mm, m, work, rwork, ifaill, ifailr, info )
```

### Discussion

This routine computes left and/or right eigenvectors of an upper Hessenberg matrix  $H$ , corresponding to selected eigenvalues.

The right eigenvector  $x$  and the left eigenvector  $y$ , corresponding to an eigenvalue  $\lambda$ , are defined by:  $Hx = \lambda x$  and  $y^H H = \lambda y^H$  (or  $H^H y = \lambda^* y$ ). Here  $\lambda^*$  denotes the conjugate of  $\lambda$ .

The eigenvectors are computed by inverse iteration. They are scaled so that, for a real eigenvector  $x$ ,  $\max|x_i| = 1$ , and for a complex eigenvector,  $\max(|\text{Re}x_i| + |\text{Im}x_i|) = 1$ .

If  $H$  has been formed by reduction of a general matrix  $A$  to upper Hessenberg form, then eigenvectors of  $H$  may be transformed to eigenvectors of  $A$  by [?ormhr](#) (page 5-182) or [?unmhr](#) (page 5-187).

### Input Parameters

<i>job</i>	CHARACTER*1. Must be ' <b>R</b> ' or ' <b>L</b> ' or ' <b>B</b> '.
If <i>job</i> = ' <b>R</b> ',	then only right eigenvectors are computed.
If <i>job</i> = ' <b>L</b> ',	then only left eigenvectors are computed.
If <i>job</i> = ' <b>B</b> ',	then all eigenvectors are computed.

<i>eigsrc</i>	CHARACTER*1. Must be ' <b>Q</b> ' or ' <b>N</b> '. If <i>eigsrc</i> ='Q', then the eigenvalues of <i>H</i> were found using ?hseqr (see <a href="#">page 5-195</a> ); thus if <i>H</i> has any zero sub-diagonal elements (and so is block triangular), then the <i>j</i> th eigenvalue can be assumed to be an eigenvalue of the block containing the <i>j</i> th row/column. This property allows the routine to perform inverse iteration on just one diagonal block. If <i>eigsrc</i> ='N', then no such assumption is made and the routine performs inverse iteration using the whole matrix.
<i>initv</i>	CHARACTER*1. Must be ' <b>N</b> ' or ' <b>U</b> '. If <i>initv</i> ='N', then no initial estimates for the selected eigenvectors are supplied. If <i>initv</i> ='U', then initial estimates for the selected eigenvectors are supplied in <i>vl</i> and/or <i>vr</i> .
<i>select</i>	LOGICAL. Array, DIMENSION at least max (1, <i>n</i> ). Specifies which eigenvectors are to be computed. <i>For real flavors:</i> To obtain the real eigenvector corresponding to the real eigenvalue <i>wr(j)</i> , set <i>select(j)</i> to .TRUE. To select the complex eigenvector corresponding to the complex eigenvalue ( <i>wr(j)</i> , <i>wi(j)</i> ) with complex conjugate ( <i>wr(j+1)</i> , <i>wi(j+1)</i> ), set <i>select(j)</i> and/or <i>select(j+1)</i> to .TRUE.; the eigenvector corresponding to the first eigenvalue in the pair is computed. <i>For complex flavors:</i> To select the eigenvector corresponding to the eigenvalue <i>w(j)</i> , set <i>select(j)</i> to .TRUE.
<i>n</i>	INTEGER. The order of the matrix <i>H</i> ( <i>n</i> ≥ 0).
<i>h, vl, vr, work</i>	REAL for shsein DOUBLE PRECISION for dhsein COMPLEX for chsein DOUBLE COMPLEX for zhsein.

Arrays:

***h(1dh,\*)*** The *n* by *n* upper Hessenberg matrix *H*.

The second dimension of *h* must be at least max(1, *n*).

***vl(1dvl,\*)***

If *initv* = 'V' and *job* = 'L' or 'B', then *vl* must contain starting vectors for inverse iteration for the left eigenvectors. Each starting vector must be stored in the same column or columns as will be used to store the corresponding eigenvector.

If *initv* = 'N', then *vl* need not be set.

The second dimension of *vl* must be at least max(1, *mm*) if *job* = 'L' or 'B' and at least 1 if *job* = 'R'.

The array *vl* is not referenced if *job* = 'R'.

***vr(1dvr,\*)***

If *initv* = 'V' and *job* = 'R' or 'B', then *vr* must contain starting vectors for inverse iteration for the right eigenvectors. Each starting vector must be stored in the same column or columns as will be used to store the corresponding eigenvector.

If *initv* = 'N', then *vr* need not be set.

The second dimension of *vr* must be at least max(1, *mm*) if *job* = 'R' or 'B' and at least 1 if *job* = 'L'.

The array *vr* is not referenced if *job* = 'L'.

***work(\*)*** is a workspace array.

**DIMENSION** at least max (1, *n*\*(*n*+2)) for real flavors and at least max (1, *n*\**n*) for complex flavors.

***ldh***

**INTEGER**. The first dimension of *h*; at least max(1, *n*).

***w***

**COMPLEX** for *chsein*

**DOUBLE COMPLEX** for *zhsein*.

Array, **DIMENSION** at least max (1, *n*).

Contains the eigenvalues of the matrix *H*.

If *eigsrc* = 'Q', the array must be exactly as returned by *?hseqr*.

<i>wr, wi</i>	<small>REAL for <b>shsein</b> DOUBLE PRECISION for <b>dhsein</b></small> Arrays, DIMENSION at least max (1, <i>n</i> ) each. Contain the real and imaginary parts, respectively, of the eigenvalues of the matrix <i>H</i> . Complex conjugate pairs of values must be stored in consecutive elements of the arrays. If <i>eigsrc</i> = 'Q', the arrays must be exactly as returned by ?hseqr.
<i>ldvl</i>	<small>INTEGER. The first dimension of <b>vl</b>. If <i>job</i> = 'L' or 'B', <i>ldvl</i> <math>\geq \max(1, n)</math>. If <i>job</i> = 'R', <i>ldvl</i> <math>\geq 1</math>.</small>
<i>ldvr</i>	<small>INTEGER. The first dimension of <b>vr</b>. If <i>job</i> = 'R' or 'B', <i>ldvr</i> <math>\geq \max(1, n)</math>. If <i>job</i> = 'L', <i>ldvr</i> <math>\geq 1</math>.</small>
<i>mm</i>	<small>INTEGER. The number of columns in <b>vl</b> and/or <b>vr</b>. Must be at least <i>m</i>, the actual number of columns required (see <i>Output Parameters</i> below). <i>For real flavors</i>, <i>m</i> is obtained by counting 1 for each selected real eigenvector and 2 for each selected complex eigenvector (see <i>select</i>). <i>For complex flavors</i>, <i>m</i> is the number of selected eigenvectors (see <i>select</i>). Constraint: <math>0 \leq mm \leq n</math>.</small>
<i>rwork</i>	<small>REAL for <b>chsein</b> DOUBLE PRECISION for <b>zhsein</b>. Array, DIMENSION at least max (1, <i>n</i>). </small>

## Output Parameters

<i>select</i>	Overwritten for real flavors only. If a complex eigenvector was selected as specified above, then <i>select(j)</i> is set to .TRUE. and <i>select(j+1)</i> to .FALSE.
<i>w</i>	The real parts of some elements of <i>w</i> may be modified, as close eigenvalues are perturbed slightly in searching for independent eigenvectors.

***wr***

Some elements of ***wr*** may be modified, as close eigenvalues are perturbed slightly in searching for independent eigenvectors.

***vl, vr***

If ***job*** = 'L' or 'B', ***vl*** contains the computed left eigenvectors (as specified by ***select***).  
If ***job*** = 'R' or 'B', ***vr*** contains the computed right eigenvectors (as specified by ***select***).

The eigenvectors are stored consecutively in the columns of the array, in the same order as their eigenvalues.

*For real flavors:* a real eigenvector corresponding to a selected real eigenvalue occupies one column;  
a complex eigenvector corresponding to a selected complex eigenvalue occupies two columns: the first column holds the real part and the second column holds the imaginary part.

***m***

**INTEGER.** *For real flavors:* the number of columns of ***vl*** and/or ***vr*** required to store the selected eigenvectors.

*For complex flavors:* the number of selected eigenvectors.

***ifaill, ifailr*** **INTEGER.**

Arrays, **DIMENSION** at least **max(1, mm)** each.

***ifaill(i)*** = 0 if the ***i***th column of ***vl*** converged;  
***ifaill(i)*** = ***j*** > 0 if the eigenvector stored in the ***i***th column of ***vl*** (corresponding to the ***j***th eigenvalue) failed to converge.

***ifailr(i)*** = 0 if the ***i***th column of ***vr*** converged;  
***ifailr(i)*** = ***j*** > 0 if the eigenvector stored in the ***i***th column of ***vr*** (corresponding to the ***j***th eigenvalue) failed to converge.

*For real flavors:* if the ***i***th and (***i***+1)th columns of ***vl*** contain a selected complex eigenvector, then

***ifaill(i)*** and ***ifaill(i+1)*** are set to the same value. A similar rule holds for ***vr*** and ***ifailr***.

The array ***ifaill*** is not referenced if ***job*** = 'R'.

The array ***ifailr*** is not referenced if ***job*** = 'L'.

*info**INTEGER.*

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*th parameter had an illegal value.

If *info* > 0, then *i* eigenvectors (as indicated by the parameters *ifail1* and/or *ifailr* above) failed to converge. The corresponding columns of *vl* and/or *vr* contain no useful information.

### Application Notes

Each computed right eigenvector  $x_i$  is the exact eigenvector of a nearby matrix  $A + E_i$ , such that  $\|E_i\| < O(\varepsilon)\|A\|$ . Hence the residual is small:  
 $\|Ax_i - \lambda_i x_i\| = O(\varepsilon)\|A\|$ .

However, eigenvectors corresponding to close or coincident eigenvalues may not accurately span the relevant subspaces.

Similar remarks apply to computed left eigenvectors.

## ?trevc

*Computes selected eigenvectors of an upper (quasi-) triangular matrix computed by ?hseqr*

```
call strevc ( job, howmny, select, n, t, ldt, vl, ldvl, vr, ldvr,
             mm, m, work, info )
call dtrevc ( job, howmny, select, n, t, ldt, vl, ldvl, vr, ldvr,
             mm, m, work, info )
call ctrevc ( job, howmny, select, n, t, ldt, vl, ldvl, vr, ldvr,
             mm, m, work, rwork, info )
call ztrevc ( job, howmny, select, n, t, ldt, vl, ldvl, vr, ldvr,
             mm, m, work, rwork, info )
```

### Discussion

This routine computes left and/or right eigenvectors of an upper triangular matrix  $T$  (or, for real flavors, an upper quasi-triangular matrix  $T$ ) in canonical Schur form. Such a matrix arises from the Schur factorization of a general matrix, as computed by ?hseqr (see [page 5-195](#)).

The right eigenvector  $x$  and the left eigenvector  $y$ , corresponding to an eigenvalue  $\lambda$ , are defined by:  $Hx = \lambda x$  and  $y^H H = \lambda y^H$  (or  $H^H y = \lambda^* y$ ). Here  $\lambda^*$  denotes the conjugate of  $\lambda$ .

The routine can compute the eigenvectors corresponding to selected eigenvalues, or it can compute all the eigenvectors. In the latter case, the eigenvectors may optionally be pre-multiplied by an input matrix  $Q$ . Normally  $Q$  is a unitary (or, for real flavors, orthogonal) matrix from the Schur factorization:  $A = QTQ^H$ . If  $x$  is a (left or right) eigenvector of  $T$ , then  $Qx$  is an eigenvector of  $A$ .

The eigenvectors are computed by forward or backward substitution. They are scaled so that, for a real eigenvector  $x$ ,  $\max|x_i| = 1$  and, for a complex eigenvector,  $\max(|\operatorname{Re}x_i| + |\operatorname{Im}x_i|) = 1$ .

## Input Parameters

<i>job</i>	CHARACTER*1. Must be 'R' or 'L' or 'B'. If <i>job</i> = 'R', then only right eigenvectors are computed. If <i>job</i> = 'L', then only left eigenvectors are computed. If <i>job</i> = 'B', then all eigenvectors are computed.
<i>howmny</i>	CHARACTER*1. Must be 'A' or 'O' or 'S'. If <i>howmny</i> = 'A', then all eigenvectors (as specified by <i>job</i> ) are computed. If <i>howmny</i> = 'O', then all eigenvectors (as specified by <i>job</i> ) are computed and then pre-multiplied by the matrix <i>Q</i> (which is overwritten). If <i>howmny</i> = 'S', then selected eigenvectors (as specified by <i>job</i> and <i>select</i> ) are computed.
<i>select</i>	LOGICAL. Array, DIMENSION at least max (1, <i>n</i> ) if <i>howmny</i> = 'S' and at least 1 otherwise. Specifies which eigenvectors are to be computed. <i>For real flavors:</i> To obtain the real eigenvector corresponding to the real eigenvalue $\lambda_j$ , set <i>select(j)</i> to .TRUE.. To select the complex eigenvector corresponding to a complex conjugate pair of eigenvalues $\lambda_j$ and $\lambda_{j+1}$ , <i>select(j)</i> and/or <i>select(j+1)</i> must be set .TRUE.; the eigenvector corresponding to the first eigenvalue in the pair is computed. <i>For complex flavors:</i> To obtain the eigenvector corresponding to the eigenvalue $\lambda_j$ , <i>select(j)</i> must be set .TRUE.. <i>select</i> is not referenced if <i>howmny</i> = 'A' or 'O'.
<i>n</i>	INTEGER. The order of the matrix <i>T</i> ( <i>n</i> $\geq$ 0).
<i>t, vl, vr, work</i>	REAL for <i>strevc</i> DOUBLE PRECISION for <i>dtrevc</i> COMPLEX for <i>ctrevc</i> DOUBLE COMPLEX for <i>ztrrevc</i> . Arrays: <i>t(ldt, *)</i> contains the <i>n</i> by <i>n</i> matrix <i>T</i> . The second dimension of <i>t</i> must be at least max(1, <i>n</i> ).

`vl( ldvl, * )`

If `howmny = 'O'` and `job = 'L'` or `'B'`, then `vl` must contain an  $n$  by  $n$  matrix  $Q$  (usually the matrix of Schur vectors returned by `?hseqr`).  
 If `howmny = 'A'` or `'S'`, then `vl` need not be set.

The second dimension of `vl` must be at least  $\max(1, mm)$  if `job = 'L'` or `'B'` and at least 1 if `job = 'R'`.

The array `vl` is not referenced if `job = 'R'`.

`vr( ldvr, * )`

If `howmny = 'O'` and `job = 'R'` or `'B'`, then `vr` must contain an  $n$  by  $n$  matrix  $Q$  (usually the matrix of Schur vectors returned by `?hseqr`).  
 If `howmny = 'A'` or `'S'`, then `vr` need not be set.

The second dimension of `vr` must be at least  $\max(1, mm)$  if `job = 'R'` or `'B'` and at least 1 if `job = 'L'`.

The array `vr` is not referenced if `job = 'L'`.

`work(*)` is a workspace array.

**DIMENSION** at least  $\max(1, 3 * n)$  for real flavors and at least  $\max(1, 2 * n)$  for complex flavors.

`ldt`

**INTEGER**. The first dimension of `t`; at least  $\max(1, n)$ .

`ldvl`

**INTEGER**. The first dimension of `vl`.

If `job = 'L'` or `'B'`, `ldvl`  $\geq \max(1, n)$ .

If `job = 'R'`, `ldvl`  $\geq 1$ .

`ldvr`

**INTEGER**. The first dimension of `vr`.

If `job = 'R'` or `'B'`, `ldvr`  $\geq \max(1, n)$ .

If `job = 'L'`, `ldvr`  $\geq 1$ .

`mm`

**INTEGER**. The number of columns in the arrays `vl` and/or `vr`. Must be at least  $m$  (the precise number of columns required). If `howmny = 'A'` or `'O'`,  $m = n$ .

If `howmny = 'S'`: for real flavors,  $m$  is obtained by counting 1 for each selected real eigenvector and 2 for each selected complex eigenvector;

for complex flavors,  $m$  is the number of selected eigenvectors (see `select`). Constraint:  $0 \leq m \leq n$ .

*rwork*                    **REAL** for **ctrevc**  
**DOUBLE PRECISION** for **ztrevc**.  
 Workspace array, **DIMENSION** at least max (1, *n*).

### Output Parameters

<i>select</i>	If a complex eigenvector of a real matrix was selected as specified above, then <i>select</i> ( <i>j</i> ) is set to <b>.TRUE.</b> and <i>select</i> ( <i>j</i> +1) to <b>.FALSE.</b>
<i>vl, vr</i>	If <i>job</i> = 'L' or 'B', <i>vl</i> contains the computed left eigenvectors (as specified by <i>howmny</i> and <i>select</i> ). If <i>job</i> = 'R' or 'B', <i>vr</i> contains the computed right eigenvectors (as specified by <i>howmny</i> and <i>select</i> ).  The eigenvectors are stored consecutively in the columns of the array, in the same order as their eigenvalues. <i>For real flavors:</i> corresponding to each real eigenvalue is a real eigenvector, occupying one column; corresponding to each complex conjugate pair of eigenvalues is a complex eigenvector, occupying two columns; the first column holds the real part and the second column holds the imaginary part.
<i>m</i>	<b>INTEGER.</b> <i>For complex flavors:</i> the number of selected eigenvectors. If <i>howmny</i> = 'A' or 'O', <i>m</i> is set to <i>n</i> . <i>For real flavors:</i> the number of columns of <i>vl</i> and/or <i>vr</i> actually used to store the selected eigenvectors. If <i>howmny</i> = 'A' or 'O', <i>m</i> is set to <i>n</i> .
<i>info</i>	<b>INTEGER.</b> If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> th parameter had an illegal value.

### Application Notes

If  $x_i$  is an exact right eigenvector and  $y_i$  is the corresponding computed eigenvector, then the angle  $\theta(y_i, x_i)$  between them is bounded as follows:  
 $\theta(y_i, x_i) \leq (c(n)\epsilon\|T\|_2)/\text{sep}_i$  where  $\text{sep}_i$  is the reciprocal condition number of  $x_i$ . The condition number  $\text{sep}_i$  may be computed by calling **?trsna**.

## ?trsna

*Estimates condition numbers for specified eigenvalues and right eigenvectors of an upper (quasi-) triangular matrix.*

```
call strsna ( job, howmny, select, n, t, ldt, vl, ldvl, vr, ldvr,
  s, sep, mm, m, work, ldwork, iwork, info )
call dtrsna ( job, howmny, select, n, t, ldt, vl, ldvl, vr, ldvr,
  s, sep, mm, m, work, ldwork, iwork, info )
call ctrsna ( job, howmny, select, n, t, ldt, vl, ldvl, vr, ldvr,
  s, sep, mm, m, work, ldwork, rwork, info )
call ztrsna ( job, howmny, select, n, t, ldt, vl, ldvl, vr, ldvr,
  s, sep, mm, m, work, ldwork, rwork, info )
```

### Discussion

This routine estimates condition numbers for specified eigenvalues and/or right eigenvectors of an upper triangular matrix  $T$  (or, for real flavors, upper quasi-triangular matrix  $T$  in canonical Schur form). These are the same as the condition numbers of the eigenvalues and right eigenvectors of an original matrix  $A = ZTZ^H$  (with unitary or, for real flavors, orthogonal  $Z$ ), from which  $T$  may have been derived.

The routine computes the reciprocal of the condition number of an eigenvalue  $\lambda_i$  as  $s_i = |v^H u| / (\|u\|_E \|v\|_E)$ , where  $u$  and  $v$  are the right and left eigenvectors of  $T$ , respectively, corresponding to  $\lambda_i$ . This reciprocal condition number always lies between zero (ill-conditioned) and one (well-conditioned).

An approximate error estimate for a computed eigenvalue  $\lambda_i$  is then given by  $\epsilon \|T\| / s_i$ , where  $\epsilon$  is the *machine precision*.

To estimate the reciprocal of the condition number of the right eigenvector corresponding to  $\lambda_i$ , the routine first calls `?trex` (see [page 5-214](#)) to reorder the eigenvalues so that  $\lambda_i$  is in the leading position:

$$T = Q \begin{bmatrix} \lambda_i & C^H \\ 0 & T_{22} \end{bmatrix} Q^H$$

The reciprocal condition number of the eigenvector is then estimated as  $sep_i$ , the smallest singular value of the matrix  $T_{22} - \lambda_i I$ . This number ranges from zero (ill-conditioned) to very large (well-conditioned).

An approximate error estimate for a computed right eigenvector  $u$  corresponding to  $\lambda_i$  is then given by  $\epsilon \|T\|/sep_i$ .

### Input Parameters

<code>job</code>	CHARACTER*1. Must be ' <code>E</code> ' or ' <code>V</code> ' or ' <code>B</code> '. If <code>job</code> = ' <code>E</code> ', then condition numbers for eigenvalues only are computed. If <code>job</code> = ' <code>V</code> ', then condition numbers for eigenvectors only are computed. If <code>job</code> = ' <code>B</code> ', then condition numbers for both eigenvalues and eigenvectors are computed.
<code>howmny</code>	CHARACTER*1. Must be ' <code>A</code> ' or ' <code>S</code> '. If <code>howmny</code> = ' <code>A</code> ', then the condition numbers for all eigenpairs are computed. If <code>howmny</code> = ' <code>S</code> ', then condition numbers for selected eigenpairs (as specified by <code>select</code> ) are computed.
<code>select</code>	LOGICAL. Array, DIMENSION at least max (1, <code>n</code> ) if <code>howmny</code> = ' <code>S</code> ' and at least 1 otherwise. Specifies the eigenpairs for which condition numbers are to be computed if <code>howmny</code> = ' <code>S</code> '. <i>For real flavors:</i> To select condition numbers for the eigenpair corresponding to the real eigenvalue $\lambda_j$ , <code>select(j)</code> must be set .TRUE.; to select condition numbers for the

eigenpair corresponding to a complex conjugate pair of eigenvalues  $\lambda_j$  and  $\lambda_{j+1}$ , `select(j)` and/or `select(j+1)` must be set `.TRUE.`.

*For complex flavors:*

To select condition numbers for the eigenpair corresponding to the eigenvalue  $\lambda_j$ , `select(j)` must be set `.TRUE.`

`select` is not referenced if `howmny = 'A'`.

`n` **INTEGER**. The order of the matrix  $T$  ( $n \geq 0$ ).

`t, vl, vr, work` **REAL** for `strsna`  
**DOUBLE PRECISION** for `dtrsna`  
**COMPLEX** for `ctrsna`  
**DOUBLE COMPLEX** for `ztrsna`.

Arrays:

`t(ldt,*)` contains the `n` by `n` matrix  $T$ .

The second dimension of `t` must be at least `max(1, n)`.

`vl(ldvl,*)`

If `job = 'E'` or `'B'`, then `vl` must contain the left eigenvectors of  $T$  (or of any matrix  $QTQ^H$  with  $Q$  unitary or orthogonal) corresponding to the eigenpairs specified by `howmny` and `select`. The eigenvectors must be stored in consecutive columns of `vl`, as returned by `?trevc` or `?hsein`.

The second dimension of `vl` must be at least `max(1, mm)` if `job = 'E'` or `'B'` and at least 1 if `job = 'V'`.

The array `vl` is not referenced if `job = 'V'`.

`vr(ldvr,*)`

If `job = 'E'` or `'B'`, then `vr` must contain the right eigenvectors of  $T$  (or of any matrix  $QTQ^H$  with  $Q$  unitary or orthogonal) corresponding to the eigenpairs specified by `howmny` and `select`. The eigenvectors must be stored in consecutive columns of `vr`, as returned by `?trevc` or `?hsein`.

The second dimension of `vr` must be at least `max(1, mm)` if `job = 'E'` or `'B'` and at least 1 if `job = 'V'`.

The array `vr` is not referenced if `job = 'V'`.

<i>work( ldwork, * )</i>	is a workspace array.
The second dimension of <i>work</i> must be at least $\max(1, n+1)$ for complex flavors and at least $\max(1, n+6)$ for real flavors if <i>job</i> = 'V' or 'B'; at least 1 if <i>job</i> = 'E'.	
The array <i>work</i> is not referenced if <i>job</i> = 'E'.	
<i>ldt</i>	<b>INTEGER.</b> The first dimension of <i>t</i> ; at least $\max(1, n)$ .
<i>ldvl</i>	<b>INTEGER.</b> The first dimension of <i>vl</i> . If <i>job</i> = 'E' or 'B', <i>ldvl</i> $\geq \max(1, n)$ . If <i>job</i> = 'V', <i>ldvl</i> $\geq 1$ .
<i>ldvr</i>	<b>INTEGER.</b> The first dimension of <i>vr</i> . If <i>job</i> = 'E' or 'B', <i>ldvr</i> $\geq \max(1, n)$ . If <i>job</i> = 'R', <i>ldvr</i> $\geq 1$ .
<i>mm</i>	<b>INTEGER.</b> The number of elements in the arrays <i>s</i> and <i>sep</i> , and the number of columns in <i>vl</i> and <i>vr</i> (if used). Must be at least <i>n</i> (the precise number required). If <i>howmny</i> = 'A', <i>m</i> = <i>n</i> ; if <i>howmny</i> = 'S', for real flavors <i>m</i> is obtained by counting 1 for each selected real eigenvalue and 2 for each selected complex conjugate pair of eigenvalues. for complex flavors <i>m</i> is the number of selected eigenpairs (see <i>select</i> ). Constraint: $0 \leq m \leq n$ .
<i>ldwork</i>	<b>INTEGER.</b> The first dimension of <i>work</i> . If <i>job</i> = 'V' or 'B', <i>ldwork</i> $\geq \max(1, n)$ . If <i>job</i> = 'E', <i>ldwork</i> $\geq 1$ .
<i>rwork</i>	<b>REAL</b> for <i>ctrsna</i> , <i>ztrsna</i> . Array, <b>DIMENSION</b> at least $\max(1, n)$ .
<i>iwork</i>	<b>INTEGER</b> for <i>strsna</i> , <i>dtrsna</i> . Array, <b>DIMENSION</b> at least $\max(1, n)$ .

## Output Parameters

<i>s</i>	<b>REAL</b> for single-precision flavors <b>DOUBLE PRECISION</b> for double-precision flavors. Array, <b>DIMENSION</b> at least $\max(1, mm)$ if <i>job</i> = 'E' or 'B' and at least 1 if <i>job</i> = 'V'.
----------	--

Contains the reciprocal condition numbers of the selected eigenvalues if  $\text{job} = \text{'E'}$  or  $\text{'B'}$ , stored in consecutive elements of the array. Thus  $s(j)$ ,  $\text{sep}(j)$  and the  $j$ th columns of  $\text{vl}$  and  $\text{vr}$  all correspond to the same eigenpair (but not in general the  $j$ th eigenpair unless all eigenpairs have been selected). *For real flavors:* For a complex conjugate pair of eigenvalues, two consecutive elements of  $S$  are set to the same value.

The array  $s$  is not referenced if  $\text{job} = \text{'V'}$ .

$\text{sep}$

`REAL` for single-precision flavors

`DOUBLE PRECISION` for double-precision flavors.

Array, `DIMENSION` at least  $\max(1, mm)$

if  $\text{job} = \text{'V'}$  or  $\text{'B'}$  and at least 1 if  $\text{job} = \text{'E'}$ .

Contains the estimated reciprocal condition numbers of the selected right eigenvectors if  $\text{job} = \text{'V'}$  or  $\text{'B'}$ , stored in consecutive elements of the array.

*For real flavors:* for a complex eigenvector, two consecutive elements of  $\text{sep}$  are set to the same value; if the eigenvalues cannot be reordered to compute  $\text{sep}(j)$ , then  $\text{sep}(j)$  is set to zero; this can only occur when the true value would be very small anyway.

The array  $\text{sep}$  is not referenced if  $\text{job} = \text{'E'}$ .

$m$

`INTEGER`.

*For complex flavors:* the number of selected eigenpairs.

If  $\text{howmny} = \text{'A'}$ ,  $m$  is set to  $n$ .

*For real flavors:* the number of elements of  $s$  and/or  $\text{sep}$  actually used to store the estimated condition numbers.

If  $\text{howmny} = \text{'A'}$ ,  $m$  is set to  $n$ .

$\text{info}$

`INTEGER`.

If  $\text{info} = 0$ , the execution is successful.

If  $\text{info} = -i$ , the  $i$ th parameter had an illegal value.

## Application Notes

The computed values  $\text{sep}_i$  may overestimate the true value, but seldom by a factor of more than 3.

## ?trexc

*Reorders the Schur factorization of a general matrix.*

---

```
call strexc ( compq, n, t, ldt, q, ldq, ifst, ilst, work, info )
call dtrexc ( compq, n, t, ldt, q, ldq, ifst, ilst, work, info )
call ctrexc ( compq, n, t, ldt, q, ldq, ifst, ilst, info )
call ztrexc ( compq, n, t, ldt, q, ldq, ifst, ilst, info )
```

### Discussion

This routine reorders the Schur factorization of a general matrix  $A=QTQ^H$ , so that the diagonal element or block of  $T$  with row index *ifst* is moved to row *ilst*.

The reordered Schur form  $S$  is computed by an unitary (or, for real flavors, orthogonal) similarity transformation:  $S = Z^HTZ$ . Optionally the updated matrix  $P$  of Schur vectors is computed as  $P = QZ$ , giving  $A=PSP^H$ .

### Input Parameters

<i>compq</i>	CHARACTER*1. Must be ' <b>V</b> ' or ' <b>N</b> '. If <i>compq</i> ='V', then the Schur vectors ( $Q$ ) are updated. If <i>compq</i> ='N', then no Schur vectors are updated.
<i>n</i>	INTEGER. The order of the matrix $T$ ( <i>n</i> $\geq$ 0).
<i>t, q</i>	REAL for <b>strexc</b> DOUBLE PRECISION for <b>dtrexc</b> COMPLEX for <b>ctrexc</b> DOUBLE COMPLEX for <b>ztrexc</b> . Arrays: <i>t(ldt,*)</i> contains the <i>n</i> by <i>n</i> matrix $T$ . The second dimension of <i>t</i> must be at least $\max(1, n)$ .
<i>q(ldq,*)</i>	If <i>compq</i> ='V', then <i>q</i> must contain $Q$ (Schur vectors). If <i>compq</i> ='N', then <i>q</i> is not referenced.

---

	The second dimension of $q$ must be at least $\max(1, n)$ if $\text{compq} = 'V'$ and at least 1 if $\text{compq} = 'N'$ .
$ldt$	<b>INTEGER.</b> The first dimension of $t$ ; at least $\max(1, n)$ .
$ldq$	<b>INTEGER.</b> The first dimension of $q$ , If $\text{compq} = 'N'$ , then $ldq \geq 1$ . If $\text{compq} = 'V'$ , then $ldq \geq \max(1, n)$ .
$ifst, ilst$	<b>INTEGER.</b> $1 \leq ifst \leq n; 1 \leq ilst \leq n$ . Must specify the reordering of the diagonal elements (or blocks, which is possible for real flavors) of the matrix $T$ . The element (or block) with row index $ifst$ is moved to row $ilst$ by a sequence of exchanges between adjacent elements (or blocks).
$work$	<b>REAL</b> for <b>strexc</b> <b>DOUBLE PRECISION</b> for <b>dtrexc</b> . Array, <b>DIMENSION</b> at least $\max(1, n)$ .

## Output Parameters

$t$	Overwritten by the updated matrix $S$ .
$q$	If $\text{compq} = 'V'$ , $q$ contains the updated matrix of Schur vectors.
$ifst, ilst$	Overwritten for real flavors only. If $ifst$ pointed to the second row of a 2 by 2 block on entry, it is changed to point to the first row; $ilst$ always points to the first row of the block in its final position (which may differ from its input value by $\pm 1$ ).
$info$	<b>INTEGER.</b> If $info = 0$ , the execution is successful. If $info = -i$ , the $i$ th parameter had an illegal value.

## Application Notes

The computed matrix  $S$  is exactly similar to a matrix  $T + E$ , where  $\|E\|_2 = O(\epsilon) \|T\|_2$ , and  $\epsilon$  is the machine precision.

Note that if a 2 by 2 diagonal block is involved in the re-ordering, its off-diagonal elements are in general changed; the diagonal elements and the eigenvalues of the block are unchanged unless the block is sufficiently ill-conditioned, in which case they may be noticeably altered. It is possible for a 2 by 2 block to break into two 1 by 1 blocks, that is, for a pair of complex eigenvalues to become purely real.

The values of eigenvalues however are never changed by the re-ordering.

The approximate number of floating-point operations is

for real flavors:	$6n(\text{ifst}-\text{ilst})$ if $\text{compq} = 'N'$ ; $12n(\text{ifst}-\text{ilst})$ if $\text{compq} = 'V'$ ;
for complex flavors:	$20n(\text{ifst}-\text{ilst})$ if $\text{compq} = 'N'$ ; $40n(\text{ifst}-\text{ilst})$ if $\text{compq} = 'V'$ .

---

## ?trsen

*Reorders the Schur factorization of a matrix  
and (optionally) computes the reciprocal  
condition numbers and invariant subspace for  
the selected cluster of eigenvalues.*

---

```
call strsen (job, compq, select, n, t, ldt, q, ldq, wr, wi, m, s,  
           sep, work, lwork, iwork, liwork, info)  
call dtrsen (job, compq, select, n, t, ldt, q, ldq, wr, wi, m, s,  
           sep, work, lwork, iwork, liwork, info)  
call ctrsen (job, compq, select, n, t, ldt, q, ldq, w, m, s,  
           sep, work, lwork, info)  
call ztrsen (job, compq, select, n, t, ldt, q, ldq, w, m, s,  
           sep, work, lwork, info)
```

### Discussion

This routine reorders the Schur factorization of a general matrix  $A = QTQ^H$  so that a selected cluster of eigenvalues appears in the leading diagonal elements (or, for real flavors, diagonal blocks) of the Schur form.

The reordered Schur form  $R$  is computed by an unitary(orthogonal) similarity transformation:  $R = Z^H TZ$ . Optionally the updated matrix  $P$  of Schur vectors is computed as  $P = QZ$ , giving  $A = PRP^H$ .

Let

$$R = \begin{bmatrix} T_{11} & T_{12} \\ 0 & T_{13} \end{bmatrix}$$

where the selected eigenvalues are precisely the eigenvalues of the leading  $m$  by  $m$  submatrix  $T_{11}$ . Let  $P$  be correspondingly partitioned as  $(Q_1 Q_2)$  where  $Q_1$  consists of the first  $m$  columns of  $Q$ . Then  $AQ_1 = Q_1T_{11}$ , and so the  $m$  columns of  $Q_1$  form an orthonormal basis for the invariant subspace corresponding to the selected cluster of eigenvalues.

Optionally the routine also computes estimates of the reciprocal condition numbers of the average of the cluster of eigenvalues and of the invariant subspace.

### Input Parameters

<i>job</i>	CHARACTER*1. Must be ' <b>N</b> ' or ' <b>E</b> ' or ' <b>V</b> ' or ' <b>B</b> '. If <i>job</i> ='N', then no condition numbers are required. If <i>job</i> ='E', then only the condition number for the cluster of eigenvalues is computed. If <i>job</i> ='V', then only the condition number for the invariant subspace is computed. If <i>job</i> ='B', then condition numbers for both the cluster and the invariant subspace are computed.
<i>compq</i>	CHARACTER*1. Must be ' <b>V</b> ' or ' <b>N</b> '. If <i>compq</i> ='V', then $Q$ of the Schur vectors is updated. If <i>compq</i> ='N', then no Schur vectors are updated.
<i>select</i>	LOGICAL. Array, DIMENSION at least max (1, <i>n</i> ). Specifies the eigenvalues in the selected cluster. To select an eigenvalue $\lambda_j$ , <i>select(j)</i> must be .TRUE.. <i>For real flavors:</i> to select a complex conjugate pair of eigenvalues $\lambda_j$ and $\lambda_{j+1}$ (corresponding 2 by 2 diagonal

block), `select(j)` and/or `select(j+1)` must be `.TRUE.`; the complex conjugate  $\lambda_j$  and  $\lambda_{j+1}$  must be either both included in the cluster or both excluded.

*n**t, q, work*`INTEGER`. The order of the matrix  $T$  ( $n \geq 0$ ).`REAL` for `strsen``DOUBLE PRECISION` for `dtrsen``COMPLEX` for `ctrsen``DOUBLE COMPLEX` for `ztrsen`.

Arrays:

*t (lvt, \*)* The *n* by *n* *T*.The second dimension of *t* must be at least `max(1, n)`.*q (ldq, \*)*If `compq='V'`, then *q* must contain *Q* of Schur vectors.If `compq='N'`, then *q* is not referenced.The second dimension of *q* must be at least `max(1, n)` if `compq='V'` and at least 1 if `compq='N'`.*work (lwork)* is a workspace array.For complex flavors: the array *work* is not referenced if `job='N'`.The actual amount of workspace required cannot exceed  $n^2/4$  if `job='E'` or  $n^2/2$  if `job='V'` or '`B`'.*lvt*`INTEGER`. The first dimension of *t*; at least `max(1, n)`.*ldq*`INTEGER`. The first dimension of *q*;If `compq='N'`, then *ldq*  $\geq 1$ .If `compq='V'`, then *ldq*  $\geq \max(1, n)$ .*lwork*`INTEGER`. The dimension of the array *work*.If `job='V'` or '`B`', *lwork*  $\geq \max(1, 2m(n-m))$ .If `job='E'`, then *lwork*  $\geq \max(1, m(n-m))$ If `job='N'`, then *lwork*  $\geq 1$  for complex flavors and *lwork*  $\geq \max(1, n)$  for real flavors.*iwork*`INTEGER`.*iwork (liwork)* is a workspace array.The array *iwork* is not referenced if `job='N'` or '`E`'.The actual amount of workspace required cannot exceed  $n^2/2$  if `job='V'` or '`B`'.

*liwork*      INTEGER.  
 The dimension of the array *iwork*.  
 If *job* = 'V' or 'B', *liwork*  $\geq \max(1, 2m(n-m))$ .  
 If *job* = 'E' or 'E', *liwork*  $\geq 1$ .

### Output Parameters

*t*      Overwritten by the updated matrix *R*.  
*q*      If *compq* = 'V', *q* contains the updated matrix of Schur vectors; the first *m* columns of the *Q* form an orthogonal basis for the specified invariant subspace.  
*w*      COMPLEX for *ctrsen*  
 DOUBLE COMPLEX for *ztrsen*.  
 Array, DIMENSION at least  $\max(1, n)$ .  
 The recorded eigenvalues of *R*. The eigenvalues are stored in the same order as on the diagonal of *R*.  
*wr, wi*      REAL for *strsen*  
 DOUBLE PRECISION for *dtrsen*  
 Arrays, DIMENSION at least  $\max(1, n)$ .  
 Contain the real and imaginary parts, respectively, of the reordered eigenvalues of *R*. The eigenvalues are stored in the same order as on the diagonal of *R*. Note that if a complex eigenvalue is sufficiently ill-conditioned, then its value may differ significantly from its value before reordering.  
*m*      INTEGER.  
 For complex flavors: the number of the specified invariant subspaces, which is the same as the number of selected eigenvalues (see *select*).  
 For real flavors: the dimension of the specified invariant subspace. The value of *m* is obtained by counting 1 for each selected real eigenvalue and 2 for each selected complex conjugate pair of eigenvalues (see *select*).  
 Constraint:  $0 \leq m \leq n$ .

<i>s</i>	<code>REAL</code> for single-precision flavors <code>DOUBLE PRECISION</code> for double-precision flavors. If <i>job</i> = 'E' or 'B', <i>s</i> is a lower bound on the reciprocal condition number of the average of the selected cluster of eigenvalues. If <i>m</i> = 0 or <i>n</i> , then <i>s</i> = 1. <i>For real flavors:</i> if <i>info</i> = 1, then <i>s</i> is set to zero. <i>s</i> is not referenced if <i>job</i> = 'N' or 'V'.
<i>sep</i>	<code>REAL</code> for single-precision flavors <code>DOUBLE PRECISION</code> for double-precision flavors. If <i>job</i> = 'V' or 'B', <i>sep</i> is the estimated reciprocal condition number of the specified invariant subspace. If <i>m</i> = 0 or <i>n</i> , then <i>sep</i> = $\ T\ $ . <i>For real flavors:</i> if <i>info</i> = 1, then <i>sep</i> is set to zero. <i>sep</i> is not referenced if <i>job</i> = 'N' or 'E'.
<i>info</i>	<code>INTEGER</code> . If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> th parameter had an illegal value.

## Application Notes

The computed matrix *R* is exactly similar to a matrix *T* + *E*, where  $\|E\|_2 = O(\epsilon)\|T\|_2$ , and  $\epsilon$  is the machine precision.

The computed *s* cannot underestimate the true reciprocal condition number by more than a factor of  $(\min(m, n-m))^{1/2}$ ; *sep* may differ from the true value by  $(m*(n-m)^2)^{1/2}$ . The angle between the computed invariant subspace and the true subspace is  $O(\epsilon) \|A\|_2 / \text{sep}$ .

Note that if a 2 by 2 diagonal block is involved in the re-ordering, its off-diagonal elements are in general changed; the diagonal elements and the eigenvalues of the block are unchanged unless the block is sufficiently ill-conditioned, in which case they may be noticeably altered. It is possible for a 2 by 2 block to break into two 1 by 1 blocks, that is, for a pair of complex eigenvalues to become purely real.

The values of eigenvalues however are never changed by the re-ordering.

## ?trsyl

Solves Sylvester's equation for real  
quasi-triangular or complex triangular  
matrices.

```
call strsyl ( trana,tranb,isgn,m,n,a,lda,b,ldb,c,ldc,scale,info )
call dtrsyl ( trana,tranb,isgn,m,n,a,lda,b,ldb,c,ldc,scale,info )
call ctrsyl ( trana,tranb,isgn,m,n,a,lda,b,ldb,c,ldc,scale,info )
call ztrsyl ( trana,tranb,isgn,m,n,a,lda,b,ldb,c,ldc,scale,info )
```

### Discussion

This routine solves the Sylvester matrix equation  $\text{op}(A)X \pm X\text{op}(B) = \alpha C$ , where  $\text{op}(A) = A$  or  $A^H$ , and the matrices  $A$  and  $B$  are upper triangular (or, for real flavors, upper quasi-triangular in canonical Schur form);  $\alpha \leq 1$  is a scale factor determined by the routine to avoid overflow in  $X$ ;  $A$  is  $m$  by  $m$ ,  $B$  is  $n$  by  $n$ , and  $C$  and  $X$  are both  $m$  by  $n$ . The matrix  $X$  is obtained by a straightforward process of back substitution.

The equation has a unique solution if and only if  $\alpha_i \pm \beta_i \neq 0$ , where  $\{\alpha_i\}$  and  $\{\beta_i\}$  are the eigenvalues of  $A$  and  $B$ , respectively, and the sign (+ or -) is the same as that used in the equation to be solved.

### Input Parameters

<i>trana</i>	CHARACTER*1. Must be ' <b>N</b> ' or ' <b>T</b> ' or ' <b>C</b> '. If <i>trana</i> = ' <b>N</b> ', then $\text{op}(A) = A$ . If <i>trana</i> = ' <b>T</b> ', then $\text{op}(A) = A^T$ (real flavors only). If <i>trana</i> = ' <b>C</b> ', then $\text{op}(A) = A^H$ .
<i>tranb</i>	CHARACTER*1. Must be ' <b>N</b> ' or ' <b>T</b> ' or ' <b>C</b> '. If <i>tranb</i> = ' <b>N</b> ', then $\text{op}(B) = B$ . If <i>tranb</i> = ' <b>T</b> ', then $\text{op}(B) = B^T$ (real flavors only). If <i>tranb</i> = ' <b>C</b> ', then $\text{op}(B) = B^H$ .
<i>isgn</i>	INTEGER. Indicates the form of the Sylvester equation. If <i>isgn</i> = +1, $\text{op}(A)X + X\text{op}(B) = \alpha C$ . If <i>isgn</i> = -1, $\text{op}(A)X - X\text{op}(B) = \alpha C$ .

<i>m</i>	<b>INTEGER.</b> The order of <i>A</i> , and the number of rows in <i>X</i> and <i>C</i> ( <i>m</i> ≥ 0).
<i>n</i>	<b>INTEGER.</b> The order of <i>B</i> , and the number of columns in <i>X</i> and <i>C</i> ( <i>n</i> ≥ 0).
<i>a, b, c</i>	<b>REAL</b> for <i>strsyl</i> <b>DOUBLE PRECISION</b> for <i>dtrsyl</i> <b>COMPLEX</b> for <i>ctrssyl</i> <b>DOUBLE COMPLEX</b> for <i>ztrsyl</i> .
	Arrays:
<i>a( lda, * )</i>	contains the matrix <i>A</i> . The second dimension of <i>a</i> must be at least $\max(1, m)$ .
<i>b( ldb, * )</i>	contains the matrix <i>B</i> . The second dimension of <i>b</i> must be at least $\max(1, n)$ .
<i>c( ldc, * )</i>	contains the matrix <i>C</i> . The second dimension of <i>c</i> must be at least $\max(1, n)$ .
<i>lda</i>	<b>INTEGER.</b> The first dimension of <i>a</i> ; at least $\max(1, m)$ .
<i>ldb</i>	<b>INTEGER.</b> The first dimension of <i>b</i> ; at least $\max(1, n)$ .
<i>ldc</i>	<b>INTEGER.</b> The first dimension of <i>c</i> ; at least $\max(1, n)$ .

## Output Parameters

<i>c</i>	Overwritten by the solution matrix <i>X</i> .
<i>scale</i>	<b>REAL</b> for single-precision flavors <b>DOUBLE PRECISION</b> for double-precision flavors. The value of the scale factor $\alpha$ .
<i>info</i>	<b>INTEGER.</b> If <i>info</i> = 0, the execution is successful. If <i>info</i> = <i>i</i> , the <i>i</i> th parameter had an illegal value. If <i>info</i> = 1, <i>A</i> and <i>B</i> have common or close eigenvalues perturbed values were used to solve the equation.

## Application Notes

Let *X* be the exact, *Y* the corresponding computed solution, and *R* the residual matrix:  $R = C - (AY \pm YB)$ . Then the residual is always small:

$$\|R\|_F = O(\epsilon) (\|A\|_F + \|B\|_F) \|Y\|_F.$$

However,  $Y$  is not necessarily the exact solution of a slightly perturbed equation; in other words, the solution is not backwards stable.

For the forward error, the following bound holds:

$$\|Y - X\|_F \leq \|R\|_F / \text{sep}(A, B)$$

but this may be a considerable overestimate. See [[Golub96](#)] for a definition of  $\text{sep}(A, B)$ .

The approximate number of floating-point operations for real flavors is  $m^* n^*(m + n)$ . For complex flavors it is 4 times greater.

## Generalized Nonsymmetric Eigenvalue Problems

This section describes LAPACK routines for solving generalized nonsymmetric eigenvalue problems, reordering the generalized Schur factorization of a pair of matrices, as well as performing a number of related computational tasks.

A *generalized nonsymmetric eigenvalue problem* is as follows: given a pair of nonsymmetric (or non-Hermitian) n-by-n matrices  $A$  and  $B$ , find the *generalized eigenvalues*  $\lambda$  and the corresponding *generalized eigenvectors*  $x$  and  $y$  that satisfy the equations

$$Ax = \lambda Bx \quad (\text{right generalized eigenvectors } x)$$

and

$$y^H A = \lambda y^H B \quad (\text{left generalized eigenvectors } y).$$

[Table 5-6](#) lists LAPACK routines used to solve the generalized nonsymmetric eigenvalue problems and the generalized Sylvester equation.

**Table 5-6 Computational Routines for Solving Generalized Nonsymmetric Eigenvalue Problems**

Routine name	Operation performed
<a href="#">?gghrd</a>	Reduces a pair of matrices to generalized upper Hessenberg form using orthogonal/unitary transformations.
<a href="#">?ggbal</a>	Balances a pair of general real or complex matrices.
<a href="#">?ggbak</a>	Forms the right or left eigenvectors of a generalized eigenvalue problem.
<a href="#">?hgeqz</a>	Implements the QZ method for finding the generalized eigenvalues $w_i$ of the equation $\det(A - w_i^* B) = 0$ .
<a href="#">?tgevc</a>	Computes some or all of the right and/or left generalized eigenvectors of a pair of upper triangular matrices
<a href="#">?tgexc</a>	Reorders the generalized Shur decomposition of a pair of matrices (A,B) so that one diagonal block of (A,B) moves to another row index.
<a href="#">?tgsen</a>	Reorders the generalized Shur decomposition of a pair of matrices (A,B) so that a selected cluster of eigenvalues appears in the leading diagonal blocks of (A,B).
<a href="#">?tgsvl</a>	Solves the generalized Sylvester equation.
<a href="#">?tgnsa</a>	Estimates reciprocal condition numbers for specified eigenvalues and/or eigenvectors of a pair of matrices in generalized real Shur canonical form.

## ?gghrd

*Reduces a pair of matrices to generalized upper Hessenberg form using orthogonal/unitary transformations.*

```
call sgghrd ( compq, compz, n, ilo, ihi, a, lda, b, ldb, q, ldq,
              z, ldz, info )
call dgghrd ( compq, compz, n, ilo, ihi, a, lda, b, ldb, q, ldq,
              z, ldz, info )
call cgghrd ( compq, compz, n, ilo, ihi, a, lda, b, ldb, q, ldq,
              z, ldz, info )
call zgghrd ( compq, compz, n, ilo, ihi, a, lda, b, ldb, q, ldq,
              z, ldz, info )
```

### Discussion

This routine reduces a pair of real/complex matrices (A,B) to generalized upper Hessenberg form using orthogonal/unitary transformations, where A is a general matrix and B is upper triangular:

$$Q^H A Z = H \quad \text{and} \quad Q^H B Z = T,$$

where H is upper Hessenberg, T is upper triangular, and Q and Z are orthogonal/unitary.

The orthogonal/unitary matrices Q and Z are determined as products of Givens rotations. They may either be formed explicitly, or they may be postmultiplied into input matrices  $Q_I$  and  $Z_I$ , so that

$$Q_I A Z_I^H = (Q_I Q) H (Z_I Z)^H$$

$$Q_I B Z_I^H = (Q_I Q) T (Z_I Z)^H$$

### Input Parameters

*compq*      CHARACTER\*1. Must be 'N', 'I', or 'V'.  
 If *compq* = 'N', matrix Q is not computed.  
 If *compq* = 'I', Q is initialized to the unit matrix, and the orthogonal/unitary matrix Q is returned;

<i>compz</i>	If <i>compq</i> = 'V', $Q$ must contain an orthogonal/unitary matrix $Q_1$ on entry, and the product $Q_1 Q$ is returned. CHARACTER*1. Must be 'N', 'I', or 'V'.
<i>n</i>	If <i>compz</i> = 'N', matrix $Z$ is not computed. If <i>compz</i> = 'I', $Z$ is initialized to the unit matrix, and the orthogonal/unitary matrix $Z$ is returned; If <i>compz</i> = 'V', $Z$ must contain an orthogonal/unitary matrix $Z_1$ on entry, and the product $Z_1 Z$ is returned.
<i>ilo, ihi</i>	INTEGER. The order of the matrices $A$ and $B$ ( <i>n</i> $\geq 0$ ). INTEGER. It is assumed that $A$ is already upper triangular in rows and columns 1: <i>ilo</i> -1 and <i>ihi</i> +1: <i>n</i> . Values of <i>ilo</i> and <i>ihi</i> are normally set by a previous call to ?ggbal; otherwise they should be set to 1 and <i>n</i> respectively. Constraint: If <i>n</i> > 0, then 1 $\leq$ <i>ilo</i> $\leq$ <i>ihi</i> $\leq$ <i>n</i> ; if <i>n</i> = 0, then <i>ilo</i> = 1 and <i>ihi</i> = 0.
<i>a, b, q, z</i>	REAL for sgghrd DOUBLE PRECISION for dgghrd COMPLEX for cgghrd DOUBLE COMPLEX for zgghrd. Arrays: <i>a</i> ( <i>lda</i> ,*) contains the matrix $A$ . The second dimension of <i>a</i> must be at least max(1, <i>n</i> ). <i>b</i> ( <i>ldb</i> ,*) contains the matrix $B$ . The second dimension of <i>b</i> must be at least max(1, <i>n</i> ). <i>q</i> ( <i>ldq</i> ,*) If <i>compq</i> = 'N', then <i>q</i> is not referenced. If <i>compq</i> = 'I', then, on entry, <i>q</i> need not be set. If <i>compq</i> = 'V', then <i>q</i> must contain the orthogonal/unitary matrix $Q_1$ . The second dimension of <i>q</i> must be at least max(1, <i>n</i> ). <i>z</i> ( <i>ldz</i> ,*) If <i>compq</i> = 'N', then <i>z</i> is not referenced. If <i>compq</i> = 'I', then, on entry, <i>z</i> need not be set.

If  $\text{compq} = 'V'$ , then  $\text{z}$  must contain the orthogonal/unitary matrix  $Z_1$ .  
The second dimension of  $\text{z}$  must be at least  $\max(1, n)$ .

***lda***      INTEGER. The first dimension of  $\text{a}$ ; at least  $\max(1, n)$ .

***ldb***      INTEGER. The first dimension of  $\text{b}$ ; at least  $\max(1, n)$ .

***ldq***      INTEGER. The first dimension of  $\text{q}$ ;  
If  $\text{compq} = 'N'$ , then  $\text{ldq} \geq 1$ .  
If  $\text{compq} = 'I'$  or ' $V$ ', then  $\text{ldq} \geq \max(1, n)$ .

***ldz***      INTEGER. The first dimension of  $\text{z}$ ;  
If  $\text{compq} = 'N'$ , then  $\text{ldz} \geq 1$ .  
If  $\text{compq} = 'I'$  or ' $V$ ', then  $\text{ldz} \geq \max(1, n)$ .

### Output Parameters

***a***      On exit, the upper triangle and the first subdiagonal of  $A$  are overwritten with the upper Hessenberg matrix  $H$ , and the rest is set to zero.

***b***      On exit, overwritten by the upper triangular matrix  $T$ .  
The elements below the diagonal are set to zero

***q***      If  $\text{compq} = 'I'$ , then  $\text{q}$  contains the orthogonal/unitary matrix  $Q$ , where  $Q^H$  is the product of the Givens transformations which are applied to  $A$  and  $B$  on the left;  
If  $\text{compq} = 'V'$ , then  $\text{q}$  is overwritten by the product  $Q_1 Q$ .

***z***      If  $\text{compq} = 'I'$ , then  $\text{z}$  contains the orthogonal/unitary matrix  $Z$ , which is the product of the Givens transformations which are applied to  $A$  and  $B$  on the right;  
If  $\text{compq} = 'V'$ , then  $\text{z}$  is overwritten by the product  $Z_1 Z$ .

***info***      INTEGER.  
If  $\text{info} = 0$ , the execution is successful.  
If  $\text{info} = -i$ , the  $i$ th parameter had an illegal value.

## ?ggbal

Balances a pair of general real or complex matrices.

---

```
call sggbal ( job, n, a, lda, b, ldb, ilo, ihi, lscale, rscale,
              work, info )
call dggbal ( job, n, a, lda, b, ldb, ilo, ihi, lscale, rscale,
              work, info )
call cggbal ( job, n, a, lda, b, ldb, ilo, ihi, lscale, rscale,
              work, info )
call zggbal ( job, n, a, lda, b, ldb, ilo, ihi, lscale, rscale,
              work, info )
```

### Discussion

This routine balances a pair of general real/complex matrices ( $A, B$ ). This involves, first, permuting  $A$  and  $B$  by similarity transformations to isolate eigenvalues in the first  $1$  to  $ilo-1$  and last  $ihi+1$  to  $n$  elements on the diagonal; and second, applying a diagonal similarity transformation to rows and columns  $ilo$  to  $ihi$  to make the rows and columns as close in norm as possible. Both steps are optional.

Balancing may reduce the 1-norm of the matrices, and improve the accuracy of the computed eigenvalues and/or eigenvectors in the generalized eigenvalue problem  $Ax = \lambda Bx$ .

### Input Parameters

<i>job</i>	CHARACTER*1. Specifies the operations to be performed on $A$ and $B$ . Must be ' <b>N</b> ' or ' <b>P</b> ' or ' <b>S</b> ' or ' <b>B</b> '. If <i>job</i> = ' <b>N</b> ', then no operations are done; simply set <i>ilo</i> =1, <i>ihi</i> = <i>n</i> , <i>lscale</i> (i)=1.0 and <i>rscale</i> (i)=1.0 for <i>i</i> = 1,..., <i>n</i> . If <i>job</i> = ' <b>P</b> ', then permute only. If <i>job</i> = ' <b>S</b> ', then scale only. If <i>job</i> = ' <b>B</b> ', then both permute and scale.
<i>n</i>	INTEGER. The order of the matrices $A$ and $B$ ( $n \geq 0$ ).

*a, b*                    REAL for sggbal  
DOUBLE PRECISION for dggbal  
COMPLEX for cggbal  
DOUBLE COMPLEX for zggbal.

Arrays:

*a( lda, \* )* contains the matrix *A*.  
 The second dimension of *a* must be at least  $\max(1, n)$ .

*b( ldb, \* )* contains the matrix *B*.  
 The second dimension of *b* must be at least  $\max(1, n)$ .

*lda*                    INTEGER. The first dimension of *a*; at least  $\max(1, n)$ .

*ldb*                    INTEGER. The first dimension of *b*; at least  $\max(1, n)$ .

*work*                    REAL for single precision flavors  
DOUBLE PRECISION for double precision flavors.  
Workspace array, DIMENSION at least  $\max(1, 6n)$ .

## Output Parameters

*a, b*                    Overwritten by the balanced matrices *A* and *B*, respectively. If *job* = 'N', *a* and *b* are not referenced.

*ilo, ihi*                    INTEGER. *ilo* and *ihi* are set to integers such that on exit *a(i, j)=0* and *b(i, j)=0* if *i>j* and *j=1,...,ilo-1* or *i=ihi+1,...,n*.  
 If *job* = 'N' or 'S', then *ilo* = 1 and *ihi* = *n*.

*lscale, rscale*                    REAL for single precision flavors  
DOUBLE PRECISION for double precision flavors.  
Arrays, DIMENSION at least  $\max(1, n)$ .

*lscale* contains details of the permutations and scaling factors applied to the left side of *A* and *B*.  
 If  $P_j$  is the index of the row interchanged with row *j*, and  $D_j$  is the scaling factor applied to row *j*, then

$$\begin{aligned} lscale(j) &= P_j, \text{ for } j = 1, \dots, ilo-1 \\ &= D_j, \text{ for } j = ilo, \dots, ihi \\ &= P_j, \text{ for } j = ihi+1, \dots, n. \end{aligned}$$

*rscale* contains details of the permutations and scaling factors applied to the right side of *A* and *B*.

If  $P_j$  is the index of the column interchanged with column  $j$ , and  $D_j$  is the scaling factor applied to column  $j$ , then

$$\begin{aligned}rscal(\textcolor{red}{j}) &= P_j \text{ , for } j = 1, \dots, \textcolor{red}{ilo}-1 \\&= D_j \text{ , for } j = \textcolor{red}{ilo}, \dots, \textcolor{red}{ih}i \\&= P_j \text{ , for } j = \textcolor{red}{ih}i+1, \dots, n\end{aligned}$$

The order in which the interchanges are made is  $n$  to  $\textcolor{red}{ih}i+1$ , then 1 to  $\textcolor{red}{ilo}-1$ .

*info*

INTEGER.

If *info* = 0, the execution is successful.

If *info* =  $-i$ , the  $i$ th parameter had an illegal value.

## ?ggbak

*Forms the right or left eigenvectors of a generalized eigenvalue problem.*

```
call sggbak(job, side, n, ilo, ihi, lscale, rscale, m, v, ldv, info)
call dggbak(job, side, n, ilo, ihi, lscale, rscale, m, v, ldv, info)
call cggbak(job, side, n, ilo, ihi, lscale, rscale, m, v, ldv, info)
call zggbak(job, side, n, ilo, ihi, lscale, rscale, m, v, ldv, info)
```

### Discussion

This routine forms the right or left eigenvectors of a real/complex generalized eigenvalue problem

$$Ax = \lambda Bx$$

by backward transformation on the computed eigenvectors of the balanced pair of matrices output by [?ggbal](#).

### Input Parameters

<i>job</i>	CHARACTER*1. Specifies the type of backward transformation required. Must be ' <b>N</b> ', ' <b>P</b> ', ' <b>S</b> ', or ' <b>B</b> '.
	If <i>job</i> = ' <b>N</b> ', then no operations are done; return.
	If <i>job</i> = ' <b>P</b> ', then do backward transformation for permutation only.
	If <i>job</i> = ' <b>S</b> ', then do backward transformation for scaling only.
	If <i>job</i> = ' <b>B</b> ', then do backward transformation for both permutation and scaling.
	This argument must be the same as the argument <i>job</i> supplied to <a href="#">?ggbal</a> .
<i>side</i>	CHARACTER*1. Must be ' <b>L</b> ' or ' <b>R</b> '.
	If <i>side</i> = ' <b>L</b> ', then <i>v</i> contains left eigenvectors .
	If <i>side</i> = ' <b>R</b> ', then <i>v</i> contains right eigenvectors .
<i>n</i>	INTEGER. The number of rows of the matrix <i>V</i> ( <i>n</i> $\geq 0$ ).

<i>ilo, ihi</i>	<b>INTEGER.</b> The integers <i>ilo</i> and <i>ihi</i> determined by <b>?gebal.</b> Constraint: If <i>n</i> > 0, then $1 \leq ilo \leq ihi \leq n$ ; if <i>n</i> = 0, then <i>ilo</i> = 1 and <i>ihi</i> = 0.
<i>lscale,rscale</i>	<b>REAL</b> for single precision flavors <b>DOUBLE PRECISION</b> for double precision flavors. Arrays, <b>DIMENSION</b> at least $\max(1, n)$ . The array <i>lscale</i> contains details of the permutations and/or scaling factors applied to the left side of <i>A</i> and <i>B</i> , as returned by <b>?ggbal</b> . The array <i>rscale</i> contains details of the permutations and/or scaling factors applied to the right side of <i>A</i> and <i>B</i> , as returned by <b>?ggbal</b> .
<i>m</i>	<b>INTEGER.</b> The number of columns of the matrix <i>V</i> ( <i>m</i> $\geq 0$ ).
<i>v</i>	<b>REAL</b> for <b>sggbak</b> <b>DOUBLE PRECISION</b> for <b>dggbak</b> <b>COMPLEX</b> for <b>cggbak</b> <b>DOUBLE COMPLEX</b> for <b>zggbak</b> . Array <i>v</i> ( <i>ldv</i> , *). Contains the matrix of right or left eigenvectors to be transformed, as returned by <b>?tgevc</b> . The second dimension of <i>v</i> must be at least $\max(1, m)$ .
<i>ldv</i>	<b>INTEGER.</b> The first dimension of <i>v</i> , at least $\max(1, n)$ .

## Output Parameters

<i>v</i>	Overwritten by the transformed eigenvectors
<i>info</i>	<b>INTEGER.</b> If <i>info</i> = 0, the execution is successful. If <i>info</i> = <i>-i</i> , the <i>i</i> th parameter had an illegal value.

## ?hgeqz

Implements the QZ method for finding the generalized eigenvalues  $w_i$  of the equation  $\det(A - w_i^* B) = 0$ .

```
call shgeqz(job, compq, compz, n, ilo, ihi, a, lda, b, ldb, alphar,
            alphai, beta, q, ldq, z, ldz, work, lwork, info )
call dhgeqz(job, compq, compz, n, ilo, ihi, a, lda, b, ldb, alphar,
            alphai, beta, q, ldq, z, ldz, work, lwork, info )
call chgeqz(job, compq, compz, n, ilo, ihi, a, lda, b, ldb, alpha,
            beta, q, ldq, z, ldz, work, lwork, rwork, info )
call zhgeqz(job, compq, compz, n, ilo, ihi, a, lda, b, ldb, alpha,
            beta, q, ldq, z, ldz, work, lwork, rwork, info )
```

### Discussion

This routine implements a single-/double-shift version (for real flavors) or single-shift version (for complex flavors) of the QZ method for finding the generalized eigenvalues

$\omega_j = (\text{alphar}(j) + i * \text{alphai}(j)) / \text{beta}(j)$  (for `chgeqz`/`zhgeqz`) or  
 $\omega_j = \text{alpha}(j) / \text{beta}(j)$  (for `chgeqz`/`zhgeqz`)  
of the equation

$$\det(A - \omega_j B) = 0$$

For real flavors:

In addition, the pair  $A, B$  may be reduced to generalized Schur form:  $B$  is upper triangular, and  $A$  is block upper triangular, where the diagonal blocks are either 1-by-1 or 2-by-2, the 2-by-2 blocks having complex generalized eigenvalues (see the description of the argument `job`). If `job`='S', then the pair  $(A, B)$  is simultaneously reduced to Schur form by applying one orthogonal transformation (usually called  $Q$ ) on the left and another (usually called  $Z$ ) on the right. The 2-by-2 upper-triangular diagonal blocks of  $B$  corresponding to 2-by-2 blocks of  $A$  will be reduced to positive diagonal matrices. (That is, if  $A(j+1,j)$  is non-zero, then  $B(j+1,j)=B(j,j+1)=0$  and  $B(j,j)$  and  $B(j+1,j+1)$  will be positive). If `job`='E', then at each

iteration, the same transformations are computed, but they are only applied to those parts of  $A$  and  $B$  which are needed to compute *alphar*, *alphai*, and *betar*.

*For complex flavors:*

If *job* = 'S', then the pair  $(A, B)$  is simultaneously reduced to Schur form (that is,  $A$  and  $B$  are both upper triangular) by applying one unitary transformation (usually called  $Q$ ) on the left and another (usually called  $Z$ ) on the right. The diagonal elements of  $A$  are then *alpha*(1),..., *alpha*(*n*) and of  $B$  are *beta*(1),..., *beta*(*n*).

For all function flavors, If *job* = 'S' and *compq* and *compz* are 'V' or 'I', then the orthogonal/unitary transformations used to reduce  $(A, B)$  are accumulated into the arrays  $Q$  and  $Z$  such that:

$$\begin{aligned} Q(\text{in}) * A(\text{in}) * Z(\text{in})^H &= Q(\text{out}) * A(\text{out}) * Z(\text{out})^H \\ Q(\text{in}) * B(\text{in}) * Z(\text{in})^H &= Q(\text{out}) * B(\text{out}) * Z(\text{out})^H. \end{aligned}$$

## Input Parameters

<i>job</i>	CHARACTER*1. Specifies the operations to be performed. Must be 'E' or 'S'. If <i>job</i> = 'E', then compute only <i>alphar</i> , <i>alphai</i> (for real flavors) or <i>alpha</i> (for complex flavors) and <i>beta</i> . $A$ and $B$ will not necessarily be put into generalized Shur form. If <i>job</i> = 'S', then put $A$ and $B$ into generalized Shur form and compute <i>alphar</i> , <i>alphai</i> (for real flavors) or <i>alpha</i> (for complex flavors) and <i>beta</i> .
<i>compq</i>	CHARACTER*1. Must be 'N', 'V', or 'I'. If <i>compq</i> = 'N', do not modify <i>q</i> . If <i>compq</i> = 'V', multiply the array <i>q</i> on the right by the transpose/conjugate transpose of the orthogonal/unitary transformation that is applied to the left side of $A$ and $B$ to reduce them to Schur form. If <i>compq</i> = 'I', do like <i>compq</i> = 'V', except that <i>q</i> will be initialized to the identity first.
<i>compz</i>	CHARACTER*1. Must be 'N', 'V', or 'I'.

If  $\text{compz} = \text{'N'}$ , do not modify  $\text{z}$ .

If  $\text{compz} = \text{'V'}$ , multiply the array  $\text{z}$  on the right by the orthogonal/unitary transformation that is applied to the right side of  $A$  and  $B$  to reduce them to Schur form.

If  $\text{compz} = \text{'I'}$ , do like  $\text{compz} = \text{'V'}$ , except that  $\text{z}$  will be initialized to the identity first.

$n$  INTEGER. The order of the matrices  $A$ ,  $B$ ,  $Q$ , and  $Z$  ( $n \geq 0$ ).

$ilo, ihi$  INTEGER. It is assumed that  $A$  is already upper triangular in rows and columns  $1:ilo-1$  and  $ihi+1:n$ . Constraint:

If  $n > 0$ , then  $1 \leq ilo \leq ihi \leq n$ ;  
if  $n = 0$ , then  $ilo = 1$  and  $ihi = 0$ .

$a, b, q, z, work$  REAL for `shgeqz`  
DOUBLE PRECISION for `dhgeqz`  
COMPLEX for `chgeqz`  
DOUBLE COMPLEX for `zhgeqz`.

Arrays:

On entry,  $a(1:ilo, *)$  contains the  $n$ -by- $n$  upper Hessenberg matrix  $A$ . Elements below the subdiagonal must be zero.

The second dimension of  $a$  must be at least  $\max(1, n)$ .

On entry,  $b(1:ilo, *)$  contains the  $n$ -by- $n$  upper triangular matrix  $B$ . Elements below the diagonal must be zero.

The second dimension of  $b$  must be at least  $\max(1, n)$ .

The arrays  $q(1:ldq, *)$  and  $z(1:ldz, *)$  accumulate orthogonal/unitary transformations used to reduce  $(A, B)$  to generalized Schur form.

If  $\text{compq} = \text{'N'}$ , then  $q$  is not referenced.

If  $\text{compq} = \text{'V'}$  or  $\text{'I'}$ , then the transpose/conjugate transpose of the orthogonal/unitary transformations which are applied to  $A$  and  $B$  on the left will be applied to the array  $q$  on the right.

The second dimension of  $q$  must be at least  $\max(1, n)$ .

If  $\text{compz} = \text{'N'}$ , then  $\text{z}$  is not referenced.  
 If  $\text{compz} = \text{'V}'$  or  $\text{'I'}$ , then the orthogonal/unitary transformations which are applied to  $A$  and  $B$  on the right will be applied to the array  $\text{z}$  on the right.  
 The second dimension of  $\text{z}$  must be at least  $\max(1, n)$ .

$\text{z}(\text{ldz}, *)$

If  $\text{compq} = \text{'N'}$ , then  $\text{z}$  is not referenced.  
 If  $\text{compq} = \text{'I'}$ , then, on entry,  $\text{z}$  need not be set.  
 If  $\text{compq} = \text{'V'}$ , then  $\text{z}$  must contain the orthogonal/unitary matrix  $Z_1$ .

The second dimension of  $\text{z}$  must be at least  $\max(1, n)$ .

$\text{work}(\text{lwork})$  is a workspace array.

$\text{lda}$  **INTEGER**. The first dimension of  $\text{a}$ ; at least  $\max(1, n)$ .

$\text{ldb}$  **INTEGER**. The first dimension of  $\text{b}$ ; at least  $\max(1, n)$ .

$\text{ldq}$  **INTEGER**. The first dimension of  $\text{q}$ ;

If  $\text{compq} = \text{'N'}$ , then  $\text{ldq} \geq 1$ .

If  $\text{compq} = \text{'I'}$  or  $\text{'V'}$ , then  $\text{ldq} \geq \max(1, n)$ .

$\text{ldz}$  **INTEGER**. The first dimension of  $\text{z}$ ;

If  $\text{compq} = \text{'N'}$ , then  $\text{ldz} \geq 1$ .

If  $\text{compq} = \text{'I'}$  or  $\text{'V'}$ , then  $\text{ldz} \geq \max(1, n)$ .

$\text{lwork}$  **INTEGER**. The dimension of the array  $\text{work}$ .

$\text{lwork} \geq \max(1, n)$ .

$\text{rwork}$  **REAL** for  $\text{chgeqz}$

**DOUBLE PRECISION** for  $\text{zhgeqz}$ .

Workspace array, **DIMENSION** at least  $\max(1, n)$ . Used in complex flavors only.

## Output Parameters

$\text{a}$  If  $\text{job} = \text{'S'}$ , then on exit  $A$  and  $B$  will have been simultaneously reduced to generalized Shur form.

If  $\text{job} = \text{'E'}$ , then on exit  $A$  will be overwritten. For real flavors, the diagonal blocks will be correct, but the off-diagonal portion will be meaningless.

*b*

If *job* = 'S', then on exit *A* and *B* will have been simultaneously reduced to generalized Schur form.  
 If *job* = 'E', then on exit *B* will be overwritten.  
*For real flavors:*  
 2-by-2 blocks in *B* corresponding to 2-by-2 blocks in *A* will be reduced to positive diagonal form. (I.e., if *A*(j+1,j) is non-zero, then *B*(j+1,j)=*B*(j,j+1)=0 and *B*(j,j) and *B*(j+1,j+1) will be positive.)

Elements corresponding to the diagonal blocks of *A* will be correct, but the off-diagonal portion will be meaningless.

*alphar, alphai* REAL for **shgeqz**;  
 DOUBLE PRECISION for **dhgeqz**.  
 Arrays, DIMENSION at least max(1,*n*).  
*alphar*(1:*n*) and *alphai*(1:*n*) will be set, respectively, to real and imaginary parts of the diagonal elements of *A* that would result from reducing *A* and *B* to Schur form and then further reducing them both to triangular form using unitary transformations such that the diagonal of *B* was non-negative real.

Thus, if *A*(j,j) is in a 1-by-1 block (i.e., *A*(j+1,j)=*A*(j,j+1)=0), then *alphar*(j)=*A*(j,j) and *alphai*(j)=0.

Note that the (real or complex) values (*alphar*(j) + i\**alphai*(j))/*beta*(j), j=1,...,*n*, are the generalized eigenvalues of the matrix pencil *A* - *wB*.

*alpha* COMPLEX for **chgeqz**;  
 DOUBLE COMPLEX for **zhgeqz**.  
 Array, DIMENSION at least max(1,*n*).  
 Contains the diagonal elements of *A* when the pair (*A*,*B*) has been reduced to Schur form.  
*alphai*(i)/*beta*(i), i=1,...,*n* are the generalized eigenvalues .

*beta* REAL for **shgeqz**  
 DOUBLE PRECISION for **dhgeqz**  
 COMPLEX for **chgeqz**

**DOUBLE COMPLEX** for `zhgeqz`.

Array, **DIMENSION** at least  $\max(1, n)$ .

*For real flavors:*

`beta(1:n)` will be set to the (real) diagonal elements of  $B$  that would result from reducing  $A$  and  $B$  to Schur form and then further reducing them both to triangular form using unitary transformations such that the diagonal of  $B$  was non-negative real.

Thus, if  $A(j,j)$  is in a 1-by-1 block

(i.e.,  $A(j+1,j)=A(j,j+1)=0$ ), then `beta(j)=B(j,j)`.

Note that `beta(1:n)` will always be non-negative, and no `betai` is necessary.)

*For complex flavors:*

The diagonal elements of  $B$  when the pair  $(A,B)$  has been reduced to Shur form. `alpha(i)/beta(i)`,  $i=1,\dots,n$  are the generalized eigenvalues.  $A$  and  $B$  are normalized so that `beta(1),...,beta(n)` are nonnegative real numbers.

`q, z`

Accumulated orthogonal/unitary transformations used to reduce  $(A,B)$  to generalized Shur form. See `q, z` in *Input Parameters* section.

`work(1)`

If  $info \geq 0$ , on exit, `work(1)` contains the minimum value of `lwork` required for optimum performance. Use this `lwork` for subsequent runs.

`info`

**INTEGER.**

If  $info = 0$ , the execution is successful.

If  $info = -i$ , the  $i$ th parameter had an illegal value.

If  $info = 1,\dots,n$ , the  $QZ$  iteration did not converge.

$(A,B)$  is not in Shur form, but `alphar(i)`, `alphai(i)` (for real flavors), `alpha(i)` (for complex flavors), and `beta(i)`,  $i=info+1,\dots,n$  should be correct.

If  $info = n+1,\dots,2n$ , the shift calculation failed.

$(A,B)$  is not in Shur form, but `alphar(i)`, `alphai(i)` (for real flavors), `alpha(i)` (for complex flavors), and `beta(i)`,  $i=info-n+1,\dots,n$  should be correct.

If  $info > 2n$ , then various “impossible” errors occurred.

## ?tgevc

*Computes some or all of the right  
and/or left generalized eigenvectors of a  
pair of upper triangular matrices.*

---

```
call stgevc ( side, howmny, select, n, a, lda, b, ldb, vl, ldvl, vr,
              ldvr, mm, m, work, info )
call stgevc ( side, howmny, select, n, a, lda, b, ldb, vl, ldvl, vr,
              ldvr, mm, m, work, info )
call stgevc ( side, howmny, select, n, a, lda, b, ldb, vl, ldvl, vr,
              ldvr, mm, m, work, rwork, info )
call stgevc ( side, howmny, select, n, a, lda, b, ldb, vl, ldvl, vr,
              ldvr, mm, m, work, rwork, info )
```

### Discussion

This routine computes some or all of the right and/or left generalized eigenvectors of a pair of real/complex upper triangular matrices  $(A, B)$ . The right generalized eigenvector  $x$  and the left generalized eigenvector  $y$  of  $(A, B)$  corresponding to a generalized eigenvalue  $w$  are defined by:

$$(A - wB) * x = 0 \text{ and } y^H * (A - wB) = 0$$

If an eigenvalue  $w$  is determined by zero diagonal elements of both  $A$  and  $B$ , a unit vector is returned as the corresponding eigenvector.

If all eigenvectors are requested, the routine may either return the matrices  $X$  and/or  $Y$  of right or left eigenvectors of  $(A, B)$ , or the products  $ZX$  and/or  $QY$ , where  $Z$  and  $Q$  are input orthogonal/unitary matrices. If  $(A, B)$  was obtained from the generalized real-Schur/Shur factorization of an original pair of matrices

$$(A_0, B_0) = (QAZ^H, QBZ^H),$$

then  $ZX$  and  $QY$  are the matrices of right or left eigenvectors of  $A$ .

For real flavors,  $A$  must be block upper triangular, with 1-by-1 and 2-by-2 diagonal blocks. Corresponding to each 2-by-2 diagonal block is a complex conjugate pair of eigenvalues and eigenvectors; only one eigenvector of the pair is computed, namely the one corresponding to the eigenvalue with positive imaginary part.

### Input Parameters

<i>side</i>	CHARACTER*1. Must be ' <code>R</code> ', ' <code>L</code> ', or ' <code>B</code> '. If <i>side</i> = ' <code>R</code> ', compute right eigenvectors only. If <i>side</i> = ' <code>L</code> ', compute left eigenvectors only. If <i>side</i> = ' <code>B</code> ', compute both right and left eigenvectors.
<i>howmny</i>	CHARACTER*1. Must be ' <code>A</code> ' , ' <code>B</code> ' , or ' <code>S</code> '. If <i>howmny</i> = ' <code>A</code> ', compute all right and/or left eigenvectors. If <i>howmny</i> = ' <code>B</code> ', compute all right and/or left eigenvectors, and backtransform them using the input matrices supplied in <i>vr</i> and/or <i>vl</i> . If <i>howmny</i> = ' <code>S</code> ', compute selected right and/or left eigenvectors, specified by the logical array <i>select</i> .
<i>select</i>	LOGICAL. Array, DIMENSION at least max (1, <i>n</i> ). If <i>howmny</i> = ' <code>S</code> ', <i>select</i> specifies the eigenvectors to be computed. If <i>howmny</i> = ' <code>A</code> ' or ' <code>B</code> ', <i>select</i> is not referenced. <i>For real flavors:</i> To select the real eigenvector corresponding to a real eigenvalue $\omega_j$ , <i>select</i> ( <i>j</i> ) must be set to .TRUE.; to select the complex eigenvector corresponding to a complex conjugate pair of eigenvalues $\omega_j$ and $\omega_{j+1}$ , either <i>select</i> ( <i>j</i> ) or <i>select</i> ( <i>j</i> +1) must be set to .TRUE.. <i>For complex flavors:</i> To select the eigenvector corresponding to the <i>j</i> -th eigenvalue, <i>select</i> ( <i>j</i> ) must be set to .TRUE..
<i>n</i>	INTEGER. The order of the matrices $A$ and $B$ ( <i>n</i> $\geq$ 0).

*a, b, vl, vr, work* REAL for **stgevc**  
 DOUBLE PRECISION for **dtgevc**  
 COMPLEX for **ctgevc**  
 DOUBLE COMPLEX for **ztgevc**.

Arrays:

*a( lda, \* )* contains the matrix *A*, which is upper quasi-triangular for real flavors, and upper triangular for complex flavors.

The second dimension of *a* must be at least  $\max(1, n)$ .

*b( ldb, \* )* contains the upper triangular matrix *B*.

For real flavors, if *A* has 2-by-2 diagonal block, then the corresponding 2-by-2 block of *B* must be diagonal with positive elements.

For complex flavors, *B* must have real diagonal elements.

The second dimension of *b* must be at least  $\max(1, n)$ .

If *side* = 'L' or 'B' and *howmny* = 'B',  
*vl( ldvl, \* )* must contain an *n*-by-*n* matrix *Q* (usually the orthogonal/unitary matrix *Q* of left Schur vectors returned by **?hgeqz**). The second dimension of *vl* must be at least  $\max(1, mm)$ .

If *side* = 'R' or 'B' and *howmny* = 'B',  
*vr( ldvr, \* )* must contain an *n*-by-*n* matrix *Z* (usually the orthogonal/unitary matrix *Z* of right Schur vectors returned by **?hgeqz**). The second dimension of *vr* must be at least  $\max(1, mm)$ .

*work( \* )* is a workspace array.

DIMENSION at least  $\max(1, 6 * n)$  for real flavors and at least  $\max(1, 2 * n)$  for complex flavors.

*lda* INTEGER. The first dimension of *a*; at least  $\max(1, n)$ .

*ldb* INTEGER. The first dimension of *b*; at least  $\max(1, n)$ .

*ldvl* INTEGER. The first dimension of *vl*;

If *side* = 'L' or 'B', then *ldvl*  $\geq \max(1, n)$ .

If *side* = 'R', then *ldvl*  $\geq 1$ .

<i>ldvr</i>	<b>INTEGER.</b> The first dimension of <i>vr</i> , If <i>side</i> = 'R' or 'B', then <i>ldvr</i> $\geq \max(1, n)$ . If <i>side</i> = 'L', then <i>ldvr</i> $\geq 1$ .
<i>mm</i>	<b>INTEGER.</b> The number of columns in the arrays <i>vl</i> and/or <i>vr</i> ( <i>mm</i> $\geq m$ ).
<i>rwork</i>	<b>REAL</b> for <i>ctgevc</i> <b>DOUBLE PRECISION</b> for <i>ztgevc</i> . Workspace array, <b>DIMENSION</b> at least $\max(1, 2 * n)$ . Used in complex flavors only.

## Output Parameters

<i>vl</i>	On exit, if <i>side</i> = 'L' or 'B', <i>vl</i> contains: if <i>howmny</i> = 'A', the matrix <i>Y</i> of left eigenvectors of <i>(A,B)</i> ; if <i>howmny</i> = 'B', the matrix <i>QY</i> ; if <i>howmny</i> = 'S', the left eigenvectors of <i>(A,B)</i> specified by <i>select</i> , stored consecutively in the columns of <i>vl</i> , in the same order as their eigenvalues. If <i>side</i> = 'R', <i>vl</i> is not referenced. <i>For real flavors:</i> A complex eigenvector corresponding to a complex eigenvalue is stored in two consecutive columns, the first holding the real part, and the second the imaginary part.
<i>vr</i>	On exit, if <i>side</i> = 'R' or 'B', <i>vr</i> contains: if <i>howmny</i> = 'A', the matrix <i>X</i> of right eigenvectors of <i>(A,B)</i> ; if <i>howmny</i> = 'B', the matrix <i>ZX</i> ; if <i>howmny</i> = 'S', the right eigenvectors of <i>(A,B)</i> specified by <i>select</i> , stored consecutively in the columns of <i>vr</i> , in the same order as their eigenvalues. If <i>side</i> = 'L', <i>vr</i> is not referenced. <i>For real flavors:</i> A complex eigenvector corresponding to a complex

eigenvalue is stored in two consecutive columns, the first holding the real part, and the second the imaginary part.

*m*

**INTEGER.** The number of columns in the arrays *vl* and/or *vr* actually used to store the eigenvectors.

If *howmny* = 'A' or 'B', *m* is set to *n*.

*For real flavors:*

Each selected real eigenvector occupies one column and each selected complex eigenvector occupies two columns.

*For complex flavors:*

Each selected eigenvector occupies one column.

*info*

**INTEGER.**

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*th parameter had an illegal value.

*For real flavors:*

If *info* = *i*>0, the 2-by-2 block (*i*:*i*+1) does not have a complex eigenvalue.

## ?tgexc

*Reorders the generalized Schur decomposition of a pair of matrices (A,B) so that one diagonal block of (A,B) moves to another row index.*

---

```
call stgexc ( wantq, wantz, n, a, lda, b, ldb, q, ldq, z, ldz,
              ifst, ilst, work, lwork, info )
call dtgexc ( wantq, wantz, n, a, lda, b, ldb, q, ldq, z, ldz,
              ifst, ilst, work, lwork, info )
call ctgexc ( wantq, wantz, n, a, lda, b, ldb, q, ldq, z, ldz,
              ifst, ilst, info )
call ztgexc ( wantq, wantz, n, a, lda, b, ldb, q, ldq, z, ldz,
              ifst, ilst, info )
```

### Discussion

This routine reorders the generalized real-Schur/Shur decomposition of a real/complex matrix pair ( $A, B$ ) using an orthogonal/unitary equivalence transformation

$$(A, B) = Q (A, B) Z^H,$$

so that the diagonal block of ( $A, B$ ) with row index *ifst* is moved to row *ilst*.

Matrix pair ( $A, B$ ) must be in generalized real-Schur/Shur canonical form (as returned by [?gges](#)), i.e.  $A$  is block upper triangular with 1-by-1 and 2-by-2 diagonal blocks and  $B$  is upper triangular.

Optionally, the matrices  $Q$  and  $Z$  of generalized Schur vectors are updated.

$$\begin{aligned} Q(\text{in}) * A(\text{in}) * Z(\text{in})' &= Q(\text{out}) * A(\text{out}) * Z(\text{out})' \\ Q(\text{in}) * B(\text{in}) * Z(\text{in})' &= Q(\text{out}) * B(\text{out}) * Z(\text{out})'. \end{aligned}$$

### Input Parameters

*wantq, wantz* **LOGICAL.**

If *wantq*=.TRUE., update the left transformation matrix  $Q$ ;

If `wantq = .FALSE.`, do not update  $Q$ ;  
 If `wantz = .TRUE.`, update the right transformation  
 matrix  $Z$ ;  
 If `wantz = .FALSE.`, do not update  $Z$ .

`n` INTEGER. The order of the matrices  $A$  and  $B$  ( $n \geq 0$ ).

`a, b, q, z` REAL for `stgexc`  
 DOUBLE PRECISION for `dtgexc`  
 COMPLEX for `ctgexc`  
 DOUBLE COMPLEX for `ztgexc`.

Arrays:

`a(lda, *)` contains the matrix  $A$ .

The second dimension of `a` must be at least  $\max(1, n)$ .

`b ldb, *)` contains the matrix  $B$ .

The second dimension of `b` must be at least  $\max(1, n)$ .

`q ldq, *)`

If `wantq = .FALSE.`, then `q` is not referenced.

If `wantq = .TRUE.`, then `q` must contain the  
 orthogonal/unitary matrix  $Q$ .

The second dimension of `q` must be at least  $\max(1, n)$ .

`z ldz, *)`

If `wantz = .FALSE.`, then `z` is not referenced.

If `wantz = .TRUE.`, then `z` must contain the  
 orthogonal/unitary matrix  $Z$ .

The second dimension of `z` must be at least  $\max(1, n)$ .

`lda` INTEGER. The first dimension of `a`; at least  $\max(1, n)$ .

`ldb` INTEGER. The first dimension of `b`; at least  $\max(1, n)$ .

`ldq` INTEGER. The first dimension of `q`;

If `wantq = .FALSE.`, then `ldq`  $\geq 1$ .

If `wantq = .TRUE.`, then `ldq`  $\geq \max(1, n)$ .

`ldz` INTEGER. The first dimension of `z`;

If `wantz = .FALSE.`, then `ldz`  $\geq 1$ .

If `wantz = .TRUE.`, then `ldz`  $\geq \max(1, n)$ .

<i>ifst, ilst</i>	<b>INTEGER.</b> Specify the reordering of the diagonal blocks of ( $A, B$ ). The block with row index <i>ifst</i> is moved to row <i>ilst</i> , by a sequence of swapping between adjacent blocks. Constraint: $1 \leq ifst, ilst \leq n$ .
<i>work</i>	<b>REAL</b> for <b>stgexc</b> ; <b>DOUBLE PRECISION</b> for <b>dtgexc</b> . Workspace array, <b>DIMENSION</b> ( <i>lwork</i> ). Used in real flavors only.
<i>lwork</i>	<b>INTEGER.</b> The dimension of <i>work</i> ; must be at least $4n + 16$ .

### Output Parameters

<i>a, b</i>	Overwritten by the updated matrices $A$ and $B$ .
<i>ifst, ilst</i>	Overwritten for real flavors only. If <i>ifst</i> pointed to the second row of a 2 by 2 block on entry, it is changed to point to the first row; <i>ilst</i> always points to the first row of the block in its final position (which may differ from its input value by $\pm 1$ ).
<i>info</i>	<b>INTEGER.</b> If <i>info</i> = 0, the execution is successful. If <i>info</i> = $-i$ , the <i>i</i> th parameter had an illegal value. If <i>info</i> = 1, the transformed matrix pair ( $A, B$ ) would be too far from generalized Schur form; the problem is ill-conditioned. ( $A, B$ ) may have been partially reordered, and <i>ilst</i> points to the first row of the current position of the block being moved.

## ?tgse

*Reorders the generalized Shur decomposition of a pair of matrices (A,B) so that a selected cluster of eigenvalues appears in the leading diagonal blocks of (A,B).*

---

```
call stgse ( ijob, wantq, wantz, select, n, a, lda, b, ldb, alphar,
            alphai, beta, q, ldq, z, ldz, m, pl, pr, dif, work,
            lwork, iwork, liwork, info )
call dtgse ( ijob, wantq, wantz, select, n, a, lda, b, ldb, alphar,
            alphai, beta, q, ldq, z, ldz, m, pl, pr, dif, work,
            lwork, iwork, liwork, info )
call ctgse ( ijob, wantq, wantz, select, n, a, lda, b, ldb, alpha,
            beta, q, ldq, z, ldz, m, pl, pr, dif, work,
            lwork, iwork, liwork, info )
call ztgse ( ijob, wantq, wantz, select, n, a, lda, b, ldb, alpha,
            beta, q, ldq, z, ldz, m, pl, pr, dif, work,
            lwork, iwork, liwork, info )
```

### Discussion

This routine reorders the generalized real-Schur/Shur decomposition of a real/complex matrix pair ( $A, B$ ) (in terms of an orthogonal/unitary equivalence transformation  $Q' * (A, B) * Z$ ), so that a selected cluster of eigenvalues appears in the leading diagonal blocks of the pair ( $A, B$ ).

The leading columns of  $Q$  and  $Z$  form orthonormal/unitary bases of the corresponding left and right eigenspaces (deflating subspaces).

( $A, B$ ) must be in generalized real-Schur/Shur canonical form (as returned by [?gges](#)), that is,  $A$  and  $B$  are both upper triangular.

?tgse also computes the generalized eigenvalues

$\omega_j = (\text{alphar}(j) + \text{alphai}(j)*i)/\text{beta}(j)$  (for real flavors)  
 $\omega_j = \text{alpha}(j)/\text{beta}(j)$  (for complex flavors)

of the reordered matrix pair ( $A, B$ ).

Optionally, the routine computes the estimates of reciprocal condition numbers for eigenvalues and eigenspaces. These are Difu[ $(A_{11}, B_{11}), (A_{22}, B_{22})$ ] and Difl[ $(A_{11}, B_{11}), (A_{22}, B_{22})$ ], that is, the separation(s) between the matrix pairs  $(A_{11}, B_{11})$  and  $(A_{22}, B_{22})$  that correspond to the selected cluster and the eigenvalues outside the cluster, respectively, and norms of "projections" onto left and right eigenspaces with respect to the selected cluster in the (1,1)-block.

### Input Parameters

<i>ijob</i>	<b>INTEGER.</b> Specifies whether condition numbers are required for the cluster of eigenvalues ( <i>pl</i> and <i>pr</i> ) or the deflating subspaces Difu and Difl. If <i>ijob</i> =0, only reorder with respect to <i>select</i> ; If <i>ijob</i> =1, reciprocal of norms of "projections" onto left and right eigenspaces with respect to the selected cluster ( <i>pl</i> and <i>pr</i> ); If <i>ijob</i> =2, compute upper bounds on Difu and Difl, using F-norm-based estimate ( <i>dif</i> (1:2)); If <i>ijob</i> =3, compute estimate of Difu and Difl, using 1-norm-based estimate ( <i>dif</i> (1:2)). This option is about 5 times as expensive as <i>ijob</i> =2; If <i>ijob</i> =4, compute <i>pl</i> , <i>pr</i> and <i>dif</i> (i.e., options 0, 1 and 2 above). This is an economic version to get it all; If <i>ijob</i> =5, compute <i>pl</i> , <i>pr</i> and <i>dif</i> (i.e., options 0, 1 and 3 above).
<i>wantq</i> , <i>wantz</i>	<b>LOGICAL.</b> If <i>wantq</i> =.TRUE., update the left transformation matrix <i>Q</i> ; If <i>wantq</i> =.FALSE., do not update <i>Q</i> ; If <i>wantz</i> =.TRUE., update the right transformation matrix <i>Z</i> ; If <i>wantz</i> =.FALSE., do not update <i>Z</i> .
<i>select</i>	<b>LOGICAL.</b> Array, <b>DIMENSION</b> at least max (1, <i>n</i> ). Specifies the eigenvalues in the selected cluster. To select an eigenvalue $\omega_j$ , <i>select(j)</i> must be .TRUE..

*For real flavors:* to select a complex conjugate pair of eigenvalues  $\omega_j$  and  $\omega_{j+1}$  (corresponding 2 by 2 diagonal block), `select(j)` and/or `select(j+1)` must be set to `.TRUE.`; the complex conjugate  $\omega_j$  and  $\omega_{j+1}$  must be either both included in the cluster or both excluded.

`n` `INTEGER`. The order of the matrices  $A$  and  $B$  ( $n \geq 0$ ).

`a, b, q, z, work` `REAL` for `stgSEN`  
`DOUBLE PRECISION` for `dtgSEN`  
`COMPLEX` for `ctgSEN`  
`DOUBLE COMPLEX` for `ztgSEN`.

Arrays:

`a(lda, *)` contains the matrix  $A$ .

*For real flavors:*  $A$  is upper quasi-triangular, with  $(A, B)$  in generalized real Schur canonical form.

*For complex flavors:*  $A$  is upper triangular, in generalized Schur canonical form.

The second dimension of `a` must be at least  $\max(1, n)$ .

`b(lDb, *)` contains the matrix  $B$ .

*For real flavors:*  $B$  is upper triangular, with  $(A, B)$  in generalized real Schur canonical form.

*For complex flavors:*  $B$  is upper triangular, in generalized Schur canonical form.

The second dimension of `b` must be at least  $\max(1, n)$ .

`q(ldq, *)`

If `wantq = .TRUE.`, then `q` is an  $n$ -by- $n$  matrix;

If `wantq = .FALSE.`, then `q` is not referenced.

The second dimension of `q` must be at least  $\max(1, n)$ .

`z(ldz, *)`

If `wantz = .TRUE.`, then `z` is an  $n$ -by- $n$  matrix;

If `wantz = .FALSE.`, then `z` is not referenced.

The second dimension of `z` must be at least  $\max(1, n)$ .

`work(lwork)` is a workspace array. If `iJob=0`, `work` is not referenced.

`lda` `INTEGER`. The first dimension of `a`; at least  $\max(1, n)$ .

`ldb` `INTEGER`. The first dimension of `b`; at least  $\max(1, n)$ .

<i>ldq</i>	<b>INTEGER.</b> The first dimension of <i>q</i> ; <i>ldq</i> $\geq 1$ . If <i>wantq</i> = .TRUE., then <i>ldq</i> $\geq \max(1, n)$ .
<i>ldz</i>	<b>INTEGER.</b> The first dimension of <i>z</i> ; <i>ldz</i> $\geq 1$ . If <i>wantz</i> = .TRUE., then <i>ldz</i> $\geq \max(1, n)$ .
<i>lwork</i>	<b>INTEGER.</b> The dimension of the array <i>work</i> . <i>For real flavors:</i> If <i>i job</i> = 1, 2, or 4, <i>lwork</i> $\geq \max(4n+16, 2m(n-m))$ . If <i>i job</i> = 3 or 5, <i>lwork</i> $\geq \max(4n+16, 4m(n-m))$ . <i>For complex flavors:</i> If <i>i job</i> = 1, 2, or 4, <i>lwork</i> $\geq \max(1, 2m(n-m))$ . If <i>i job</i> = 3 or 5, <i>lwork</i> $\geq \max(1, 4m(n-m))$ .
<i>iwork</i>	<b>INTEGER.</b> Workspace array, <b>DIMENSION</b> ( <i>liwork</i> ). If <i>i job</i> = 0, <i>iwork</i> is not referenced.
<i>liwork</i>	<b>INTEGER.</b> The dimension of the array <i>iwork</i> . <i>For real flavors:</i> If <i>i job</i> = 1, 2, or 4, <i>liwork</i> $\geq n+6$ . If <i>i job</i> = 3 or 5, <i>liwork</i> $\geq \max(n+6, 2m(n-m))$ . <i>For complex flavors:</i> If <i>i job</i> = 1, 2, or 4, <i>liwork</i> $\geq n+2$ . If <i>i job</i> = 3 or 5, <i>liwork</i> $\geq \max(n+2, 2m(n-m))$ .

## Output Parameters

<i>a</i> , <i>b</i>	Overwritten by the reordered matrices <i>A</i> and <i>B</i> , respectively.
<i>alphar</i> , <i>alphai</i>	<b>REAL</b> for <b>stgSEN</b> ; <b>DOUBLE PRECISION</b> for <b>dtgSEN</b> . Arrays, <b>DIMENSION</b> at least $\max(1, n)$ . Contain values that form generalized eigenvalues in real flavors. See <i>beta</i> .
<i>alpha</i>	<b>COMPLEX</b> for <b>ctgSEN</b> ; <b>DOUBLE COMPLEX</b> for <b>ztgSEN</b> . Array, <b>DIMENSION</b> at least $\max(1, n)$ . Contain values that form generalized eigenvalues in complex flavors. See <i>beta</i> .
<i>beta</i>	<b>REAL</b> for <b>stgSEN</b> <b>DOUBLE PRECISION</b> for <b>dtgSEN</b> <b>COMPLEX</b> for <b>ctgSEN</b>

**DOUBLE COMPLEX** for `ztgsen`.

Array, **DIMENSION** at least  $\max(1, n)$ .

*For real flavors:*

On exit,  $(\text{alphar}(j) + \text{alphaai}(j)*i)/\text{beta}(j)$ ,  $j=1,\dots,n$ , will be the generalized eigenvalues.

$\text{alphar}(j) + \text{alphaai}(j)*i$  and  $\text{beta}(j)$ ,  $j=1,\dots,n$  are the diagonals of the complex Schur form  $(S, T)$  that would result if the 2-by-2 diagonal blocks of the real generalized Schur form of  $(A, B)$  were further reduced to triangular form using complex unitary transformations. If  $\text{alphaai}(j)$  is zero, then the  $j$ -th eigenvalue is real; if positive, then the  $j$ -th and  $(j+1)$ -st eigenvalues are a complex conjugate pair, with  $\text{alphaai}(j+1)$  negative.

*For complex flavors:*

The diagonal elements of  $A$  and  $B$ , respectively, when the pair  $(A, B)$  has been reduced to generalized Schur form.  $\text{alpha}(i)/\text{beta}(i)$ ,  $i=1,\dots,n$  are the generalized eigenvalues.

`q`

If `wantq = .TRUE.`, then, on exit,  $Q$  has been postmultiplied by the left orthogonal transformation matrix which reorder  $(A, B)$ . The leading  $m$  columns of  $Q$  form orthonormal bases for the specified pair of left eigenspaces (deflating subspaces).

`z`

If `wantz = .TRUE.`, then, on exit,  $Z$  has been postmultiplied by the left orthogonal transformation matrix which reorder  $(A, B)$ . The leading  $m$  columns of  $Z$  form orthonormal bases for the specified pair of left eigenspaces (deflating subspaces).

`m`

**INTEGER**. The dimension of the specified pair of left and right eigen-spaces (deflating subspaces);  $0 \leq m \leq n$ .

`pl, pr`

**REAL** for single precision flavors;

**DOUBLE PRECISION** for double precision flavors.

If  $i_{\text{job}} = 1, 4$ , or  $5$ ,  $pl$  and  $pr$  are lower bounds on the reciprocal of the norm of "projections" onto left and

right eigenspaces with respect to the selected cluster.  
 $0 < \text{pl}, \text{pr} \leq 1$ . If  $m = 0$  or  $m = n$ ,  $\text{pl} = \text{pr} = 1$ .  
If  $i_{\text{job}} = 0, 2$  or  $3$ ,  $\text{pl}$  and  $\text{pr}$  are not referenced

*dif*                    **REAL** for single precision flavors;  
**DOUBLE PRECISION** for double precision flavors.  
Array, **DIMENSION** (2).  
If  $i_{\text{job}} \geq 2$ ,  $\text{dif}(1:2)$  store the estimates of Difu and Difl.  
If  $i_{\text{job}} = 2$  or  $4$ ,  $\text{dif}(1:2)$  are F-norm-based upper bounds on Difu and Difl.  
If  $i_{\text{job}} = 3$  or  $5$ ,  $\text{dif}(1:2)$  are 1-norm-based estimates of Difu and Difl. If  $m = 0$  or  $n$ ,  
 $\text{dif}(1:2) = \text{F-norm}([A, B])$ .  
If  $i_{\text{job}} = 0$  or  $1$ ,  $\text{dif}$  is not referenced.

*work(1)*            If  $i_{\text{job}}$  is not  $0$  and  $\text{info} = 0$ , on exit,  $\text{work}(1)$  contains the minimum value of  $l_{\text{work}}$  required for optimum performance. Use this  $l_{\text{work}}$  for subsequent runs.

*iwork(1)*            If  $i_{\text{job}}$  is not  $0$  and  $\text{info} = 0$ , on exit,  $i_{\text{work}}(1)$  contains the minimum value of  $l_{\text{iwork}}$  required for optimum performance. Use this  $l_{\text{iwork}}$  for subsequent runs.

*info*                **INTEGER**.  
If  $\text{info} = 0$ , the execution is successful.  
If  $\text{info} = -i$ , the  $i$ th parameter had an illegal value.  
If  $\text{info} = 1$ , Reordering of  $(A, B)$  failed because the transformed matrix pair  $(A, B)$  would be too far from generalized Schur form; the problem is very ill-conditioned.  $(A, B)$  may have been partially reordered. If requested, 0 is returned in  $\text{dif}(*), \text{pl}$  and  $\text{pr}$ .

## ?tgssyl

*Solves the generalized Sylvester equation.*

```
call stgssyl ( trans, ijob, m, n, a, lda, b, ldb, c, ldc, d, ldd, e,
               lde, f, ldf, scale, dif, work, lwork, iwork, info )
call dtgssyl ( trans, ijob, m, n, a, lda, b, ldb, c, ldc, d, ldd, e,
               lde, f, ldf, scale, dif, work, lwork, iwork, info )
call ctgssyl ( trans, ijob, m, n, a, lda, b, ldb, c, ldc, d, ldd, e,
               lde, f, ldf, scale, dif, work, lwork, iwork, info )
call ztgssyl ( trans, ijob, m, n, a, lda, b, ldb, c, ldc, d, ldd, e,
               lde, f, ldf, scale, dif, work, lwork, iwork, info )
```

### Discussion

This routine solves the generalized Sylvester equation:

$$A R - L B = \text{scale} * C$$

$$D R - L E = \text{scale} * F$$

where  $R$  and  $L$  are unknown  $m$ -by- $n$  matrices,  $(A, D)$ ,  $(B, E)$  and  $(C, F)$  are given matrix pairs of size  $m$ -by- $m$ ,  $n$ -by- $n$  and  $m$ -by- $n$ , respectively, with real/complex entries.  $(A, D)$  and  $(B, E)$  must be in generalized real-Schur/Shur canonical form, that is,  $A, B$  are upper quasi-triangular/triangular and  $D, E$  are upper triangular.

The solution  $(R, L)$  overwrites  $(C, F)$ . The factor  $\text{scale}$ ,  $0 \leq \text{scale} \leq 1$ , is an output scaling factor chosen to avoid overflow.

In matrix notation the above equation is equivalent to the following:  
solve  $Zx = \text{scale} * b$ , where  $Z$  is defined as

$$Z = \begin{pmatrix} \text{kron}(I_n, A) & -\text{kron}(B', I_m) \\ \text{kron}(I_n, D) & -\text{kron}(E', I_m) \end{pmatrix}$$

Here  $I_k$  is the identity matrix of size  $k$  and  $X'$  is the transpose/conjugate-transpose of  $X$ .  $\text{kron}(X, Y)$  is the Kronecker product between the matrices  $X$  and  $Y$ .

If  $\text{trans} = \text{'T'}$  (for real flavors), or  $\text{trans} = \text{'C'}$  (for complex flavors), the routine `?tgssyl` solves the transposed/conjugate-transposed system  $Z'y = \text{scale} * b$ , which is equivalent to solve for  $R$  and  $L$  in

$$A'R + D'L = \text{scale} * C$$

$$R B' + L E' = \text{scale} * (-F)$$

This case ( $\text{trans} = \text{'T'}$  for `stgssyl/dtgssyl` or  $\text{trans} = \text{'C'}$  for `ctgssyl/ztgssyl`) is used to compute an one-norm-based estimate of  $\text{Dif}[(A,D), (B,E)]$ , the separation between the matrix pairs  $(A,D)$  and  $(B,E)$ , using `slacon/clacon`.

If  $i\text{job} \geq 1$ , `?tgssyl` computes a Frobenius norm-based estimate of  $\text{Dif}[(A,D), (B,E)]$ . That is, the reciprocal of a lower bound on the reciprocal of the smallest singular value of  $Z$ . This is a level 3 BLAS algorithm.

## Input Parameters

<code>trans</code>	<code>CHARACTER*1</code> . Must be ' <code>N</code> ', ' <code>T</code> ', or ' <code>C</code> '. If $\text{trans} = \text{'N'}$ , solve the generalized Sylvester equation. If $\text{trans} = \text{'T'}$ , solve the 'transposed' system (for real flavors only). If $\text{trans} = \text{'C'}$ , solve the 'conjugate transposed' system (for complex flavors only).
<code>ijob</code>	<code>INTEGER</code> . Specifies what kind of functionality to be performed: If $i\text{job} = 0$ , solve the generalized Sylvester equation only ; If $i\text{job} = 1$ , perform the functionality of $i\text{job} = 0$ and $i\text{job} = 3$ ; If $i\text{job} = 2$ , perform the functionality of $i\text{job} = 0$ and $i\text{job} = 4$ ; If $i\text{job} = 3$ , only an estimate of $\text{Dif}[(A,D), (B,E)]$ is computed (look ahead strategy is used);

If  $i_{job} = 4$ , only an estimate of  $\text{Dif}[(A,D), (B,E)]$  is computed ([?gecon](#) on sub-systems is used).

If  $trans = 'T'$  or ' $C$ ',  $i_{job}$  is not referenced.

$m$

**INTEGER.**

The order of the matrices  $A$  and  $D$ , and the row dimension of the matrices  $C, F, R$  and  $L$ .

$n$

**INTEGER.**

The order of the matrices  $B$  and  $E$ , and the column dimension of the matrices  $C, F, R$  and  $L$ .

$a, b, c, d, e, f, work$  **REAL** for **stgsyl**

**DOUBLE PRECISION** for **dtgsyl**

**COMPLEX** for **ctgsyl**

**DOUBLE COMPLEX** for **ztgsyl**.

Arrays:

$a(1da, *)$  contains the upper quasi-triangular (for real flavors) or upper triangular (for complex flavors) matrix  $A$ .

The second dimension of  $a$  must be at least  $\max(1, m)$ .

$b(1db, *)$  contains the upper quasi-triangular (for real flavors) or upper triangular (for complex flavors) matrix  $B$ .

The second dimension of  $b$  must be at least  $\max(1, n)$ .

$c(1dc, *)$  contains the right-hand-side of the first matrix equation in the generalized Sylvester equation (as defined by  $trans$ )

The second dimension of  $c$  must be at least  $\max(1, n)$ .

$d(1dd, *)$  contains the upper triangular matrix  $D$ .

The second dimension of  $d$  must be at least  $\max(1, m)$ .

$e(1de, *)$  contains the upper triangular matrix  $E$ .

The second dimension of  $e$  must be at least  $\max(1, n)$ .

$f(1df, *)$  contains the right-hand-side of the second matrix equation in the generalized Sylvester equation (as defined by  $trans$ )

The second dimension of  $f$  must be at least  $\max(1, n)$ .

*work(lwork)* is a workspace array. If *ijob*=0, *work* is not referenced.

<i>lda</i>	INTEGER. The first dimension of <i>a</i> ; at least max(1, <i>m</i> ).
<i>ldb</i>	INTEGER. The first dimension of <i>b</i> ; at least max(1, <i>n</i> ).
<i>ldc</i>	INTEGER. The first dimension of <i>c</i> ; at least max(1, <i>m</i> ).
<i>lld</i>	INTEGER. The first dimension of <i>d</i> ; at least max(1, <i>m</i> ).
<i>lde</i>	INTEGER. The first dimension of <i>e</i> ; at least max(1, <i>n</i> ).
<i>ldf</i>	INTEGER. The first dimension of <i>f</i> ; at least max(1, <i>m</i> ).
<i>lwork</i>	INTEGER. The dimension of the array <i>work</i> . <i>lwork</i> $\geq 1$ . If <i>ijob</i> =1 or 2 and <i>trans</i> = 'N', <i>lwork</i> $\geq 2mn$ .
<i>iwork</i>	INTEGER. Workspace array, DIMENSION at least ( <i>m+n+6</i> ) for real flavors, and at least ( <i>m+n+2</i> ) for complex flavors. If <i>ijob</i> =0, <i>iwork</i> is not referenced.

## Output Parameters

<i>c</i>	If <i>ijob</i> =0, 1, or 2, overwritten by the solution <i>R</i> . If <i>ijob</i> =3 or 4 and <i>trans</i> = 'N', <i>c</i> holds <i>R</i> , the solution achieved during the computation of the Dif-estimate.
<i>f</i>	If <i>ijob</i> =0, 1, or 2, overwritten by the solution <i>L</i> . If <i>ijob</i> =3 or 4 and <i>trans</i> = 'N', <i>f</i> holds <i>L</i> , the solution achieved during the computation of the Dif-estimate.
<i>dif</i>	REAL for single-precision flavors DOUBLE PRECISION for double-precision flavors. On exit, <i>dif</i> is the reciprocal of a lower bound of the reciprocal of the Dif-function, i.e. <i>dif</i> is an upper bound of Dif[( <i>A,D</i> ), ( <i>B,E</i> )] = sigma_min( <i>Z</i> ), where <i>Z</i> as in (2). If <i>ijob</i> =0, or <i>trans</i> = 'T' (for real flavors), or <i>trans</i> = 'C' (for complex flavors), <i>dif</i> is not touched.

<i>scale</i>	<i>REAL</i> for single-precision flavors <i>DOUBLE PRECISION</i> for double-precision flavors. On exit, <i>scale</i> is the scaling factor in the generalized Sylvester equation. If $0 < \text{scale} < 1$ , <i>c</i> and <i>f</i> hold the solutions <i>R</i> and <i>L</i> , respectively, to a slightly perturbed system but the input matrices <i>A</i> , <i>B</i> , <i>D</i> and <i>E</i> have not been changed. If <i>scale</i> = 0, <i>c</i> and <i>f</i> hold the solutions <i>R</i> and <i>L</i> , respectively, to the homogeneous system with <i>C</i> = <i>F</i> = 0. Normally, <i>scale</i> = 1.
<i>work(1)</i>	If <i>ijob</i> is not 0 and <i>info</i> = 0, on exit, <i>work(1)</i> contains the minimum value of <i>lwork</i> required for optimum performance. Use this <i>lwork</i> for subsequent runs.
<i>info</i>	<i>INTEGER</i> . If <i>info</i> = 0, the execution is successful. If <i>info</i> = <i>-i</i> , the <i>i</i> th parameter had an illegal value. If <i>info</i> > 0, ( <i>A</i> , <i>D</i> ) and ( <i>B</i> , <i>E</i> ) have common or close eigenvalues.

---

## ?tgsna

*Estimates reciprocal condition numbers  
for specified eigenvalues and/or  
eigenvectors of a pair of matrices in  
generalized real Schur canonical form.*

---

```
call stgsna ( job, howmny, select, n, a, lda, b, ldb, vl, ldvl, vr,
              ldvr, s, dif, mm, m, work, lwork, iwork, info )
call dtgsna ( job, howmny, select, n, a, lda, b, ldb, vl, ldvl, vr,
              ldvr, s, dif, mm, m, work, lwork, iwork, info )
call ctgsna ( job, howmny, select, n, a, lda, b, ldb, vl, ldvl, vr,
              ldvr, s, dif, mm, m, work, lwork, iwork, info )
call ztgsna ( job, howmny, select, n, a, lda, b, ldb, vl, ldvl, vr,
              ldvr, s, dif, mm, m, work, lwork, iwork, info )
```

### Discussion

The real flavors `stgsna/dtgsna` of this routine estimate reciprocal condition numbers for specified eigenvalues and/or eigenvectors of a matrix pair  $(A, B)$  in generalized real Schur canonical form (or of any matrix pair  $(Q A Z^T, Q B Z^T)$  with orthogonal matrices  $Q$  and  $Z$ .

$(A, B)$  must be in generalized real Schur form (as returned by `sgges/dgges`), that is,  $A$  is block upper triangular with 1-by-1 and 2-by-2 diagonal blocks.  $B$  is upper triangular.

The complex flavors `ctgsna/ztgsna` estimate reciprocal condition numbers for specified eigenvalues and/or eigenvectors of a matrix pair  $(A, B)$ .  $(A, B)$  must be in generalized Schur canonical form , that is,  $A$  and  $B$  are both upper triangular.

### Input Parameters

<code>job</code>	<code>CHARACTER*1</code> . Specifies whether condition numbers are required for eigenvalues or eigenvectors . Must be ' <code>E</code> ' or ' <code>V</code> ' or ' <code>B</code> '. If <code>job</code> = ' <code>E</code> ', for eigenvalues only (compute <code>s</code> ).
------------------	---

If `job = 'V'`, for eigenvectors only (compute `dif`).  
 If `job = 'B'`, for both eigenvalues and eigenvectors  
 (compute both `s` and `dif`).

`howmny` CHARACTER\*1. Must be '`A`' or '`S`'.  
 If `howmny = 'A'`, compute condition numbers for all eigenpairs.  
 If `howmny = 'S'`, compute condition numbers for selected eigenpairs specified by the logical array `select`.

`select` LOGICAL.  
 Array, DIMENSION at least max (1, `n`).  
 If `howmny = 'S'`, `select` specifies the eigenpairs for which condition numbers are required.

If `howmny = 'A'`, `select` is not referenced.

*For real flavors:*

To select condition numbers for the eigenpair corresponding to a real eigenvalue  $\omega_j$ , `select(j)` must be set to `.TRUE.`; to select condition numbers corresponding to a complex conjugate pair of eigenvalues  $\omega_j$  and  $\omega_{j+1}$ , either `select(j)` or `select(j+1)` must be set to `.TRUE.`.

*For complex flavors:*

To select condition numbers for the corresponding  $j$ -th eigenvalue and/or eigenvector, `select(j)` must be set to `.TRUE..`

`n` INTEGER. The order of the square matrix pair ( $A, B$ ) ( $n \geq 0$ ).

`a, b, vl, vr, work` REAL for `stgsna`  
 DOUBLE PRECISION for `dtgsna`  
 COMPLEX for `ctgsna`  
 DOUBLE COMPLEX for `ztgsna`.

Arrays:

`a( lda, * )` contains the upper quasi-triangular (for real flavors) or upper triangular (for complex flavors) matrix  $A$  in the pair ( $A, B$ ).

The second dimension of `a` must be at least  $\max(1, n)$ .

$b(\ ldb, *)$  contains the upper triangular matrix  $B$  in the pair  $(A, B)$ .

The second dimension of  $b$  must be at least  $\max(1, n)$ .

If  $job = 'E'$  or ' $B$ ',

$vl(lv1, *)$  must contain left eigenvectors of  $(A, B)$ , corresponding to the eigenpairs specified by  $howmny$  and  $select$ . The eigenvectors must be stored in consecutive columns of  $vl$ , as returned by  $?tgevc$ .

If  $job = 'V'$ ,  $vl$  is not referenced.

The second dimension of  $vl$  must be at least  $\max(1, m)$ .

If  $job = 'E'$  or ' $B$ ',

$vr(lv1, *)$  must contain right eigenvectors of  $(A, B)$ , corresponding to the eigenpairs specified by  $howmny$  and  $select$ . The eigenvectors must be stored in consecutive columns of  $vr$ , as returned by  $?tgevc$ .

If  $job = 'V'$ ,  $vr$  is not referenced.

The second dimension of  $vr$  must be at least  $\max(1, m)$ .

$work(lwork)$  is a workspace array. If  $job = 'E'$ ,  $work$  is not referenced.

$lda$  **INTEGER.** The first dimension of  $a$ ; at least  $\max(1, n)$ .

$ldb$  **INTEGER.** The first dimension of  $b$ ; at least  $\max(1, n)$ .

$lv1$  **INTEGER.** The first dimension of  $vl$ ;  $lv1 \geq 1$ .

If  $job = 'E'$  or ' $B$ ', then  $lv1 \geq \max(1, n)$ .

$lv1$  **INTEGER.** The first dimension of  $vr$ ;  $lv1 \geq 1$ .

If  $job = 'E'$  or ' $B$ ', then  $lv1 \geq \max(1, n)$ .

$mm$  **INTEGER.** The number of elements in the arrays  $s$  and  $dif$  ( $mm \geq m$ ).

$lwork$  **INTEGER.** The dimension of the array  $work$ .

*For real flavors:*

$lwork \geq n$ .

If  $job = 'V'$  or ' $B$ ',  $lwork \geq 2n(n+2)+16$ .

*For complex flavors:*

$lwork \geq 1$ .

If  $job = 'V'$  or ' $B$ ',  $lwork \geq 2n^2$ .

*iwork*      **INTEGER.** Workspace array, **DIMENSION** at least (*n*+6) for real flavors, and at least (*n*+2) for complex flavors.  
If *ijob*='E', *iwork* is not referenced.

## Output Parameters

*s*      **REAL** for single-precision flavors  
**DOUBLE PRECISION** for double-precision flavors.  
Array, **DIMENSION** (*mm*).  
If *job*='E' or 'B', contains the reciprocal condition numbers of the selected eigenvalues, stored in consecutive elements of the array.  
If *job*='V', *s* is not referenced.  
*For real flavors:*  
For a complex conjugate pair of eigenvalues two consecutive elements of *s* are set to the same value.  
Thus, *s*(j), *dif*(j), and the j-th columns of *vl* and *vr* all correspond to the same eigenpair (but not in general the j-th eigenpair, unless all eigenpairs are selected).  
  
*dif*      **REAL** for single-precision flavors  
**DOUBLE PRECISION** for double-precision flavors.  
Array, **DIMENSION** (*mm*).  
If *job*='V' or 'B', contains the estimated reciprocal condition numbers of the selected eigenvectors, stored in consecutive elements of the array. If the eigenvalues cannot be reordered to compute *dif*(j), *dif*(j) is set to 0; this can only occur when the true value would be very small anyway.  
If *job*='E', *dif* is not referenced.  
*For real flavors:*  
For a complex eigenvector, two consecutive elements of *dif* are set to the same value.  
*For complex flavors:*  
For each eigenvalue/vector specified by *select*, *dif* stores a Frobenius norm-based estimate of Difl.

<i>m</i>	<b>INTEGER.</b> The number of elements in the arrays <i>s</i> and <i>dif</i> used to store the specified condition numbers; for each selected eigenvalue one element is used. If <i>howmny</i> = 'A', <i>m</i> is set to <i>n</i> .
<i>work(1)</i>	<b>work(1)</b> If <i>job</i> is not 'E' and <i>info</i> = 0, on exit, <i>work(1)</i> contains the minimum value of <i>lwork</i> required for optimum performance. Use this <i>lwork</i> for subsequent runs.
<i>info</i>	<b>INTEGER.</b> If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> th parameter had an illegal value.

## Generalized Singular Value Decomposition

This section describes LAPACK computational routines used for finding the generalized singular value decomposition (GSVD) of two matrices  $A$  and  $B$  as

$$U^H A Q = D_1 * (0 \ R), \\ V^H B Q = D_2 * (0 \ R),$$

where  $U$ ,  $V$ , and  $Q$  are orthogonal/unitary matrices,  $R$  is a nonsingular upper triangular matrix, and  $D_1$ ,  $D_2$  are “diagonal” matrices of the structure detailed in the routines description section.

**Table 5-7 Computational Routines for Generalized Singular Value Decomposition**

Routine name	Operation performed
<a href="#">?ggsvp</a>	Computes the preprocessing decomposition for the generalized SVD
<a href="#">?tgsja</a>	Computes the generalized SVD of two upper triangular or trapezoidal matrices

You can use routines listed in the above table as well as the driver routine [?ggsvd](#) to find the GSVD of a pair of general rectangular matrices.

---

## ?ggsvp

*Computes the preprocessing decomposition for the generalized SVD.*

---

```
call sggsvp ( jobu, jobv, jobq, m, p, n, a, lda, b, ldb, tola, tolb,
              k, l, u, ldu, v, ldv, q, ldq, iwork, tau, work, info )
call dggsvp ( jobu, jobv, jobq, m, p, n, a, lda, b, ldb, tola, tolb,
              k, l, u, ldu, v, ldv, q, ldq, iwork, tau, work, info )
call cggsvp ( jobu, jobv, jobq, m, p, n, a, lda, b, ldb, tola, tolb,
              k, l, u, ldu, v, ldv, q, ldq, iwork, rwork, tau, work, info )
call zggsvp ( jobu, jobv, jobq, m, p, n, a, lda, b, ldb, tola, tolb,
              k, l, u, ldu, v, ldv, q, ldq, iwork, rwork, tau, work, info )
```

### Discussion

This routine computes orthogonal matrices  $U$ ,  $V$  and  $Q$  such that

$$U^H A Q = \begin{pmatrix} & n-k-l & k & l \\ & 0 & A_{12} & A_{13} \\ & 0 & 0 & A_{23} \\ m-k-l & 0 & 0 & 0 \end{pmatrix}, \quad \text{if } m-k-l \geq 0$$

$$= \begin{pmatrix} & n-k-l & k & l \\ & 0 & A_{12} & A_{13} \\ & 0 & 0 & A_{23} \\ m-k & 0 & 0 & 0 \end{pmatrix}, \quad \text{if } m-k-l < 0$$

$$V^H B Q = \begin{pmatrix} & n-k-l & k & l \\ & 0 & 0 & B_{13} \\ p-l & 0 & 0 & 0 \end{pmatrix}$$

where the  $k$ -by- $k$  matrix  $A_{12}$  and  $l$ -by- $l$  matrix  $B_{13}$  are nonsingular upper triangular;  $A_{23}$  is  $l$ -by- $l$  upper triangular if  $m-k-l \geq 0$ , otherwise  $A_{23}$  is  $(m-k)$ -by- $l$  upper trapezoidal. The sum  $k+l$  is equal to the effective numerical rank of the  $(m+p)$ -by- $n$  matrix  $(A^H, B^{H^H})^H$ .

This decomposition is the preprocessing step for computing the Generalized Singular Value Decomposition (GSVD), see subroutine [?ggsvd](#).

### Input Parameters

<i>jobu</i>	CHARACTER*1. Must be ' <b>U</b> ' or ' <b>N</b> '. If <i>jobu</i> = ' <b>U</b> ', orthogonal/unitary matrix $U$ is computed. If <i>jobu</i> = ' <b>N</b> ', $U$ is not computed.
<i>jobv</i>	CHARACTER*1. Must be ' <b>V</b> ' or ' <b>N</b> '. If <i>jobv</i> = ' <b>V</b> ', orthogonal/unitary matrix $V$ is computed. If <i>jobv</i> = ' <b>N</b> ', $V$ is not computed.
<i>jobq</i>	CHARACTER*1. Must be ' <b>Q</b> ' or ' <b>N</b> '. If <i>jobq</i> = ' <b>Q</b> ', orthogonal/unitary matrix $Q$ is computed. If <i>jobq</i> = ' <b>N</b> ', $Q$ is not computed.
<i>m</i>	INTEGER. The number of rows of the matrix $A$ ( $m \geq 0$ ).
<i>p</i>	INTEGER. The number of rows of the matrix $B$ ( $p \geq 0$ ).
<i>n</i>	INTEGER. The number of columns of the matrices $A$ and $B$ ( $n \geq 0$ ).
<i>a, b, tau, work</i>	REAL for <b>sggsvp</b> DOUBLE PRECISION for <b>dggsvp</b> COMPLEX for <b>cggsvp</b> DOUBLE COMPLEX for <b>zggsvp</b> . Arrays: <i>a( lda, * )</i> contains the $m$ -by- $n$ matrix $A$ . The second dimension of <i>a</i> must be at least $\max(1, n)$ . <i>b( ldb, * )</i> contains the $p$ -by- $n$ matrix $B$ . The second dimension of <i>b</i> must be at least $\max(1, n)$ . <i>tau( * )</i> is a workspace array. The dimension of <i>tau</i> must be at least $\max(1, n)$ . <i>work( * )</i> is a workspace array. The dimension of <i>work</i> must be at least $\max(1, 3n, m, p)$ .

<i>lda</i>	<code>INTEGER</code> . The first dimension of <i>a</i> ; at least $\max(1, m)$ .
<i>ldb</i>	<code>INTEGER</code> . The first dimension of <i>b</i> ; at least $\max(1, p)$ .
<i>tola, tolb</i>	<code>REAL</code> for single-precision flavors <code>DOUBLE PRECISION</code> for double-precision flavors. <i>tola</i> and <i>tolb</i> are the thresholds to determine the effective numerical rank of matrix <i>B</i> and a subblock of <i>A</i> . Generally, they are set to $\text{tola} = \max(m, n) * \ A\  * \text{MACHEPS},$ $\text{tolb} = \max(p, n) * \ B\  * \text{MACHEPS}.$ The size of <i>tola</i> and <i>tolb</i> may affect the size of backward errors of the decomposition.
<i>ldu</i>	<code>INTEGER</code> . The first dimension of the output array <i>u</i> . $\text{ldu} \geq \max(1, m)$ if $\text{jobu} = 'U'$ ; $\text{ldu} \geq 1$ otherwise.
<i>ldv</i>	<code>INTEGER</code> . The first dimension of the output array <i>v</i> . $\text{ldv} \geq \max(1, p)$ if $\text{jobv} = 'V'$ ; $\text{ldv} \geq 1$ otherwise.
<i>ldq</i>	<code>INTEGER</code> . The first dimension of the output array <i>q</i> . $\text{ldq} \geq \max(1, n)$ if $\text{jobq} = 'Q'$ ; $\text{ldq} \geq 1$ otherwise.
<i>iwork</i>	<code>INTEGER</code> . Workspace array, <code>DIMENSION</code> at least $\max(1, n)$ .
<i>rwork</i>	<code>REAL</code> for <i>cggsvp</i> <code>DOUBLE PRECISION</code> for <i>zggsvp</i> . Workspace array, <code>DIMENSION</code> at least $\max(1, 2n)$ . Used in complex flavors only.

## Output Parameters

<i>a</i>	Overwritten by the triangular (or trapezoidal) matrix described in the <i>Discussion</i> section.
<i>b</i>	Overwritten by the triangular matrix described in the <i>Discussion</i> section.
<i>k, l</i>	<code>INTEGER</code> . On exit, <i>k</i> and <i>l</i> specify the dimension of subblocks. The sum <i>k + l</i> is equal to effective numerical rank of $(A^H, B^{H,H})$ .

*u, v, q*      REAL for `srgsvp`  
                   DOUBLE PRECISION for `drgsvp`  
                   COMPLEX for `crgsvp`  
                   DOUBLE COMPLEX for `zrgsvp`.

Arrays:

If  $\text{jobu} = \text{'U'}$ ,  $u(1:du, *)$  contains the orthogonal/unitary matrix  $U$ .

The second dimension of  $u$  must be at least  $\max(1, m)$ .

If  $\text{jobu} = \text{'N'}$ ,  $u$  is not referenced.

If  $\text{jobv} = \text{'V'}$ ,  $v(1:dv, *)$  contains the orthogonal/unitary matrix  $V$ .

The second dimension of  $v$  must be at least  $\max(1, m)$ .

If  $\text{jobv} = \text{'N'}$ ,  $v$  is not referenced.

If  $\text{jobq} = \text{'Q'}$ ,  $q(1:dq, *)$  contains the orthogonal/unitary matrix  $Q$ .

The second dimension of  $q$  must be at least  $\max(1, n)$ .

If  $\text{jobq} = \text{'N'}$ ,  $q$  is not referenced.

*info*      INTEGER.

If  $\text{info} = 0$ , the execution is successful.

'If  $\text{info} = -i$ , the  $i$ th parameter had an illegal value.

## ?tgsja

*Computes the generalized SVD of two upper triangular or trapezoidal matrices.*

---

```
call stgsja ( jobu, jobv, jobq, m, p, n, k, l, a, lda, b, ldb, tola,
              tolb, alpha, beta, u, ldu, v, ldv, q, ldq, work, ncycle, info )
call dtgsja ( jobu, jobv, jobq, m, p, n, k, l, a, lda, b, ldb, tola,
              tolb, alpha, beta, u, ldu, v, ldv, q, ldq, work, ncycle, info )
call ctgsja ( jobu, jobv, jobq, m, p, n, k, l, a, lda, b, ldb, tola,
              tolb, alpha, beta, u, ldu, v, ldv, q, ldq, work, ncycle, info )
call ztgsja ( jobu, jobv, jobq, m, p, n, k, l, a, lda, b, ldb, tola,
              tolb, alpha, beta, u, ldu, v, ldv, q, ldq, work, ncycle, info )
```

### Discussion

This routine computes the generalized singular value decomposition (GSVD) of two real/complex upper triangular (or trapezoidal) matrices  $A$  and  $B$ . On entry, it is assumed that matrices  $A$  and  $B$  have the following forms, which may be obtained by the preprocessing subroutine [?ggsvp](#) from a general  $m$ -by- $n$  matrix  $A$  and  $p$ -by- $n$  matrix  $B$ :

$$A = \begin{matrix} & n-k-l & k & l \\ & 0 & A_{12} & A_{13} \\ & 0 & 0 & A_{23} \\ m-k-l & 0 & 0 & 0 \end{matrix}, \quad \text{if } m-k-l \geq 0$$

$$= \begin{matrix} & n-k-l & k & l \\ & 0 & A_{12} & A_{13} \\ & 0 & 0 & A_{23} \\ m-k & 0 & 0 & 0 \end{matrix}, \quad \text{if } m-k-l < 0$$

$$B = \begin{matrix} & n-k-l & k & l \\ & l & \left( \begin{matrix} 0 & 0 & B_{13} \\ 0 & 0 & 0 \end{matrix} \right) \\ p-l & & \end{matrix}$$

where the  $k$ -by- $k$  matrix  $A_{12}$  and  $l$ -by- $l$  matrix  $B_{13}$  are nonsingular upper triangular;  $A_{23}$  is  $l$ -by- $l$  upper triangular if  $m-k-l \geq 0$ , otherwise  $A_{23}$  is  $(m-k)$ -by- $l$  upper trapezoidal.

On exit,

$$U^H A Q = D_1 * (0 \ R), \quad V^H B Q = D_2 * (0 \ R),$$

where  $U$ ,  $V$  and  $Q$  are orthogonal/unitary matrices,  $R$  is a nonsingular upper triangular matrix, and  $D_1$  and  $D_2$  are “diagonal” matrices, which are of the following structures:

If  $m-k-l \geq 0$ ,

$$D_1 = \begin{matrix} & k & l \\ & l & \left( \begin{matrix} I & 0 \\ 0 & C \\ 0 & 0 \end{matrix} \right) \\ m-k-l & & \end{matrix}$$

$$D_2 = \begin{matrix} & k & l \\ & p-l & \left( \begin{matrix} 0 & S \\ 0 & 0 \end{matrix} \right) \\ & & \end{matrix}$$

$$(0 \ R) = \begin{matrix} & n-k-l & k & l \\ & l & \left( \begin{matrix} 0 & R_{11} & R_{12} \\ 0 & 0 & R_{22} \end{matrix} \right) \\ & & \end{matrix}$$

where

$$C = \text{diag}(\alpha(k+1), \dots, \alpha(k+1))$$

$$S = \text{diag}(\beta(k+1), \dots, \beta(k+1))$$

$$C^2 + S^2 = I$$

$R$  is stored in  $a(1:k+1, n-k-l+1:n)$  on exit.

If  $m-k-l < 0$ ,

$$k \quad m-k \quad k+l-m$$

$$D_1 = \begin{matrix} k \\ m-k \end{matrix} \begin{pmatrix} I & 0 & 0 \\ 0 & C & 0 \end{pmatrix}$$

$$D_2 = \begin{matrix} k \\ m-k \\ k+l-m \\ p-l \end{matrix} \begin{pmatrix} 0 & S & 0 \\ 0 & 0 & I \\ 0 & 0 & 0 \end{pmatrix}$$

$$(0 \ R) = \begin{matrix} n-k-l \\ k \\ m-k \\ k+l-m \end{matrix} \begin{pmatrix} 0 & R_{11} & R_{12} & R_{13} \\ 0 & 0 & R_{22} & R_{23} \\ 0 & 0 & 0 & R_{33} \end{pmatrix}$$

where

$$C = \text{diag}(\alpha(k+1), \dots, \alpha(m)),$$

$$S = \text{diag}(\beta(k+1), \dots, \beta(m)),$$

$$C^2 + S^2 = I$$

On exit,  $\begin{pmatrix} R_{11}R_{12}R_{13} \\ 0 & R_{22}R_{23} \end{pmatrix}$  is stored in  $a(1:m, n-k-l+1:n)$  and  $R_{33}$  is stored

in  $b(m-k+1:l, n+m-k-l+1:n)$ .

The computation of the orthogonal/unitary transformation matrices  $U$ ,  $V$  or  $Q$  is optional. These matrices may either be formed explicitly, or they may be postmultiplied into input matrices  $U_1$ ,  $V_1$ , or  $Q_1$ .

### Input Parameters

<i>jobu</i>	CHARACTER*1. Must be ' <b>U</b> ', ' <b>I</b> ', or ' <b>N</b> '. If <i>jobu</i> = ' <b>U</b> ', <i>u</i> must contain an orthogonal/unitary matrix $U_I$ on entry. If <i>jobu</i> = ' <b>I</b> ', <i>u</i> is initialized to the unit matrix. If <i>jobu</i> = ' <b>N</b> ', <i>u</i> is not computed.
<i>jobv</i>	CHARACTER*1. Must be ' <b>V</b> ', ' <b>I</b> ', or ' <b>N</b> '. If <i>jobv</i> = ' <b>V</b> ', <i>v</i> must contain an orthogonal/unitary matrix $V_I$ on entry. If <i>jobv</i> = ' <b>I</b> ', <i>v</i> is initialized to the unit matrix. If <i>jobv</i> = ' <b>N</b> ', <i>v</i> is not computed.
<i>jobq</i>	CHARACTER*1. Must be ' <b>Q</b> ', ' <b>I</b> ', or ' <b>N</b> '. If <i>jobq</i> = ' <b>Q</b> ', <i>q</i> must contain an orthogonal/unitary matrix $Q_I$ on entry. If <i>jobq</i> = ' <b>I</b> ', <i>q</i> is initialized to the unit matrix. If <i>jobq</i> = ' <b>N</b> ', <i>q</i> is not computed.
<i>m</i>	INTEGER. The number of rows of the matrix <i>A</i> ( <i>m</i> ≥ 0).
<i>p</i>	INTEGER. The number of rows of the matrix <i>B</i> ( <i>p</i> ≥ 0).
<i>n</i>	INTEGER. The number of columns of the matrices <i>A</i> and <i>B</i> ( <i>n</i> ≥ 0).
<i>k, l</i>	INTEGER. Specify the subblocks in the input matrices <i>A</i> and <i>B</i> , whose GSVD is going to be computed by ?tgsja.
<i>a, b, u, v, q, work</i>	REAL for stgsja DOUBLE PRECISION for dtgsja COMPLEX for ctgsja DOUBLE COMPLEX for ztgsja. Arrays: <i>a</i> ( <i>lda</i> , *) contains the <i>m</i> -by- <i>n</i> matrix <i>A</i> . The second dimension of <i>a</i> must be at least max(1, <i>n</i> ). <i>b</i> ( <i>ldb</i> , *) contains the <i>p</i> -by- <i>n</i> matrix <i>B</i> . The second dimension of <i>b</i> must be at least max(1, <i>n</i> ).

If  $\text{jobu} = 'U'$ ,  $u(\text{ldu}, *)$  must contain a matrix  $U_I$  (usually the orthogonal/unitary matrix returned by ?ggsvp).

The second dimension of  $u$  must be at least  $\max(1, m)$ .

If  $\text{jobv} = 'V'$ ,  $v(\text{ldv}, *)$  must contain a matrix  $V_I$  (usually the orthogonal/unitary matrix returned by ?ggsvp).

The second dimension of  $v$  must be at least  $\max(1, p)$ .

If  $\text{jobq} = 'Q'$ ,  $q(\text{ldq}, *)$  must contain a matrix  $Q_I$  (usually the orthogonal/unitary matrix returned by ?ggsvp).

The second dimension of  $q$  must be at least  $\max(1, n)$ .

$\text{work}(*)$  is a workspace array. The dimension of  $\text{work}$  must be at least  $\max(1, 2n)$ .

$\text{lda}$  INTEGER. The first dimension of  $a$ ; at least  $\max(1, m)$ .

$\text{ldb}$  INTEGER. The first dimension of  $b$ ; at least  $\max(1, p)$ .

$\text{ldu}$  INTEGER. The first dimension of the array  $u$ .  
 $\text{ldu} \geq \max(1, m)$  if  $\text{jobu} = 'U'$ ;  $\text{ldu} \geq 1$  otherwise.

$\text{ldv}$  INTEGER. The first dimension of the array  $v$ .  
 $\text{ldv} \geq \max(1, p)$  if  $\text{jobv} = 'V'$ ;  $\text{ldv} \geq 1$  otherwise.

$\text{ldq}$  INTEGER. The first dimension of the array  $q$ .  
 $\text{ldq} \geq \max(1, n)$  if  $\text{jobq} = 'Q'$ ;  $\text{ldq} \geq 1$  otherwise.

$\text{tola}, \text{tolb}$  REAL for single-precision flavors  
DOUBLE PRECISION for double-precision flavors.  
 $\text{tola}$  and  $\text{tolb}$  are the convergence criteria for the Jacobi-Kogbetliantz iteration procedure. Generally, they are the same as used in ?ggsvp :  
 $\text{tola} = \max(m, n) * \|A\| * \text{MACHEPS}$ ,  
 $\text{tolb} = \max(p, n) * \|B\| * \text{MACHEPS}$ .

## Output Parameters

$a$  On exit,  $a(n-k+1:n, 1:\min(k+1, m))$  contains the triangular matrix  $R$  or part of  $R$ .

<i>b</i>	On exit, if necessary, $b(m-k+1: l, n+m-k-l+1: n)$ contains a part of $R$ .
<i>alpha, beta</i>	<i>REAL</i> for single-precision flavors <i>DOUBLE PRECISION</i> for double-precision flavors. Arrays, <i>DIMENSION</i> at least $\max(1, n)$ . Contain the generalized singular value pairs of $A$ and $B$ :
	$\alpha(k+1:k) = 1$ , $\beta(k+1:k) = 0$ ,
	and if $m-k-l \geq 0$ , $\alpha(k+1:k+1) = \text{diag}(C)$ , $\beta(k+1:k+1) = \text{diag}(S)$ ,
	or if $m-k-l < 0$ , $\alpha(k+1:m) = C$ , $\alpha(m+1:k+1) = 0$ $\beta(k+1:m) = S$ , $\beta(m+1:k+1) = 1$ .
	Furthermore, if $k+l < n$ , $\alpha(k+1+1:n) = 0$ and $\beta(k+1+1:n) = 0$ .
<i>u</i>	If $\text{jobu} = 'I'$ , <i>u</i> contains the orthogonal/unitary matrix $U$ . If $\text{jobu} = 'U'$ , <i>u</i> contains the product $U_I U$ . If $\text{jobu} = 'N'$ , <i>u</i> is not referenced.
<i>v</i>	If $\text{jobv} = 'I'$ , <i>v</i> contains the orthogonal/unitary matrix $V$ . If $\text{jobv} = 'V'$ , <i>v</i> contains the product $V_I V$ . If $\text{jobv} = 'N'$ , <i>v</i> is not referenced.
<i>q</i>	If $\text{jobq} = 'I'$ , <i>q</i> contains the orthogonal/unitary matrix $Q$ . If $\text{jobq} = 'Q'$ , <i>q</i> contains the product $Q_I Q$ . If $\text{jobq} = 'N'$ , <i>q</i> is not referenced.
<i>ncycle</i>	<i>INTEGER</i> . The number of cycles required for convergence.

*info*

INTEGER.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*th parameter had an illegal value.

If *info* = 1, the procedure does not converge after MAXIT cycles.

## Driver Routines

Each of the LAPACK driver routines solves a complete problem. To arrive at the solution, driver routines typically call a sequence of appropriate [computational routines](#). Driver routines are described in the following sections:

[Linear Least Squares \(LLS\) Problems](#)  
[Generalized LLS Problems](#)  
[Symmetric Eigenproblems](#)  
[Nonsymmetric Eigenproblems](#)  
[Singular Value Decomposition](#)  
[Generalized Symmetric Definite Eigenproblems](#)  
[Generalized Nonsymmetric Eigenproblems](#)

### Linear Least Squares (LLS) Problems

This section describes LAPACK driver routines used for solving linear least-squares problems. [Table 5-8](#) lists routines described in more detail below.

**Table 5-8      Driver Routines for Solving LLS Problems**

Routine Name	Operation performed
<a href="#">?gels</a>	Uses QR or LQ factorization to solve a overdetermined or underdetermined linear system with full rank matrix.
<a href="#">?gelsy</a>	Computes the minimum-norm solution to a linear least squares problem using a complete orthogonal factorization of A.
<a href="#">?gelss</a>	Computes the minimum-norm solution to a linear least squares problem using the singular value decomposition of A.
<a href="#">?gelsd</a>	Computes the minimum-norm solution to a linear least squares problem using the singular value decomposition of A and a divide and conquer method.

---

## ?gels

Uses QR or LQ factorization to solve a  
overdetermined or underdetermined  
linear system with full rank matrix.

---

```
call sgels ( trans, m, n, nrhs, a, lda, b, ldb, work, lwork, info )
call dgels ( trans, m, n, nrhs, a, lda, b, ldb, work, lwork, info )
call cgels ( trans, m, n, nrhs, a, lda, b, ldb, work, lwork, info )
call zgels ( trans, m, n, nrhs, a, lda, b, ldb, work, lwork, info )
```

### Discussion

This routine solves overdetermined or underdetermined real/ complex linear systems involving an  $m$ -by- $n$  matrix  $A$ , or its transpose/ conjugate-transpose, using a  $QR$  or  $LQ$  factorization of  $A$ . It is assumed that  $A$  has full rank.

The following options are provided:

1. If  $\text{trans} = \text{'N'}$  and  $m \geq n$ : find the least squares solution of an overdetermined system, that is, solve the least squares problem  
$$\text{minimize } \| b - A x \|_2$$
2. If  $\text{trans} = \text{'N'}$  and  $m < n$ : find the minimum norm solution of an underdetermined system  $A X = B$ .
3. If  $\text{trans} = \text{'T'}$  or  $\text{'C'}$  and  $m \geq n$ : find the minimum norm solution of an undetermined system  $A^H X = B$ .
4. If  $\text{trans} = \text{'T'}$  or  $\text{'C'}$  and  $m < n$ : find the least squares solution of an overdetermined system, that is, solve the least squares problem  
$$\text{minimize } \| b - A^H x \|_2$$

Several right hand side vectors  $b$  and solution vectors  $x$  can be handled in a single call; they are stored as the columns of the  $m$ -by- $nrhs$  right hand side matrix  $B$  and the  $n$ -by- $nrh$  solution matrix  $X$ .

## Input Parameters

<code>trans</code>	<code>CHARACTER*1</code> . Must be ' <code>N</code> ', ' <code>T</code> ', or ' <code>C</code> '. If <code>trans</code> = ' <code>N</code> ', the linear system involves matrix $A$ ; If <code>trans</code> = ' <code>T</code> ', the linear system involves the transposed matrix $A^T$ (for real flavors only); If <code>trans</code> = ' <code>C</code> ', the linear system involves the conjugate-transposed matrix $A^H$ (for complex flavors only).
<code>m</code>	<code>INTEGER</code> . The number of rows of the matrix $A$ ( <code>m</code> $\geq 0$ ).
<code>n</code>	<code>INTEGER</code> . The number of columns of the matrix $A$ ( <code>n</code> $\geq 0$ ).
<code>nrhs</code>	<code>INTEGER</code> . The number of right-hand sides; the number of columns in $B$ ( <code>nrhs</code> $\geq 0$ ).
<code>a, b, work</code>	<code>REAL</code> for <code>sgels</code> <code>DOUBLE PRECISION</code> for <code>dgels</code> <code>COMPLEX</code> for <code>cgels</code> <code>DOUBLE COMPLEX</code> for <code>zgels</code> . Arrays: <code>a( lda, * )</code> contains the <code>m</code> -by- <code>n</code> matrix $A$ . The second dimension of <code>a</code> must be at least $\max(1, n)$ . <code>b( ldb, * )</code> contains the matrix $B$ of right hand side vectors, stored columnwise; $B$ is <code>m</code> -by- <code>nrhs</code> if <code>trans</code> = ' <code>N</code> ', or <code>n</code> -by- <code>nrhs</code> if <code>trans</code> = ' <code>T</code> ' or ' <code>C</code> '. The second dimension of <code>b</code> must be at least $\max(1, nrhs)$ . <code>work( lwork )</code> is a workspace array. <code>lda</code> <code>INTEGER</code> . The first dimension of <code>a</code> ; at least $\max(1, m)$ . <code>ldb</code> <code>INTEGER</code> . The first dimension of <code>b</code> ; must be at least $\max(1, m, n)$ . <code>lwork</code> <code>INTEGER</code> . The size of the <code>work</code> array; must be at least $\min(m, n) + \max(1, m, n, nrhs)$ . See <i>Application notes</i> for the suggested value of <code>lwork</code> .

## Output Parameters

- a* On exit, overwritten by the factorization data as follows:  
 if  $m \geq n$ , array *a* contains the details of the *QR* factorization of the matrix *A* as returned by [?geqrf](#);  
 if  $m < n$ , array *a* contains the details of the *LQ* factorization of the matrix *A* as returned by [?gelqf](#).
- b* Overwritten by the solution vectors, stored columnwise:  
 If *trans* = 'N' and  $m \geq n$ , rows 1 to *n* of *b* contain the least squares solution vectors; the residual sum of squares for the solution in each column is given by the sum of squares of elements *n*+1 to *m* in that column;  
 If *trans* = 'N' and  $m < n$ , rows 1 to *n* of *b* contain the minimum norm solution vectors;  
 if *trans* = 'T' or 'C' and  $m \geq n$ , rows 1 to *m* of *b* contain the minimum norm solution vectors;  
 if *trans* = 'T' or 'C' and  $m < n$ , rows 1 to *m* of *b* contain the least squares solution vectors; the residual sum of squares for the solution in each column is given by the sum of squares of elements *m*+1 to *n* in that column.
- work(1)* If *info* = 0, on exit *work(1)* contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.
- info* INTEGER.  
 If *info* = 0, the execution is successful.  
 If *info* = -*i*, the *i*th parameter had an illegal value.

## Application Notes

For better performance, try using  
 $lwork = \min(m, n) + \max(1, m, n, nrhs) * blocksize$ , where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run. On exit, examine *work(1)* and use this value for subsequent runs.

## ?gelsy

*Computes the minimum-norm solution to  
a linear least squares problem using a  
complete orthogonal factorization of A.*

```
call sgelsy ( m, n, nrhs, a, lda, b, ldb, jpvt, rcond, rank, work,
              lwork, info )
call dgelsy ( m, n, nrhs, a, lda, b, ldb, jpvt, rcond, rank, work,
              lwork, info )
call cgelsy ( m, n, nrhs, a, lda, b, ldb, jpvt, rcond, rank, work,
              lwork, rwork, info )
call zgelsy ( m, n, nrhs, a, lda, b, ldb, jpvt, rcond, rank, work,
              lwork, rwork, info )
```

### Discussion

This routine computes the minimum-norm solution to a real/complex linear least squares problem:

$$\text{minimize } \| b - Ax \|_2$$

using a complete orthogonal factorization of  $A$ .  $A$  is an  $m$ -by- $n$  matrix which may be rank-deficient.

Several right hand side vectors  $b$  and solution vectors  $x$  can be handled in a single call; they are stored as the columns of the  $m$ -by- $nrhs$  right hand side matrix  $B$  and the  $n$ -by- $nrhs$  solution matrix  $X$ .

The routine first computes a  $QR$  factorization with column pivoting:

$$AP = Q \begin{pmatrix} R_{11} & R_{12} \\ 0 & R_{22} \end{pmatrix}$$

with  $R_{11}$  defined as the largest leading submatrix whose estimated condition number is less than  $1/rcond$ . The order of  $R_{11}$ ,  $rank$ , is the effective rank of  $A$ .

Then,  $R_{22}$  is considered to be negligible, and  $R_{12}$  is annihilated by orthogonal/unitary transformations from the right, arriving at the complete orthogonal factorization:

$$AP = Q \begin{pmatrix} T_{11} & 0 \\ 0 & 0 \end{pmatrix} Z$$

The minimum-norm solution is then

$$x = PZ^H \begin{pmatrix} T_{11}^{-1} Q_1^H b \\ 0 \end{pmatrix}$$

where  $Q_1$  consists of the first *rank* columns of  $Q$ . This routine is basically identical to the original `?gelsx` except three differences:

- The call to the subroutine `?geqpf` has been substituted by the call to the subroutine `?geqp3`. This subroutine is a BLAS-3 version of the *QR* factorization with column pivoting.
- Matrix  $B$  (the right hand side) is updated with BLAS-3.
- The permutation of matrix  $B$  (the right hand side) is faster and more simple.

## Input Parameters

<i>m</i>	<b>INTEGER.</b> The number of rows of the matrix $A$ ( $m \geq 0$ ).
<i>n</i>	<b>INTEGER.</b> The number of columns of the matrix $A$ ( $n \geq 0$ ).
<i>nrhs</i>	<b>INTEGER.</b> The number of right-hand sides; the number of columns in $B$ ( $nrhs \geq 0$ ).
<i>a, b, work</i>	<b>REAL</b> for <code>sgelsy</code> <b>DOUBLE PRECISION</b> for <code>dgelsy</code> <b>COMPLEX</b> for <code>cgelsy</code> <b>DOUBLE COMPLEX</b> for <code>zgelsy</code> .
Arrays:	
<i>a</i> ( <i>lda</i> ,*)	contains the <i>m</i> -by- <i>n</i> matrix $A$ .
The second dimension of <i>a</i> must be at least $\max(1, n)$ .	

$b(\text{ldb}, *)$  contains the  $m$ -by- $nrhs$  right hand side matrix  $B$ .

The second dimension of  $b$  must be at least  $\max(1, nrhs)$ .

$work(lwork)$  is a workspace array.

$lda$  INTEGER. The first dimension of  $a$ ; at least  $\max(1, m)$ .

$ldb$  INTEGER. The first dimension of  $b$ ; must be at least  $\max(1, m, n)$ .

$jpvt$  INTEGER. Array, DIMENSION at least  $\max(1, n)$ .

On entry, if  $jpvt(i) \neq 0$ , the  $i$ th column of  $A$  is permuted to the front of  $AP$ , otherwise the  $i$ th column of  $A$  is a free column.

$rcond$  REAL for single-precision flavors

DOUBLE PRECISION for double-precision flavors.

$rcond$  is used to determine the effective rank of  $A$ , which is defined as the order of the largest leading triangular submatrix  $R_{11}$  in the  $QR$  factorization with pivoting of  $A$ , whose estimated condition number <  $1/rcond$ .

$lwork$  INTEGER. The size of the  $work$  array. See *Application notes* for the suggested value of  $lwork$ .

$rwork$  REAL for  $cgelsy$

DOUBLE PRECISION for  $zgelsy$ .

Workspace array, DIMENSION at least  $\max(1, 2n)$ . Used in complex flavors only.

## Output Parameters

$a$  On exit, overwritten by the details of the complete orthogonal factorization of  $A$ .

$b$  Overwritten by the  $n$ -by- $nrhs$  solution matrix  $X$ .

$jpvt$  On exit, if  $jpvt(i) = k$ , then the  $i$ th column of  $AP$  was the  $k$ th column of  $A$ .

*rank*`INTEGER.`

The effective rank of  $A$ , that is, the order of the submatrix  $R_{11}$ . This is the same as the order of the submatrix  $T_{11}$  in the complete orthogonal factorization of  $A$ .

*info*`INTEGER.`

If  $\text{info} = 0$ , the execution is successful.

If  $\text{info} = -i$ , the  $i$ th parameter had an illegal value.

## Application Notes

*For real flavors:*

The unblocked strategy requires that:

$$\text{lwork} \geq \max( mn+3n+1, 2*mn + nrhs ),$$

where  $mn = \min( m, n )$ .

The block algorithm requires that:

$$\text{lwork} \geq \max( mn+2n+nb*(n+1), 2*mn+nb*nrhs ),$$

where  $nb$  is an upper bound on the blocksize returned by `ilaenv` for the routines `sgeqp3/dgeqp3, stzrzf/dtzrzf, stzrjf/dtzrjf, sormqr/dormqr`, and `sormrz/dormrz`.

*For complex flavors:*

The unblocked strategy requires that:

$$\text{lwork} \geq mn + \max( 2*mn, n+1, mn + nrhs ),$$

where  $mn = \min( m, n )$ .

The block algorithm requires that:

$$\text{lwork} \geq mn + \max( 2*mn, nb*(n+1), mn+mn*nb, mn+nb*nrhs ),$$

where  $nb$  is an upper bound on the blocksize returned by `ilaenv` for the routines `cgeqp3/zgeqp3, ctzrzf/ztzrzf, ctzrjf/ztzrjf, cunmqr/zunmqr`, and `cunmrz/zunmrz`.

## ?gelss

Computes the minimum-norm solution to  
a linear least squares problem using the  
singular value decomposition of  $A$ .

```
call sgelss ( m, n, nrhs, a, lda, b, ldb, s, rcond, rank, work,
              lwork, info )
call dgelss ( m, n, nrhs, a, lda, b, ldb, s, rcond, rank, work,
              lwork, info )
call cgelss ( m, n, nrhs, a, lda, b, ldb, s, rcond, rank, work,
              lwork, rwork, info )
call zgelss ( m, n, nrhs, a, lda, b, ldb, s, rcond, rank, work,
              lwork, rwork, info )
```

### Discussion

This routine computes the minimum norm solution to a real linear least squares problem:

$$\text{minimize } \| b - A x \|_2$$

using the singular value decomposition (SVD) of  $A$ .  $A$  is an  $m$ -by- $n$  matrix which may be rank-deficient.

Several right hand side vectors  $b$  and solution vectors  $x$  can be handled in a single call; they are stored as the columns of the  $m$ -by- $nrhs$  right hand side matrix  $B$  and the  $n$ -by- $nrhs$  solution matrix  $X$ .

The effective rank of  $A$  is determined by treating as zero those singular values which are less than  $rcond$  times the largest singular value.

### Input Parameters

$m$	<b>INTEGER.</b> The number of rows of the matrix $A$ ( $m \geq 0$ ).
$n$	<b>INTEGER.</b> The number of columns of the matrix $A$ ( $n \geq 0$ ).
$nrhs$	<b>INTEGER.</b> The number of right-hand sides; the number of columns in $B$ ( $nrhs \geq 0$ ).

<i>a, b, work</i>	<small>REAL for sgelss DOUBLE PRECISION for dgelss COMPLEX for cgelss DOUBLE COMPLEX for zgelss.</small>
	<small>Arrays: <i>a( lda, * )</i> contains the <i>m</i>-by-<i>n</i> matrix <i>A</i>. The second dimension of <i>a</i> must be at least max(1, <i>n</i>). <i>b( ldb, * )</i> contains the <i>m</i>-by-<i>nrhs</i> right hand side matrix <i>B</i>. The second dimension of <i>b</i> must be at least max(1, <i>nrhs</i>). <i>work( lwork )</i> is a workspace array.</small>
<i>lda</i>	<small>INTEGER. The first dimension of <i>a</i>; at least max(1, <i>m</i>).</small>
<i>ldb</i>	<small>INTEGER. The first dimension of <i>b</i>; must be at least max(1, <i>m</i>, <i>n</i>).</small>
<i>rcond</i>	<small>REAL for single-precision flavors DOUBLE PRECISION for double-precision flavors. <i>rcond</i> is used to determine the effective rank of <i>A</i>. Singular values <i>s(i) ≤ rcond * s(1)</i> are treated as zero. If <i>rcond &lt; 0</i>, machine precision is used instead.</small>
<i>lwork</i>	<small>INTEGER. The size of the <i>work</i> array; <i>lwork</i> ≥ 1. See <i>Application notes</i> for the suggested value of <i>lwork</i>.</small>
<i>rwork</i>	<small>REAL for cgelss DOUBLE PRECISION for zgelss. Workspace array used in complex flavors only. DIMENSION at least max(1, 5*min(<i>m</i>, <i>n</i>)).</small>

## Output Parameters

<i>a</i>	On exit, the first min( <i>m</i> , <i>n</i> ) rows of <i>A</i> are overwritten with its right singular vectors, stored row-wise.
<i>b</i>	Overwritten by the <i>n</i> -by- <i>nrhs</i> solution matrix <i>X</i> . If <i>m ≥ n</i> and <i>rank = n</i> , the residual sum-of-squares for the solution in the <i>i</i> -th column is given by the sum of squares of elements <i>n+1:m</i> in that column.

---

<i>s</i>	<code>REAL</code> for single precision flavors <code>DOUBLE PRECISION</code> for double precision flavors. Array, <code>DIMENSION</code> at least $\max(1, \min(m, n))$ . The singular values of $A$ in decreasing order. The condition number of $A$ in the 2-norm is $k_2(A) = s(1) / s(\min(m, n)).$
<i>rank</i>	<code>INTEGER</code> . The effective rank of $A$ , that is, the number of singular values which are greater than $rcond * s(1)$ .
<i>work(1)</i>	If <code>info</code> = 0, on exit, <code>work(1)</code> contains the minimum value of <code>lwork</code> required for optimum performance. Use this <code>lwork</code> for subsequent runs.
<i>info</i>	<code>INTEGER</code> . If <code>info</code> = 0, the execution is successful. If <code>info</code> = $-i$ , the $i$ th parameter had an illegal value. If <code>info</code> = $i$ , then the algorithm for computing the SVD failed to converge; $i$ indicates the number of off-diagonal elements of an intermediate bidiagonal form which did not converge to zero.

## Application Notes

For real flavors:

$$lwork \geq 3 * \min(m, n) + \max(2 * \min(m, n), \max(m, n), nrhs)$$

For complex flavors:

$$lwork \geq 2 * \min(m, n) + \max(m, n, nrhs)$$

For good performance, `lwork` should generally be larger. If you are in doubt how much workspace to supply, use a generous value of `lwork` for the first run. On exit, examine `work(1)` and use this value for subsequent runs.

---

## ?gelsd

*Computes the minimum-norm solution to a linear least squares problem using the singular value decomposition of A and a divide and conquer method.*

---

```
call sgelsd ( m, n, nrhs, a, lda, b, ldb, s, rcond, rank, work,
              lwork, iwork, info )
call dgelsd ( m, n, nrhs, a, lda, b, ldb, s, rcond, rank, work,
              lwork, iwork, info )
call cgelsd ( m, n, nrhs, a, lda, b, ldb, s, rcond, rank, work,
              lwork, rwork, iwork, info )
call zgelsd ( m, n, nrhs, a, lda, b, ldb, s, rcond, rank, work,
              lwork, rwork, iwork, info )
```

### Discussion

This routine computes the minimum-norm solution to a real linear least squares problem:

$$\text{minimize } \| b - Ax \|_2$$

using the singular value decomposition (SVD) of  $A$ .  $A$  is an  $m$ -by- $n$  matrix which may be rank-deficient.

Several right hand side vectors  $b$  and solution vectors  $x$  can be handled in a single call; they are stored as the columns of the  $m$ -by- $nrhs$  right hand side matrix  $B$  and the  $n$ -by- $nrhs$  solution matrix  $X$ .

The problem is solved in three steps:

1. Reduce the coefficient matrix  $A$  to bidiagonal form with Householder transformations, reducing the original problem into a "bidiagonal least squares problem" (BLS).
2. Solve the BLS using a divide and conquer approach.
3. Apply back all the Householder transformations to solve the original least squares problem.

The effective rank of  $A$  is determined by treating as zero those singular values which are less than  $rcond$  times the largest singular value.

## Input Parameters

<i>m</i>	INTEGER. The number of rows of the matrix <i>A</i> ( <i>m</i> ≥ 0).
<i>n</i>	INTEGER. The number of columns of the matrix <i>A</i> ( <i>n</i> ≥ 0).
<i>nrhs</i>	INTEGER. The number of right-hand sides; the number of columns in <i>B</i> ( <i>nrhs</i> ≥ 0).
<i>a, b, work</i>	<p>REAL for <i>sgelsd</i>          DOUBLE PRECISION for <i>dgelsd</i>          COMPLEX for <i>cgelsd</i>          DOUBLE COMPLEX for <i>zgelsd</i>.</p> <p>Arrays:</p> <p><i>a( lda, * )</i> contains the <i>m</i>-by-<i>n</i> matrix <i>A</i>.          The second dimension of <i>a</i> must be at least <math>\max(1, n)</math>.</p> <p><i>b( ldb, * )</i> contains the <i>m</i>-by-<i>nrhs</i> right hand side matrix <i>B</i>.          The second dimension of <i>b</i> must be at least <math>\max(1, nrhs)</math>.</p> <p><i>work( lwork )</i> is a workspace array.</p>
<i>lda</i>	INTEGER. The first dimension of <i>a</i> ; at least $\max(1, m)$ .
<i>ldb</i>	INTEGER. The first dimension of <i>b</i> ; must be at least $\max(1, m, n)$ .
<i>rcond</i>	<p>REAL for single-precision flavors          DOUBLE PRECISION for double-precision flavors.</p> <p><i>rcond</i> is used to determine the effective rank of <i>A</i>.          Singular values <math>s(i) \leq rcond * s(1)</math> are treated as zero.          If <i>rcond</i> &lt; 0, machine precision is used instead.</p>
<i>lwork</i>	INTEGER. The size of the <i>work</i> array; <i>lwork</i> ≥ 1. See <i>Application notes</i> for the suggested value of <i>lwork</i> .
<i>iwork</i>	INTEGER. Workspace array. See <i>Application notes</i> for the suggested dimension of <i>iwork</i> .

## Output Parameters

<i>a</i>	On exit, the first $\min(m, n)$ rows of <i>A</i> are overwritten with its right singular vectors, stored row-wise.
<i>b</i>	Overwritten by the <i>n</i> -by- <i>nrhs</i> solution matrix <i>X</i> . If $m \geq n$ and <i>rank</i> = <i>n</i> , the residual sum-of-squares for the solution in the <i>i</i> -th column is given by the sum of squares of elements <i>n+1:m</i> in that column.
<i>s</i>	<b>REAL</b> for single precision flavors <b>DOUBLE PRECISION</b> for double precision flavors. Array, <b>DIMENSION</b> at least $\max(1, \min(m, n))$ . The singular values of <i>A</i> in decreasing order. The condition number of <i>A</i> in the 2-norm is $k_2(A) = s(1) / s(\min(m, n)).$
<i>rank</i>	<b>INTEGER</b> . The effective rank of <i>A</i> , that is, the number of singular values which are greater than <i>rcond</i> * <i>s</i> (1).
<i>work(1)</i>	If <i>info</i> = 0, on exit, <i>work(1)</i> contains the minimum value of <i>lwork</i> required for optimum performance. Use this <i>lwork</i> for subsequent runs.
<i>info</i>	<b>INTEGER</b> . If <i>info</i> = 0, the execution is successful. If <i>info</i> = <i>-i</i> , the <i>i</i> th parameter had an illegal value. If <i>info</i> = <i>i</i> , then the algorithm for computing the SVD failed to converge; <i>i</i> indicates the number of off-diagonal elements of an intermediate bidiagonal form which did not converge to zero.

## Application Notes

The divide and conquer algorithm makes very mild assumptions about floating point arithmetic. It will work on machines with a guard digit in add/subtract. It could conceivably fail on hexadecimal or decimal machines without guard digits, but we know of none.

The exact minimum amount of workspace needed depends on *m*, *n* and *nrhs*. The size of the workspace array *work* must be:

If  $m \geq n$ ,

$lwork \geq 11n + 2n*smlsiz + 8n*nlvl + n*nrhs$ ;

If  $m < n$ ,

$lwork \geq 11m + 2m*smlsiz + 8m*nlvl + m*nrhs$ ;

where  $smlsiz$  is returned by `ilaenv` and is equal to the maximum size of the subproblems at the bottom of the computation tree (usually about 25), and  $nlvl = \text{INT}(\log_2(\min(m, n)/(smlsiz+1))) + 1$ .

For good performance,  $lwork$  should generally be larger. If you are in doubt how much workspace to supply, use a generous value of  $lwork$  for the first run. On exit, examine `work(1)` and use this value for subsequent runs.

The dimension of the workspace array  $iwork$  must be at least  $3*\min(m, n)*nlvl + 11*\min(m, n)$ .

## Generalized LLS Problems

This section describes LAPACK driver routines used for solving generalized linear least-squares problems. [Table 5-9](#) lists routines described in more detail below.

**Table 5-9      Driver Routines for Solving Generalized LLS Problems**

Routine Name	Operation performed
<a href="#">?gglse</a>	Solves the linear equality-constrained least squares problem using a generalized RQ factorization.
<a href="#">?ggglm</a>	Solves a general Gauss-Markov linear model problem using a generalized QR factorization.

### ?gglse

*Solves the linear equality-constrained least squares problem using a generalized RQ factorization.*

```
call sgglse ( m, n, p, a, lda, b, ldb, c, d, x, work, lwork, info )
call dgglse ( m, n, p, a, lda, b, ldb, c, d, x, work, lwork, info )
call cgglse ( m, n, p, a, lda, b, ldb, c, d, x, work, lwork, info )
call zgglse ( m, n, p, a, lda, b, ldb, c, d, x, work, lwork, info )
```

### Discussion

This routine solves the linear equality-constrained least squares (LSE) problem:

$$\text{minimize } \| c - A x \|_2 \quad \text{subject to } B x = d$$

where  $A$  is an  $m$ -by- $n$  matrix,  $B$  is a  $p$ -by- $n$  matrix,  $c$  is a given  $m$ -vector, and  $d$  is a given  $p$ -vector.

It is assumed that  $p \leq n \leq m+p$ , and

$$\text{rank}(B) = p \quad \text{and} \quad \text{rank} \begin{pmatrix} A \\ B \end{pmatrix} = n .$$

These conditions ensure that the LSE problem has a unique solution, which is obtained using a generalized  $RQ$  factorization of the matrices  $B$  and  $A$ .

### Input Parameters

$m$  INTEGER. The number of rows of the matrix  $A$  ( $m \geq 0$ ).

$n$  INTEGER. The number of columns of the matrices  $A$  and  $B$  ( $n \geq 0$ ).

$p$  INTEGER. The number of rows of the matrix  $B$  ( $0 \leq p \leq n \leq m+p$ ).

$a, b, c, d, work$  REAL for sgglse  
DOUBLE PRECISION for dgglse  
COMPLEX for cgglse  
DOUBLE COMPLEX for zgglse.

Arrays:

$a( lda, * )$  contains the  $m$ -by- $n$  matrix  $A$ .

The second dimension of  $a$  must be at least  $\max(1, n)$ .

$b( ldb, * )$  contains the  $p$ -by- $n$  matrix  $B$ .

The second dimension of  $b$  must be at least  $\max(1, n)$ .

$c( * )$ , dimension at least  $\max(1, m)$ , contains the right hand side vector for the least squares part of the LSE problem.

$d( * )$ , dimension at least  $\max(1, p)$ , contains the right hand side vector for the constrained equation.

$work(lwork)$  is a workspace array.

$lda$  INTEGER. The first dimension of  $a$ ; at least  $\max(1, m)$ .

$ldb$  INTEGER. The first dimension of  $b$ ; at least  $\max(1, p)$ .

$lwork$  INTEGER. The size of the  $work$  array;

$lwork \geq \max(1, m+n+p)$ . See *Application notes* for the suggested value of  $lwork$ .

## Output Parameters

$x$	<code>REAL</code> for <code>sgglse</code> <code>DOUBLE PRECISION</code> for <code>dgglse</code> <code>COMPLEX</code> for <code>cgglse</code> <code>DOUBLE COMPLEX</code> for <code>zgglse</code> . Array, <code>DIMENSION</code> at least <code>max(1, n)</code> . On exit, contains the solution of the LSE problem.
$a, b, d$	On exit, these arrays are overwritten.
$c$	On exit, the residual sum-of-squares for the solution is given by the sum of squares of elements $n-p+1$ to $m$ of vector $c$ .
$work(1)$	If $info = 0$ , on exit, $work(1)$ contains the minimum value of $lwork$ required for optimum performance. Use this $lwork$ for subsequent runs.
$info$	<code>INTEGER</code> . If $info = 0$ , the execution is successful. If $info = -i$ , the $i$ th parameter had an illegal value.

## Application Notes

For optimum performance use

$$lwork \geq p + \min(m, n) + \max(m, n) * nb,$$

where  $nb$  is an upper bound for the optimal blocksizes for `?geqrf`, `?gerqf`, `?ormqr / ?unmqr` and `?ormrq / ?unmrq`.

## ?ggglm

*Solves a general Gauss-Markov linear model problem using a generalized QR factorization.*

```
call sgglm ( n, m, p, a, lda, b, ldb, d, x, y, work, lwork, info )
call dgglm ( n, m, p, a, lda, b, ldb, d, x, y, work, lwork, info )
call cgglm ( n, m, p, a, lda, b, ldb, d, x, y, work, lwork, info )
call zgglm ( n, m, p, a, lda, b, ldb, d, x, y, work, lwork, info )
```

### Discussion

This routine solves a general Gauss-Markov linear model (GLM) problem:

$$\text{minimize}_x \|y\|_2 \text{ subject to } d = Ax + By$$

where  $A$  is an  $n$ -by- $m$  matrix,  $B$  is an  $n$ -by- $p$  matrix, and  $d$  is a given  $n$ -vector.

It is assumed that  $m \leq n \leq m+p$ , and

$$\text{rank}(A) = m \text{ and } \text{rank}(AB) = n.$$

Under these assumptions, the constrained equation is always consistent, and there is a unique solution  $x$  and a minimal 2-norm solution  $y$ , which is obtained using a generalized QR factorization of  $A$  and  $B$ .

In particular, if matrix  $B$  is square nonsingular, then the problem GLM is equivalent to the following weighted linear least squares problem

$$\text{minimize}_x \|B^{-1}(d - Ax)\|_2.$$

### Input Parameters

$n$	<b>INTEGER.</b> The number of rows of the matrices $A$ and $B$ ( $n \geq 0$ ).
$m$	<b>INTEGER.</b> The number of columns in $A$ ( $m \geq 0$ ).
$p$	<b>INTEGER.</b> The number of columns in $B$ ( $p \geq n - m$ ).
$a, b, d, work$	<b>REAL</b> for <code>sgglm</code> <b>DOUBLE PRECISION</b> for <code>dgglm</code> <b>COMPLEX</b> for <code>cgglm</code> <b>DOUBLE COMPLEX</b> for <code>zgglm</code> .

Arrays:

$a(\text{lda}, *)$  contains the  $n$ -by- $m$  matrix  $A$ .

The second dimension of  $a$  must be at least  $\max(1, m)$ .

$b(\text{ldb}, *)$  contains the  $n$ -by- $p$  matrix  $B$ .

The second dimension of  $b$  must be at least  $\max(1, p)$ .

$d(*),$  dimension at least  $\max(1, n)$ , contains the left hand side of the GLM equation.

$work(lwork)$  is a workspace array.

$lda$	<code>INTEGER</code> . The first dimension of $a$ ; at least $\max(1, n)$ .
$ldb$	<code>INTEGER</code> . The first dimension of $b$ ; at least $\max(1, n)$ .
$lwork$	<code>INTEGER</code> . The size of the $work$ array; $lwork \geq \max(1, n+m+p)$ . See <i>Application notes</i> for the suggested value of $lwork$ .

## Output Parameters

$x, y$	<code>REAL</code> for <code>sggglm</code> <code>DOUBLE PRECISION</code> for <code>dggglm</code> <code>COMPLEX</code> for <code>cggglm</code> <code>DOUBLE COMPLEX</code> for <code>zggglm</code> . Arrays $x(*), y(*)$ . <code>DIMENSION</code> at least $\max(1, m)$ for $x$ and at least $\max(1, p)$ for $y$ . On exit, $x$ and $y$ are the solutions of the GLM problem.
$a, b, d$	On exit, these arrays are overwritten.
$work(1)$	If $info = 0$ , on exit, $work(1)$ contains the minimum value of $lwork$ required for optimum performance.
$info$	<code>INTEGER</code> . If $info = 0$ , the execution is successful. If $info = -i$ , the $i$ th parameter had an illegal value.

## Application Notes

For optimum performance use

$$lwork \geq m + \min(n, p) + \max(n, p) * nb,$$

where  $nb$  is an upper bound for the optimal blocksizes for `?geqrf`, `?gerqf`, `?ormqr / ?unmqr` and `?ormrq / ?unmrq`.

## Symmetric Eigenproblems

This section describes LAPACK driver routines used for solving symmetric eigenvalue problems. See also [computational routines](#) that can be called to solve these problems.

[Table 5-10](#) lists routines described in more detail below.

**Table 5-10 Driver Routines for Solving Symmetric Eigenproblems**

Routine Name	Operation performed
<a href="#">?syev / ?heev</a>	Computes all eigenvalues and, optionally, eigenvectors of a real symmetric / Hermitian matrix.
<a href="#">?syevd / ?heevd</a>	Computes all eigenvalues and (optionally) all eigenvectors of a real symmetric / Hermitian matrix using divide and conquer algorithm.
<a href="#">?syevx / ?heevx</a>	Computes selected eigenvalues and, optionally, eigenvectors of a symmetric / Hermitian matrix.
<a href="#">?syevr / ?heevr</a>	Computes selected eigenvalues and, optionally, eigenvectors of a real symmetric / Hermitian matrix using the Relatively Robust Representations.
<a href="#">?spev / ?hpev</a>	Computes all eigenvalues and, optionally, eigenvectors of a real symmetric / Hermitian matrix in packed storage.
<a href="#">?spevd / ?hpevd</a>	Uses divide and conquer algorithm to compute all eigenvalues and (optionally) all eigenvectors of a real symmetric / Hermitian matrix held in packed storage.
<a href="#">?spevx / ?hpevx</a>	Computes selected eigenvalues and, optionally, eigenvectors of a real symmetric / Hermitian matrix in packed storage.
<a href="#">?sbev / ?hbev</a>	Computes all eigenvalues and, optionally, eigenvectors of a real symmetric / Hermitian band matrix.
<a href="#">?sbefd / ?hbefd</a>	Computes all eigenvalues and (optionally) all eigenvectors of a real symmetric / Hermitian band matrix using divide and conquer algorithm.
<a href="#">?sbevx / ?hbevx</a>	Computes selected eigenvalues and, optionally, eigenvectors of a real symmetric / Hermitian band matrix.
<a href="#">?stev</a>	Computes all eigenvalues and, optionally, eigenvectors of a real symmetric tridiagonal matrix.
<a href="#">?stevd</a>	Computes all eigenvalues and (optionally) all eigenvectors of a real symmetric tridiagonal matrix using divide and conquer algorithm.
<a href="#">?stevx</a>	Computes selected eigenvalues and eigenvectors of a real symmetric tridiagonal matrix.
<a href="#">?stevr</a>	Computes selected eigenvalues and, optionally, eigenvectors of a real symmetric tridiagonal matrix using the Relatively Robust Representations.

---

## ?syev

*Computes all eigenvalues and,  
optionally, eigenvectors of a real  
symmetric matrix.*

---

```
call ssyev ( jobz, uplo, n, a, lda, w, work, lwork, info )
call dsyev ( jobz, uplo, n, a, lda, w, work, lwork, info )
```

### Discussion

This routine computes all eigenvalues and, optionally, eigenvectors of a real symmetric matrix  $A$ .

### Input Parameters

<i>jobz</i>	CHARACTER*1. Must be ' <b>N</b> ' or ' <b>V</b> '. If <i>jobz</i> = ' <b>N</b> ', then only eigenvalues are computed. If <i>jobz</i> = ' <b>V</b> ', then eigenvalues and eigenvectors are computed.
<i>uplo</i>	CHARACTER*1. Must be ' <b>U</b> ' or ' <b>L</b> '. If <i>uplo</i> = ' <b>U</b> ', <i>a</i> stores the upper triangular part of $A$ . If <i>uplo</i> = ' <b>L</b> ', <i>a</i> stores the lower triangular part of $A$ .
<i>n</i>	INTEGER. The order of the matrix $A$ ( <i>n</i> $\geq$ 0).
<i>a</i> , <i>work</i>	REAL for <b>ssyev</b> DOUBLE PRECISION for <b>dsyev</b> Arrays: <i>a</i> ( <i>lda</i> ,*) is an array containing either upper or lower triangular part of the symmetric matrix $A$ , as specified by <i>uplo</i> . The second dimension of <i>a</i> must be at least max(1, <i>n</i> ). <i>work</i> ( <i>lwork</i> ) is a workspace array.
<i>lda</i>	INTEGER. The first dimension of the array <i>a</i> . Must be at least max(1, <i>n</i> ).

*lwork*            **INTEGER.** The dimension of the array *work*.  
 Constraint:  $lwork \geq \max(1, 3n - 1)$ . See *Application notes* for the suggested value of *lwork*.

### Output Parameters

*a*                On exit, if  $jobz = 'V'$ , then if *info* = 0, array *a* contains the orthonormal eigenvectors of the matrix *A*. If  $jobz = 'N'$ , then on exit the lower triangle (if *uplo* = 'L') or the upper triangle (if *uplo* = 'U') of *A*, including the diagonal, is overwritten.

*w*                **REAL** for *ssyev*  
**DOUBLE PRECISION** for *dsyev*  
 Array, **DIMENSION** at least  $\max(1, n)$ . If *info* = 0, contains the eigenvalues of the matrix *A* in ascending order.

*work(1)*        On exit, if *lwork* > 0, then *work(1)* returns the required minimal size of *lwork*.

*info*              **INTEGER.**  
 If *info* = 0, the execution is successful.  
 If *info* =  $-i$ , the *i*th parameter had an illegal value.  
 If *info* = *i*, then the algorithm failed to converge; *i* indicates the number of elements of an intermediate tridiagonal form which did not converge to zero.

### Application Notes

For optimum performance use

$$lwork \geq (nb+2)*n,$$

where *nb* is the blocksize for *?sytrd* returned by *ilaenv*.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run. On exit, examine *work(1)* and use this value for subsequent runs.

---

## ?heev

*Computes all eigenvalues and,  
optionally, eigenvectors of a Hermitian  
matrix.*

---

```
call cheev ( jobz, uplo, n, a, lda, w, work, lwork, rwork, info )
call zheev ( jobz, uplo, n, a, lda, w, work, lwork, rwork, info )
```

### Discussion

This routine computes all eigenvalues and, optionally, eigenvectors of a complex Hermitian matrix  $A$ .

### Input Parameters

<i>jobz</i>	CHARACTER*1. Must be ' <b>N</b> ' or ' <b>V</b> '. If <i>jobz</i> = ' <b>N</b> ', then only eigenvalues are computed. If <i>jobz</i> = ' <b>V</b> ', then eigenvalues and eigenvectors are computed.
<i>uplo</i>	CHARACTER*1. Must be ' <b>U</b> ' or ' <b>L</b> '. If <i>uplo</i> = ' <b>U</b> ', <i>a</i> stores the upper triangular part of $A$ . If <i>uplo</i> = ' <b>L</b> ', <i>a</i> stores the lower triangular part of $A$ .
<i>n</i>	INTEGER. The order of the matrix $A$ ( <i>n</i> $\geq$ 0).
<i>a</i> , <i>work</i>	COMPLEX for <b>cheev</b> DOUBLE COMPLEX for <b>zheev</b> Arrays: <i>a</i> ( <i>lda</i> ,*) is an array containing either upper or lower triangular part of the Hermitian matrix $A$ , as specified by <i>uplo</i> . The second dimension of <i>a</i> must be at least max(1, <i>n</i> ). <i>work</i> ( <i>lwork</i> ) is a workspace array.
<i>lda</i>	INTEGER. The first dimension of the array <i>a</i> . Must be at least max(1, <i>n</i> ).

<i>lwork</i>	<b>INTEGER.</b> The dimension of the array <i>work</i> . Constraint: $lwork \geq \max(1, 2n-1)$ . See <i>Application notes</i> for the suggested value of <i>lwork</i> .
<i>rwork</i>	<b>REAL</b> for <i>cheev</i> <b>DOUBLE PRECISION</b> for <i>zheev</i> . Workspace array, <b>DIMENSION</b> at least $\max(1, 3n-2)$ .

## Output Parameters

<i>a</i>	On exit, if $jobz = 'V'$ , then if <i>info</i> = 0, array <i>a</i> contains the orthonormal eigenvectors of the matrix <i>A</i> . If $jobz = 'N'$ , then on exit the lower triangle (if <i>uplo</i> = 'L') or the upper triangle (if <i>uplo</i> = 'U') of <i>A</i> , including the diagonal, is overwritten.
<i>w</i>	<b>REAL</b> for <i>cheev</i> <b>DOUBLE PRECISION</b> for <i>zheev</i> Array, <b>DIMENSION</b> at least $\max(1, n)$ . If <i>info</i> = 0, contains the eigenvalues of the matrix <i>A</i> in ascending order.
<i>work(1)</i>	On exit, if <i>lwork</i> > 0, then <i>work(1)</i> returns the required minimal size of <i>lwork</i> .
<i>info</i>	<b>INTEGER.</b> If <i>info</i> = 0, the execution is successful. If <i>info</i> = <i>-i</i> , the <i>i</i> th parameter had an illegal value. If <i>info</i> = <i>i</i> , then the algorithm failed to converge; <i>i</i> indicates the number of elements of an intermediate tridiagonal form which did not converge to zero.

## Application Notes

For optimum performance use

$$lwork \geq (nb+1)*n,$$

where *nb* is the blocksize for *?hetrd* returned by *ilaenv*.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run. On exit, examine *work(1)* and use this value for subsequent runs.

---

## ?syevd

*Computes all eigenvalues and  
(optionally) all eigenvectors of a real  
symmetric matrix using divide and  
conquer algorithm.*

---

```
call ssyevd ( job, uplo, n, a, lda, w, work, lwork, iwork, liwork, info)
call dsyevd ( job, uplo, n, a, lda, w, work, lwork, iwork, liwork, info)
```

### Discussion

This routine computes all the eigenvalues, and optionally all the eigenvectors, of a real symmetric matrix  $A$ . In other words, it can compute the spectral factorization of  $A$  as:  $A = Z\Lambda Z^T$ .

Here  $\Lambda$  is a diagonal matrix whose diagonal elements are the eigenvalues  $\lambda_i$ , and  $Z$  is the orthogonal matrix whose columns are the eigenvectors  $z_i$ . Thus,

$$Az_i = \lambda_i z_i \text{ for } i = 1, 2, \dots, n.$$

If the eigenvectors are requested, then this routine uses a divide and conquer algorithm to compute eigenvalues and eigenvectors. However, if only eigenvalues are required, then it uses the Pal-Walker-Kahan variant of the  $QL$  or  $QR$  algorithm.

### Input Parameters

<i>job</i>	CHARACTER*1. Must be ' <b>N</b> ' or ' <b>V</b> '. If <i>job</i> = ' <b>N</b> ', then only eigenvalues are computed. If <i>job</i> = ' <b>V</b> ', then eigenvalues and eigenvectors are computed.
<i>uplo</i>	CHARACTER*1. Must be ' <b>U</b> ' or ' <b>L</b> '. If <i>uplo</i> = ' <b>U</b> ', <i>a</i> stores the upper triangular part of $A$ . If <i>uplo</i> = ' <b>L</b> ', <i>a</i> stores the lower triangular part of $A$ .
<i>n</i>	INTEGER. The order of the matrix $A$ ( <i>n</i> $\geq 0$ ).
<i>a</i>	REAL for <b>ssyevd</b> DOUBLE PRECISION for <b>dsyevd</b> Array, DIMENSION ( <i>lda</i> , *).

$a(\text{lda}, *)$  is an array containing either upper or lower triangular part of the symmetric matrix  $A$ , as specified by  $\text{uplo}$ .

The second dimension of  $a$  must be at least  $\max(1, n)$ .

$\text{lda}$

**INTEGER.** The first dimension of the array  $a$ .

Must be at least  $\max(1, n)$ .

$\text{work}$

**REAL** for `ssyevd`

**DOUBLE PRECISION** for `dsyevd`.

Workspace array, **DIMENSION** at least  $\text{lwork}$ .

$\text{lwork}$

**INTEGER.** The dimension of the array  $\text{work}$ .

Constraints:

if  $n \leq 1$ , then  $\text{lwork} \geq 1$ ;

if  $\text{job} = 'N'$  and  $n > 1$ , then  $\text{lwork} \geq 2n+1$ ;

if  $\text{job} = 'V'$  and  $n > 1$ , then

$\text{lwork} \geq 3n^2 + (5+2k)*n+1$ , where  $k$  is the smallest integer which satisfies  $2^k \geq n$ .

$iwork$

**INTEGER.**

Workspace array, **DIMENSION** at least  $\text{liwork}$ .

$\text{liwork}$

**INTEGER.** The dimension of the array  $iwork$ .

Constraints:

if  $n \leq 1$ , then  $\text{liwork} \geq 1$ ;

if  $\text{job} = 'N'$  and  $n > 1$ , then  $\text{liwork} \geq 1$ ;

if  $\text{job} = 'V'$  and  $n > 1$ , then  $\text{liwork} \geq 5n+2$ .

## Output Parameters

$w$

**REAL** for `ssyevd`

**DOUBLE PRECISION** for `dsyevd`

Array, **DIMENSION** at least  $\max(1, n)$ .

If  $\text{info} = 0$ , contains the eigenvalues of the matrix  $A$  in ascending order.

See also  $\text{info}$ .

$a$

If  $\text{job} = 'V'$ , then on exit this array is overwritten by the orthogonal matrix  $Z$  which contains the eigenvectors of  $A$ .

<i>work(1)</i>	On exit, if <i>lwork</i> > 0, then <i>work(1)</i> returns the required minimal size of <i>lwork</i> .
<i>iwork(1)</i>	On exit, if <i>liwork</i> > 0, then <i>iwork(1)</i> returns the required minimal size of <i>liwork</i> .
<i>info</i>	<b>INTEGER.</b> If <i>info</i> = 0, the execution is successful. If <i>info</i> = <i>i</i> , then the algorithm failed to converge; <i>i</i> indicates the number of elements of an intermediate tridiagonal form which did not converge to zero. If <i>info</i> = - <i>i</i> , the <i>i</i> th parameter had an illegal value.

### Application Notes

The computed eigenvalues and eigenvectors are exact for a matrix  $T + E$  such that  $\|E\|_2 = O(\epsilon) \|T\|_2$ , where  $\epsilon$  is the machine precision.

The complex analogue of this routine is [?heevd](#).

## ?heevd

*Computes all eigenvalues and (optionally) all eigenvectors of a complex Hermitian matrix using divide and conquer algorithm.*

```
call cheevd (job, uplo, n, a, lda, w, work, lwork, rwork, lrwork,
             iwork, liwork, info)
call zheevd (job, uplo, n, a, lda, w, work, lwork, rwork, lrwork,
             iwork, liwork, info)
```

### Discussion

This routine computes all the eigenvalues, and optionally all the eigenvectors, of a complex Hermitian matrix  $A$ . In other words, it can compute the spectral factorization of  $A$  as:  $A = Z\Lambda Z^H$ .

Here  $\Lambda$  is a real diagonal matrix whose diagonal elements are the eigenvalues  $\lambda_i$ , and  $Z$  is the (complex) unitary matrix whose columns are the eigenvectors  $z_i$ . Thus,

$$Az_i = \lambda_i z_i \text{ for } i = 1, 2, \dots, n.$$

If the eigenvectors are requested, then this routine uses a divide and conquer algorithm to compute eigenvalues and eigenvectors. However, if only eigenvalues are required, then it uses the Pal-Walker-Kahan variant of the  $QL$  or  $QR$  algorithm.

### Input Parameters

<i>job</i>	CHARACTER*1. Must be ' <b>N</b> ' or ' <b>V</b> '. If <i>job</i> = ' <b>N</b> ', then only eigenvalues are computed. If <i>job</i> = ' <b>V</b> ', then eigenvalues and eigenvectors are computed.
<i>uplo</i>	CHARACTER*1. Must be ' <b>U</b> ' or ' <b>L</b> '. If <i>uplo</i> = ' <b>U</b> ', <i>a</i> stores the upper triangular part of $A$ . If <i>uplo</i> = ' <b>L</b> ', <i>a</i> stores the lower triangular part of $A$ .
<i>n</i>	INTEGER. The order of the matrix $A$ ( <i>n</i> $\geq 0$ ).

<i>a</i>	COMPLEX for <code>cheevd</code> DOUBLE COMPLEX for <code>zheevd</code> Array, DIMENSION ( <i>lda</i> , *). <i>a</i> ( <i>lda</i> , *) is an array containing either upper or lower triangular part of the Hermitian matrix <i>A</i> , as specified by <i>uplo</i> . The second dimension of <i>a</i> must be at least $\max(1, n)$ .
<i>lda</i>	INTEGER. The first dimension of the array <i>a</i> . Must be at least $\max(1, n)$ .
<i>work</i>	COMPLEX for <code>cheevd</code> DOUBLE COMPLEX for <code>zheevd</code> . Workspace array, DIMENSION at least <i>lwork</i> .
<i>lwork</i>	INTEGER. The dimension of the array <i>work</i> . Constraints: if <i>n</i> ≤ 1, then <i>lwork</i> ≥ 1; if <i>job</i> = 'N' and <i>n</i> > 1, then <i>lwork</i> ≥ <i>n</i> +1; if <i>job</i> = 'V' and <i>n</i> > 1, then <i>lwork</i> ≥ $n^2+2n$
<i>rwork</i>	REAL for <code>cheevd</code> DOUBLE PRECISION for <code>zheevd</code> Workspace array, DIMENSION at least <i>lrwork</i> .
<i>lrwork</i>	INTEGER. The dimension of the array <i>rwork</i> . Constraints: if <i>n</i> ≤ 1, then <i>lrwork</i> ≥ 1; if <i>job</i> = 'N' and <i>n</i> > 1, then <i>lrwork</i> ≥ <i>n</i> ; if <i>job</i> = 'V' and <i>n</i> > 1, then <i>lrwork</i> ≥ $3n^2+(4+2k)*n+1$ , where <i>k</i> is the smallest integer which satisfies $2^k \geq n$ .
<i>iwork</i>	INTEGER. Workspace array, DIMENSION at least <i>liwork</i> .
<i>liwork</i>	INTEGER. The dimension of the array <i>iwork</i> . Constraints: if <i>n</i> ≤ 1, then <i>liwork</i> ≥ 1; if <i>job</i> = 'N' and <i>n</i> > 1, then <i>liwork</i> ≥ 1; if <i>job</i> = 'V' and <i>n</i> > 1, then <i>liwork</i> ≥ $5n+2$ .

## Output Parameters

<i>w</i>	REAL for <code>cheevd</code> DOUBLE PRECISION for <code>zheevd</code> Array, DIMENSION at least max(1, <i>n</i> ) . If <i>info</i> = 0, contains the eigenvalues of the matrix <i>A</i> in ascending order. See also <i>info</i> .
<i>a</i>	If <i>job</i> = 'V', then on exit this array is overwritten by the unitary matrix <i>Z</i> which contains the eigenvectors of <i>A</i> .
<i>work(1)</i>	On exit, if <i>lwork</i> > 0, then the real part of <i>work(1)</i> returns the required minimal size of <i>lwork</i> .
<i>rwork(1)</i>	On exit, if <i>lrwork</i> > 0, then <i>rwork(1)</i> returns the required minimal size of <i>lrwork</i> .
<i>iwork(1)</i>	On exit, if <i>liwork</i> > 0, then <i>iwork(1)</i> returns the required minimal size of <i>liwork</i> .
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = <i>i</i> , then the algorithm failed to converge; <i>i</i> indicates the number of elements of an intermediate tridiagonal form which did not converge to zero. If <i>info</i> = - <i>i</i> , the <i>i</i> th parameter had an illegal value.

## Application Notes

The computed eigenvalues and eigenvectors are exact for a matrix  $A + E$  such that  $\|E\|_2 = O(\epsilon) \|A\|_2$ , where  $\epsilon$  is the machine precision.

The real analogue of this routine is [?syevd](#).

See also [?hpevd](#) for matrices held in packed storage, and [?hbevd](#) for banded matrices.

---

## ?syevx

*Computes selected eigenvalues and, optionally, eigenvectors of a symmetric matrix.*

---

```
call ssyevx ( jobz, range, uplo, n, a, lda, vl, vu, il, iu, abstol,
              m, w, z, ldz, work, lwork, iwork, ifail, info)
call dsyevx ( jobz, range, uplo, n, a, lda, vl, vu, il, iu, abstol,
              m, w, z, ldz, work, lwork, iwork, ifail, info)
```

### Discussion

This routine computes selected eigenvalues and, optionally, eigenvectors of a real symmetric matrix  $A$ . Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

### Input Parameters

<i>jobz</i>	CHARACTER*1. Must be ' <b>N</b> ' or ' <b>V</b> '. If <i>jobz</i> = ' <b>N</b> ', then only eigenvalues are computed. If <i>jobz</i> = ' <b>V</b> ', then eigenvalues and eigenvectors are computed.
<i>range</i>	CHARACTER*1. Must be ' <b>A</b> ', ' <b>V</b> ', or ' <b>I</b> '. If <i>range</i> = ' <b>A</b> ', all eigenvalues will be found. If <i>range</i> = ' <b>V</b> ', all eigenvalues in the half-open interval ( <i>vl</i> , <i>vu</i> ] will be found. If <i>range</i> = ' <b>I</b> ', the eigenvalues with indices <i>il</i> through <i>iu</i> will be found.
<i>uplo</i>	CHARACTER*1. Must be ' <b>U</b> ' or ' <b>L</b> '. If <i>uplo</i> = ' <b>U</b> ', <i>a</i> stores the upper triangular part of $A$ . If <i>uplo</i> = ' <b>L</b> ', <i>a</i> stores the lower triangular part of $A$ .
<i>n</i>	INTEGER. The order of the matrix $A$ ( <i>n</i> $\geq$ 0).

<i>a, work</i>	<small>REAL for ssyevx DOUBLE PRECISION for dsyevx.</small>
	Arrays: <i>a( lda, * )</i> is an array containing either upper or lower triangular part of the symmetric matrix <i>A</i> , as specified by <i>uplo</i> . The second dimension of <i>a</i> must be at least $\max(1, n)$ . <i>work(lwork)</i> is a workspace array.
<i>lda</i>	<small>INTEGER. The first dimension of the array <i>a</i>. Must be at least <math>\max(1, n)</math>.</small>
<i>vl, vu</i>	<small>REAL for ssyevx DOUBLE PRECISION for dsyevx. If <i>range</i> = 'V', the lower and upper bounds of the interval to be searched for eigenvalues; <i>vl</i> <math>\leq</math> <i>vu</i>. Not referenced if <i>range</i> = 'A' or 'I'.</small>
<i>il, iu</i>	<small>INTEGER. If <i>range</i> = 'I', the indices of the smallest and largest eigenvalues to be returned. Constraints: <math>1 \leq il \leq iu \leq n</math>, if <i>n</i> &gt; 0; <i>il</i> = 1 and <i>iu</i> = 0, if <i>n</i> = 0. Not referenced if <i>range</i> = 'A' or 'V'.</small>
<i>abstol</i>	<small>REAL for ssyevx DOUBLE PRECISION for dsyevx. The absolute error tolerance for the eigenvalues. See <i>Application notes</i> for more information.</small>
<i>ldz</i>	<small>INTEGER. The first dimension of the output array <i>z</i>; <i>ldz</i> <math>\geq 1</math>. If <i>jobz</i> = 'V', then <i>ldz</i> <math>\geq \max(1, n)</math>.</small>
<i>lwork</i>	<small>INTEGER. The dimension of the array <i>work</i>. Constraint: <i>lwork</i> <math>\geq \max(1, 8n)</math>. See <i>Application notes</i> for the suggested value of <i>lwork</i>.</small>
<i>iwork</i>	<small>INTEGER. Workspace array, DIMENSION at least <math>\max(1, 5n)</math>.</small>

## Output Parameters

<i>a</i>	On exit, the lower triangle (if <i>uplo</i> = 'L') or the upper triangle (if <i>uplo</i> = 'U') of <i>A</i> , including the diagonal, is overwritten.
<i>m</i>	<b>INTEGER.</b> The total number of eigenvalues found; $0 \leq m \leq n$ . If <i>range</i> = 'A', <i>m</i> = <i>n</i> , and if <i>range</i> = 'I', <i>m</i> = <i>iu</i> - <i>il</i> + 1.
<i>w</i>	<b>REAL</b> for <i>ssyevx</i> <b>DOUBLE PRECISION</b> for <i>dsyevx</i> Array, <b>DIMENSION</b> at least $\max(1, n)$ . The first <i>m</i> elements contain the selected eigenvalues of the matrix <i>A</i> in ascending order.
<i>z</i>	<b>REAL</b> for <i>ssyevx</i> <b>DOUBLE PRECISION</b> for <i>dsyevx</i> . Array <i>z</i> ( <i>ldz</i> , *) contains eigenvectors. The second dimension of <i>z</i> must be at least $\max(1, m)$ . If <i>jobz</i> = 'V', then if <i>info</i> = 0, the first <i>m</i> columns of <i>z</i> contain the orthonormal eigenvectors of the matrix <i>A</i> corresponding to the selected eigenvalues, with the <i>i</i> -th column of <i>z</i> holding the eigenvector associated with <i>w</i> ( <i>i</i> ). If an eigenvector fails to converge, then that column of <i>z</i> contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in <i>ifail</i> . If <i>jobz</i> = 'N', then <i>z</i> is not referenced. Note: you must ensure that at least $\max(1, m)$ columns are supplied in the array <i>z</i> ; if <i>range</i> = 'V', the exact value of <i>m</i> is not known in advance and an upper bound must be used.
<i>work(1)</i>	On exit, if <i>lwork</i> > 0, then <i>work(1)</i> returns the required minimal size of <i>lwork</i> .
<i>ifail</i>	<b>INTEGER.</b> Array, <b>DIMENSION</b> at least $\max(1, n)$ . If <i>jobz</i> = 'V', then if <i>info</i> = 0, the first <i>m</i> elements of <i>ifail</i> are zero; if <i>info</i> > 0, then <i>ifail</i> contains the indices of the eigenvectors that failed to converge. If <i>jobz</i> = 'N', then <i>ifail</i> is not referenced.

*info*

INTEGER.

If *info* = 0, the execution is successful.If *info* = -*i*, the *i*th parameter had an illegal value.If *info* = *i*, then *i* eigenvectors failed to converge;  
their indices are stored in the array *ifail*.

## Application Notes

For optimum performance use  $lwork \geq (nb+3)*n$ , where *nb* is the maximum of the blocksize for **?sytrd** and **?ormtr** returned by **ilaenv**. If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run. On exit, examine *work(1)* and use this value for subsequent runs.

An approximate eigenvalue is accepted as converged when it is determined to lie in an interval [a,b] of width less than or equal to  $abstol + \varepsilon * \max(|a|,|b|)$ , where  $\varepsilon$  is the machine precision. If *abstol* is less than or equal to zero, then  $\varepsilon * |T|$  will be used in its place, where  $|T|$  is the 1-norm of the tridiagonal matrix obtained by reducing *A* to tridiagonal form.

Eigenvalues will be computed most accurately when *abstol* is set to twice the underflow threshold  $2 * \text{slamch}('S')$ , not zero. If this routine returns with *info* > 0, indicating that some eigenvectors did not converge, try setting *abstol* to  $2 * \text{slamch}('S')$ .

---

## ?heevx

*Computes selected eigenvalues and, optionally, eigenvectors of a Hermitian matrix.*

---

```
call cheevx (jobz, range, uplo, n, a, lda, vl, vu, il, iu, abstol,
             m, w, z, ldz, work, lwork, rwork, iwork, ifail, info)
call zheevx (jobz, range, uplo, n, a, lda, vl, vu, il, iu, abstol,
             m, w, z, ldz, work, lwork, rwork, iwork, ifail, info)
```

### Discussion

This routine computes selected eigenvalues and, optionally, eigenvectors of a complex Hermitian matrix  $A$ . Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

### Input Parameters

<i>jobz</i>	CHARACTER*1. Must be ' <b>N</b> ' or ' <b>V</b> '. If <i>jobz</i> = ' <b>N</b> ', then only eigenvalues are computed. If <i>jobz</i> = ' <b>V</b> ', then eigenvalues and eigenvectors are computed.
<i>range</i>	CHARACTER*1. Must be ' <b>A</b> ', ' <b>V</b> ', or ' <b>I</b> '. If <i>range</i> = ' <b>A</b> ', all eigenvalues will be found. If <i>range</i> = ' <b>V</b> ', all eigenvalues in the half-open interval ( <i>vl</i> , <i>vu</i> ] will be found. If <i>range</i> = ' <b>I</b> ', the eigenvalues with indices <i>il</i> through <i>iu</i> will be found.
<i>uplo</i>	CHARACTER*1. Must be ' <b>U</b> ' or ' <b>L</b> '. If <i>uplo</i> = ' <b>U</b> ', <i>a</i> stores the upper triangular part of $A$ . If <i>uplo</i> = ' <b>L</b> ', <i>a</i> stores the lower triangular part of $A$ .
<i>n</i>	INTEGER. The order of the matrix $A$ ( <i>n</i> $\geq$ 0).

---

<i>a, work</i>	COMPLEX for <code>cheevx</code> DOUBLE COMPLEX for <code>zheevx</code> . Arrays: <i>a( lda, * )</i> is an array containing either upper or lower triangular part of the Hermitian matrix <i>A</i> , as specified by <i>uplo</i> . The second dimension of <i>a</i> must be at least $\max(1, n)$ . <i>work(lwork)</i> is a workspace array.
<i>lda</i>	INTEGER. The first dimension of the array <i>a</i> . Must be at least $\max(1, n)$ .
<i>vl, vu</i>	REAL for <code>cheevx</code> DOUBLE PRECISION for <code>zheevx</code> . If <i>range</i> = 'V', the lower and upper bounds of the interval to be searched for eigenvalues; <i>vl</i> $\leq$ <i>vu</i> . Not referenced if <i>range</i> = 'A' or 'I'.
<i>il, iu</i>	INTEGER. If <i>range</i> = 'I', the indices of the smallest and largest eigenvalues to be returned. Constraints: $1 \leq il \leq iu \leq n$ , if <i>n</i> > 0; <i>il</i> = 1 and <i>iu</i> = 0, if <i>n</i> = 0. Not referenced if <i>range</i> = 'A' or 'V'.
<i>abstol</i>	REAL for <code>cheevx</code> DOUBLE PRECISION for <code>zheevx</code> . The absolute error tolerance for the eigenvalues . See <i>Application notes</i> for more information.
<i>ldz</i>	INTEGER. The first dimension of the output array <i>z</i> ; <i>ldz</i> $\geq 1$ . If <i>jobz</i> = 'V', then <i>ldz</i> $\geq \max(1, n)$ .
<i>lwork</i>	INTEGER. The dimension of the array <i>work</i> . Constraint: <i>lwork</i> $\geq \max(1, 2n-1)$ . See <i>Application notes</i> for the suggested value of <i>lwork</i> .
<i>rwork</i>	REAL for <code>cheevx</code> DOUBLE PRECISION for <code>zheevx</code> . Workspace array, DIMENSION at least $\max(1, 7n)$ .
<i>iwork</i>	INTEGER. Workspace array, DIMENSION at least $\max(1, 5n)$ .

## Output Parameters

<i>a</i>	On exit, the lower triangle (if <i>uplo</i> = 'L') or the upper triangle (if <i>uplo</i> = 'U') of <i>A</i> , including the diagonal, is overwritten.
<i>m</i>	<b>INTEGER.</b> The total number of eigenvalues found; $0 \leq m \leq n$ . If <i>range</i> = 'A', <i>m</i> = <i>n</i> , and if <i>range</i> = 'I', <i>m</i> = <i>iu</i> - <i>il</i> + 1.
<i>w</i>	<b>REAL</b> for <i>cheevx</i> <b>DOUBLE PRECISION</b> for <i>zheevx</i> Array, <b>DIMENSION</b> at least $\max(1, n)$ . The first <i>m</i> elements contain the selected eigenvalues of the matrix <i>A</i> in ascending order.
<i>z</i>	<b>COMPLEX</b> for <i>cheevx</i> <b>DOUBLE COMPLEX</b> for <i>zheevx</i> . Array <i>z</i> ( <i>ldz</i> , *) contains eigenvectors. The second dimension of <i>z</i> must be at least $\max(1, m)$ . If <i>jobz</i> = 'V', then if <i>info</i> = 0, the first <i>m</i> columns of <i>z</i> contain the orthonormal eigenvectors of the matrix <i>A</i> corresponding to the selected eigenvalues, with the <i>i</i> -th column of <i>z</i> holding the eigenvector associated with <i>w</i> ( <i>i</i> ). If an eigenvector fails to converge, then that column of <i>z</i> contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in <i>ifail</i> . If <i>jobz</i> = 'N', then <i>z</i> is not referenced. Note: you must ensure that at least $\max(1, m)$ columns are supplied in the array <i>z</i> ; if <i>range</i> = 'V', the exact value of <i>m</i> is not known in advance and an upper bound must be used.
<i>work(1)</i>	On exit, if <i>lwork</i> > 0, then <i>work(1)</i> returns the required minimal size of <i>lwork</i> .
<i>ifail</i>	<b>INTEGER.</b> Array, <b>DIMENSION</b> at least $\max(1, n)$ . If <i>jobz</i> = 'V', then if <i>info</i> = 0, the first <i>m</i> elements of <i>ifail</i> are zero; if <i>info</i> > 0, then <i>ifail</i> contains the indices of the eigenvectors that failed to converge. If <i>jobz</i> = 'N', then <i>ifail</i> is not referenced.

*info*

INTEGER.

If *info* = 0, the execution is successful.If *info* = -*i*, the *i*th parameter had an illegal value.If *info* = *i*, then *i* eigenvectors failed to converge;  
their indices are stored in the array *ifail*.

## Application Notes

For optimum performance use  $lwork \geq (nb+1)*n$ , where *nb* is the maximum of the blocksize for ?hetrd and ?unmtr returned by ilaenv. If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run. On exit, examine *work(1)* and use this value for subsequent runs.

An approximate eigenvalue is accepted as converged when it is determined to lie in an interval [a,b] of width less than or equal to  $abstol + \varepsilon * \max(|a|,|b|)$ , where  $\varepsilon$  is the machine precision. If *abstol* is less than or equal to zero, then  $\varepsilon * |T|$  will be used in its place, where  $|T|$  is the 1-norm of the tridiagonal matrix obtained by reducing *A* to tridiagonal form.

Eigenvalues will be computed most accurately when *abstol* is set to twice the underflow threshold  $2 * \text{slamch}('S')$ , not zero. If this routine returns with *info* > 0, indicating that some eigenvectors did not converge, try setting *abstol* to  $2 * \text{slamch}('S')$ .

---

**?syevr**

*Computes selected eigenvalues and, optionally, eigenvectors of a real symmetric matrix using the Relatively Robust Representations.*

---

```
call ssyevr ( jobz, range, uplo, n, a, lda, vl, vu, il, iu, abstol,
              m, w, z, ldz, isuppz, work, lwork, iwork, liwork, info)
call dsyevr ( jobz, range, uplo, n, a, lda, vl, vu, il, iu, abstol,
              m, w, z, ldz, isuppz, work, lwork, iwork, liwork, info)
```

**Discussion**

This routine computes selected eigenvalues and, optionally, eigenvectors of a real symmetric matrix  $T$ . Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

Whenever possible, **?syevr** calls **ssteqr/dsteqr** to compute the eigenspectrum using Relatively Robust Representations. **?steqr** computes eigenvalues by the *dqds* algorithm, while orthogonal eigenvectors are computed from various "good"  $LDL^T$  representations (also known as Relatively Robust Representations). Gram-Schmidt orthogonalization is avoided as far as possible. More specifically, the various steps of the algorithm are as follows. For the  $i$ -th unreduced block of  $T$ ,

- (a) Compute  $T - \sigma_i = L_i D_i L_i^T$ , such that  $L_i D_i L_i^T$  is a relatively robust representation;
- (b) Compute the eigenvalues,  $\lambda_j$ , of  $L_i D_i L_i^T$  to high relative accuracy by the *dqds* algorithm;
- (c) If there is a cluster of close eigenvalues, "choose"  $\sigma_i$  close to the cluster, and go to step (a);
- (d) Given the approximate eigenvalue  $\lambda_j$  of  $L_i D_i L_i^T$ , compute the corresponding eigenvector by forming a rank-revealing twisted factorization.

The desired accuracy of the output can be specified by the input parameter *abstol*.

The routine `?syevr` calls `ssteqr/dsteqr` when the full spectrum is requested on machines which conform to the IEEE-754 floating point standard. `?syevr` calls `sstebz/dstebz` and `sstein/dstein` on non-IEEE machines and when partial spectrum requests are made.

### Input Parameters

<i>jobz</i>	CHARACTER*1. Must be ' <code>N</code> ' or ' <code>V</code> '. If <i>jobz</i> = ' <code>N</code> ', then only eigenvalues are computed. If <i>jobz</i> = ' <code>V</code> ', then eigenvalues and eigenvectors are computed.
<i>range</i>	CHARACTER*1. Must be ' <code>A</code> ' or ' <code>V</code> ' or ' <code>I</code> '. If <i>range</i> = ' <code>A</code> ', the routine computes all eigenvalues. If <i>range</i> = ' <code>V</code> ', the routine computes eigenvalues $\lambda_i$ in the half-open interval: $v_l < \lambda_i \leq v_u$ . If <i>range</i> = ' <code>I</code> ', the routine computes eigenvalues with indices <i>il</i> to <i>iu</i> .  For <i>range</i> = ' <code>V</code> ' or ' <code>I</code> ', <code>sstebz/dstebz</code> and <code>sstein/dstein</code> are called.
<i>n</i>	INTEGER. The order of the matrix <i>A</i> ( <i>n</i> $\geq 0$ ).
<i>a</i> , <i>work</i>	REAL for <code>ssyevr</code> DOUBLE PRECISION for <code>dsyevr</code> . Arrays: <i>a</i> ( <i>lda</i> , *) is an array containing either upper or lower triangular part of the symmetric matrix <i>A</i> , as specified by <i>uplo</i> . The second dimension of <i>a</i> must be at least $\max(1, n)$ . <i>work</i> ( <i>lwork</i> ) is a workspace array.
<i>lda</i>	INTEGER. The first dimension of the array <i>a</i> . Must be at least $\max(1, n)$ .

<i>vl, vu</i>	<b>REAL</b> for <b>ssyevr</b> <b>DOUBLE PRECISION</b> for <b>dsyevr</b> . If <i>range</i> = 'V', the lower and upper bounds of the interval to be searched for eigenvalues. Constraint: <i>vl</i> < <i>vu</i> .
<i>il, iu</i>	If <i>range</i> = 'A' or 'I', <i>vl</i> and <i>vu</i> are not referenced. <b>INTEGER</b> . If <i>range</i> = 'I', the indices in ascending order of the smallest and largest eigenvalues to be returned. Constraint: $1 \leq il \leq iu \leq n$ , if <i>n</i> > 0; <i>il</i> =1 and <i>iu</i> =0 if <i>n</i> = 0.
<i>abstol</i>	If <i>range</i> = 'A' or 'V', <i>il</i> and <i>iu</i> are not referenced. <b>REAL</b> for <b>ssyevr</b> <b>DOUBLE PRECISION</b> for <b>dsyevr</b> . The absolute error tolerance to which each eigenvalue/eigenvector is required. If <i>jobz</i> = 'V', the eigenvalues and eigenvectors output have residual norms bounded by <i>abstol</i> , and the dot products between different eigenvectors are bounded by <i>abstol</i> . If <i>abstol</i> < <i>n</i> ε   <i>T</i>    <sub>1</sub> , then <i>n</i> ε   <i>T</i>    <sub>1</sub> will be used in its place, where ε is the machine precision. The eigenvalues are computed to an accuracy of ε   <i>T</i>    <sub>1</sub> irrespective of <i>abstol</i> . If high relative accuracy is important, set <i>abstol</i> to ?lamch('S').
<i>ldz</i>	<b>INTEGER</b> . The leading dimension of the output array <i>z</i> . Constraints: <i>ldz</i> ≥ 1 if <i>jobz</i> = 'N'; <i>ldz</i> ≥ max(1, <i>n</i> ) if <i>jobz</i> = 'V'.
<i>lwork</i>	<b>INTEGER</b> . The dimension of the array <i>work</i> . Constraint: <i>lwork</i> ≥ max(1, 26 <i>n</i> ). See <i>Application notes</i> for the suggested value of <i>lwork</i> .
<i>iwork</i>	<b>INTEGER</b> . Workspace array, <b>DIMENSION</b> ( <i>liwork</i> ).
<i>liwork</i>	<b>INTEGER</b> . The dimension of the array <i>iwork</i> , <i>lwork</i> ≥ max(1, 10 <i>n</i> ).

## Output Parameters

- a* On exit, the lower triangle (if *uplo* = 'L') or the upper triangle (if *uplo* = 'U') of *A*, including the diagonal, is overwritten.
- m* **INTEGER.** The total number of eigenvalues found,  $0 \leq m \leq n$ . If *range* = 'A', *m* = *n*, and if *range* = 'I', *m* = *iu* - *il* + 1.
- w, z* **REAL** for *ssyevr*  
**DOUBLE PRECISION** for *dsyevr*.  
 Arrays:  
*w*(\*), **DIMENSION** at least max(1, *n*), contains the selected eigenvalues in ascending order, stored in *w*(1) to *w*(*m*);  
*z*(*ldz*, \*), the second dimension of *z* must be at least max(1, *m*).  
 If *jobz* = 'V', then if *info* = 0, the first *m* columns of *z* contain the orthonormal eigenvectors of the matrix *T* corresponding to the selected eigenvalues, with the *i*-th column of *z* holding the eigenvector associated with *w*(*i*).  
 If *jobz* = 'N', then *z* is not referenced.  
 Note: you must ensure that at least max(1, *m*) columns are supplied in the array *z*; if *range* = 'V', the exact value of *m* is not known in advance and an upper bound must be used.
- isuppz* **INTEGER.**  
 Array, **DIMENSION** at least  $2 * \text{max}(1, m)$ .  
 The support of the eigenvectors in *z*, i.e., the indices indicating the nonzero elements in *z*. The *i*-th eigenvector is nonzero only in elements *isuppz*( $2i-1$ ) through *isuppz*( $2i$ ).
- work(1)* On exit, if *info* = 0, then *work(1)* returns the required minimal size of *lwork*.
- iwork(1)* On exit, if *info* = 0, then *iwork(1)* returns the required minimal size of *liwork*.

*info*

INTEGER.

If *info* = 0, the execution is successful.If *info* = -*i*, the *i*th parameter had an illegal value.If *info* = *i*, an internal error has occurred.

## Application Notes

For optimum performance use  $lwork \geq (nb+6)*n$ , where *nb* is the maximum of the blocksize for `?sytrd` and `?ormtr` returned by `ilaenv`. If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run. On exit, examine `work(1)` and use this value for subsequent runs.

Normal execution of `?stegr` may create NaNs and infinities and hence may abort due to a floating point exception in environments which do not handle NaNs and infinities in the IEEE standard default manner.

## ?heevr

*Computes selected eigenvalues and, optionally, eigenvectors of a Hermitian matrix using the Relatively Robust Representations.*

```
call cheevr ( jobz, range, uplo, n, a, lda, vl, vu, il, iu, abstol,
              m, w, z, ldz, isuppz, work, lwork, rwork, lrwork,
              iwork, liwork, info)
call zheevr ( jobz, range, uplo, n, a, lda, vl, vu, il, iu, abstol,
              m, w, z, ldz, isuppz, work, lwork, rwork, lrwork,
              iwork, liwork, info)
```

### Discussion

This routine computes selected eigenvalues and, optionally, eigenvectors of a complex Hermitian matrix  $T$ . Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

Whenever possible, `?heevr` calls `csteqr/zsteqr` to compute the eigenspectrum using Relatively Robust Representations. `?steqr` computes eigenvalues by the *dqds* algorithm, while orthogonal eigenvectors are computed from various "good"  $LDL^T$  representations (also known as Relatively Robust Representations). Gram-Schmidt orthogonalization is avoided as far as possible. More specifically, the various steps of the algorithm are as follows. For the  $i$ -th unreduced block of  $T$ ,

- (a) Compute  $T - \sigma_i = L_i D_i L_i^T$ , such that  $L_i D_i L_i^T$  is a relatively robust representation;
- (b) Compute the eigenvalues,  $\lambda_j$ , of  $L_i D_i L_i^T$  to high relative accuracy by the *dqds* algorithm;
- (c) If there is a cluster of close eigenvalues, "choose"  $\sigma_i$  close to the cluster, and go to step (a);
- (d) Given the approximate eigenvalue  $\lambda_j$  of  $L_i D_i L_i^T$ , compute the corresponding eigenvector by forming a rank-revealing twisted factorization.

The desired accuracy of the output can be specified by the input parameter *abstol*.

The routine `?heevr` calls `csteqr/zsteqr` when the full spectrum is requested on machines which conform to the IEEE-754 floating point standard. `?heevr` calls `sstebz/dstebz` and `cstein/zstein` on non-IEEE machines and when partial spectrum requests are made.

## Input Parameters

<i>jobz</i>	CHARACTER*1. Must be ' <code>N</code> ' or ' <code>V</code> '. If <i>job</i> = ' <code>N</code> ', then only eigenvalues are computed. If <i>job</i> = ' <code>V</code> ', then eigenvalues and eigenvectors are computed.
<i>range</i>	CHARACTER*1. Must be ' <code>A</code> ' or ' <code>V</code> ' or ' <code>I</code> '. If <i>range</i> = ' <code>A</code> ', the routine computes all eigenvalues. If <i>range</i> = ' <code>V</code> ', the routine computes eigenvalues $\lambda_i$ in the half-open interval: $v_l < \lambda_i \leq v_u$ . If <i>range</i> = ' <code>I</code> ', the routine computes eigenvalues with indices <i>il</i> to <i>iu</i> .  For <i>range</i> = ' <code>V</code> ' or ' <code>I</code> ', <code>sstebz/dstebz</code> and <code>cstein/zstein</code> are called.
<i>n</i>	INTEGER. The order of the matrix <i>A</i> ( <i>n</i> $\geq 0$ ).
<i>a</i> , <i>work</i>	COMPLEX for <code>cheevr</code> DOUBLE COMPLEX for <code>zheevr</code> .  Arrays: <i>a</i> ( <i>lda</i> , *) is an array containing either upper or lower triangular part of the Hermitian matrix <i>A</i> , as specified by <i>uplo</i> . The second dimension of <i>a</i> must be at least $\max(1, n)$ . <i>work</i> ( <i>lwork</i> ) is a workspace array.
<i>lda</i>	INTEGER. The first dimension of the array <i>a</i> . Must be at least $\max(1, n)$ .

---

<i>vl, vu</i>	<small>REAL for cheevr DOUBLE PRECISION for zheevr.</small> If <i>range</i> = 'V', the lower and upper bounds of the interval to be searched for eigenvalues. Constraint: <i>vl</i> < <i>vu</i> . If <i>range</i> = 'A' or 'I', <i>vl</i> and <i>vu</i> are not referenced.
<i>il, iu</i>	<small>INTEGER. If <i>range</i> = 'I', the indices in ascending order of the smallest and largest eigenvalues to be returned. Constraint: <math>1 \leq il \leq iu \leq n</math>, if <i>n</i> &gt; 0; <i>il</i>=1 and <i>iu</i>=0 if <i>n</i> = 0. If <i>range</i> = 'A' or 'V', <i>il</i> and <i>iu</i> are not referenced.</small>
<i>abstol</i>	<small>REAL for cheevr DOUBLE PRECISION for zheevr. The absolute error tolerance to which each eigenvalue/eigenvector is required. If <i>jobz</i> = 'V', the eigenvalues and eigenvectors output have residual norms bounded by <i>abstol</i>, and the dot products between different eigenvectors are bounded by <i>abstol</i>. If <i>abstol</i> &lt; <i>n</i><math>\epsilon\ T\ _1</math>, then <i>n</i><math>\epsilon\ T\ _1</math> will be used in its place, where <math>\epsilon</math> is the machine precision. The eigenvalues are computed to an accuracy of <math>\epsilon\ T\ _1</math> irrespective of <i>abstol</i>. If high relative accuracy is important, set <i>abstol</i> to ?lamch('S').</small>
<i>ldz</i>	<small>INTEGER. The leading dimension of the output array <i>z</i>. Constraints: <i>ldz</i> <math>\geq 1</math> if <i>jobz</i> = 'N'; <i>ldz</i> <math>\geq \max(1, n)</math> if <i>jobz</i> = 'V'.</small>
<i>lwork</i>	<small>INTEGER. The dimension of the array <i>work</i>. Constraint: <i>lwork</i> <math>\geq \max(1, 2n)</math>. See <i>Application notes</i> for the suggested value of <i>lwork</i>.</small>
<i>rwork</i>	<small>REAL for cheevr DOUBLE PRECISION for zheevr. Workspace array, DIMENSION (<i>lrwork</i>).</small>

---

<i>lrwork</i>	<b>INTEGER.</b> The dimension of the array <i>rwork</i> ; <i>lwork</i> $\geq \max(1, 24n)$ . .
<i>iwork</i>	<b>INTEGER.</b> Workspace array, <b>DIMENSION</b> ( <i>liwork</i> ).
<i>liwork</i>	<b>INTEGER.</b> The dimension of the array <i>iwork</i> , <i>lwork</i> $\geq \max(1, 10n)$ .

## Output Parameters

<i>a</i>	On exit, the lower triangle (if <i>uplo</i> = 'L') or the upper triangle (if <i>uplo</i> = 'U') of <i>A</i> , including the diagonal, is overwritten.
<i>m</i>	<b>INTEGER.</b> The total number of eigenvalues found, $0 \leq m \leq n$ . If <i>range</i> = 'A', <i>m</i> = <i>n</i> , and if <i>range</i> = 'I', <i>m</i> = <i>iu</i> - <i>il</i> + 1.
<i>w</i>	<b>REAL</b> for <i>cheevr</i> <b>DOUBLE PRECISION</b> for <i>zheevr</i> . Array, <b>DIMENSION</b> at least $\max(1, n)$ , contains the selected eigenvalues in ascending order, stored in <i>w(1)</i> to <i>w(m)</i> .
<i>z</i>	<b>COMPLEX</b> for <i>cheevr</i> <b>DOUBLE COMPLEX</b> for <i>zheevr</i> . Array <i>z(ldz, *)</i> ; the second dimension of <i>z</i> must be at least $\max(1, m)$ . If <i>jobz</i> = 'V', then if <i>info</i> = 0, the first <i>m</i> columns of <i>z</i> contain the orthonormal eigenvectors of the matrix <i>T</i> corresponding to the selected eigenvalues, with the <i>i</i> -th column of <i>z</i> holding the eigenvector associated with <i>w(i)</i> . If <i>jobz</i> = 'N', then <i>z</i> is not referenced. Note: you must ensure that at least $\max(1, m)$ columns are supplied in the array <i>z</i> ; if <i>range</i> = 'V', the exact value of <i>m</i> is not known in advance and an upper bound must be used.
<i>isuppz</i>	<b>INTEGER.</b> Array, <b>DIMENSION</b> at least $2 * \max(1, m)$ .

The support of the eigenvectors in  $\mathbf{z}$ , i.e., the indices indicating the nonzero elements in  $\mathbf{z}$ . The  $i$ -th eigenvector is nonzero only in elements  $\text{isuppz}(2i-1)$  through  $\text{isuppz}(2i)$ .

$\text{work}(1)$	On exit, if $\text{info} = 0$ , then $\text{work}(1)$ returns the required minimal size of $\text{lwork}$ .
$\text{rwork}(1)$	On exit, if $\text{info} = 0$ , then $\text{rwork}(1)$ returns the required minimal size of $\text{lrwork}$ .
$\text{iwork}(1)$	On exit, if $\text{info} = 0$ , then $\text{iwork}(1)$ returns the required minimal size of $\text{liwork}$ .
$\text{info}$	<p><b>INTEGER.</b></p> <p>If <math>\text{info} = 0</math>, the execution is successful.</p> <p>If <math>\text{info} = -i</math>, the <math>i</math>th parameter had an illegal value.</p> <p>If <math>\text{info} = i</math>, an internal error has occurred.</p>

### Application Notes

For optimum performance use  $\text{lwork} \geq (nb+1)*n$ , where  $nb$  is the maximum of the blocksize for  $\text{?hetrd}$  and  $\text{?unmtr}$  returned by  $\text{ilaenv}$ . If you are in doubt how much workspace to supply, use a generous value of  $\text{lwork}$  for the first run. On exit, examine  $\text{work}(1)$  and use this value for subsequent runs.

Normal execution of  $\text{?steqr}$  may create NaNs and infinities and hence may abort due to a floating point exception in environments which do not handle NaNs and infinities in the IEEE standard default manner.

---

## ?spev

*Computes all eigenvalues and,  
optionally, eigenvectors of a real  
symmetric matrix in packed storage.*

---

```
call sspev (jobz, uplo, n, ap, w, z, ldz, work, info)
call dspev (jobz, uplo, n, ap, w, z, ldz, work, info)
```

### Discussion

This routine computes all the eigenvalues and, optionally, eigenvectors of a real symmetric matrix  $A$  in packed storage.

### Input Parameters

<i>jobz</i>	CHARACTER*1. Must be ' <b>N</b> ' or ' <b>V</b> '. If <i>job</i> = ' <b>N</b> ', then only eigenvalues are computed. If <i>job</i> = ' <b>V</b> ', then eigenvalues and eigenvectors are computed.
<i>uplo</i>	CHARACTER*1. Must be ' <b>U</b> ' or ' <b>L</b> '. If <i>uplo</i> = ' <b>U</b> ', <i>ap</i> stores the packed upper triangular part of $A$ . If <i>uplo</i> = ' <b>L</b> ', <i>ap</i> stores the packed lower triangular part of $A$ .
<i>n</i>	INTEGER. The order of the matrix $A$ ( $n \geq 0$ ).
<i>ap, work</i>	REAL for <b>sspev</b> DOUBLE PRECISION for <b>dspev</b> Arrays: <i>ap</i> (*) contains the packed upper or lower triangle of symmetric matrix $A$ , as specified by <i>uplo</i> . The dimension of <i>ap</i> must be at least $\max(1, n*(n+1)/2)$ . <i>work</i> (*) is a workspace array, DIMENSION at least $\max(1, 3n)$ .

*ldz*            **INTEGER.** The leading dimension of the output array *z*.

Constraints:

if *jobz* = 'N', then *ldz*  $\geq 1$ ;  
 if *jobz* = 'V', then *ldz*  $\geq \max(1, n)$ .

## Output Parameters

*w, z*            **REAL** for *sspev*

**DOUBLE PRECISION** for *dspev*

Arrays:

*w(\*)*, **DIMENSION** at least  $\max(1, n)$ .

If *info* = 0, *w* contains the eigenvalues of the matrix *A* in ascending order.

*z(ldz, \*)*. The second dimension of *z* must be at least  $\max(1, n)$ .

If *jobz* = 'V', then if *info* = 0, *z* contains the orthonormal eigenvectors of the matrix *A*, with the *i*-th column of *z* holding the eigenvector associated with *w(i)*.

If *jobz* = 'N', then *z* is not referenced.

*ap*

On exit, this array is overwritten by the values generated during the reduction to tridiagonal form. The elements of the diagonal and the off-diagonal of the tridiagonal matrix overwrite the corresponding elements of *A*.

*info*

**INTEGER.**

If *info* = 0, the execution is successful.

If *info* = *-i*, the *i*th parameter had an illegal value.

If *info* = *i*, then the algorithm failed to converge; *i* indicates the number of elements of an intermediate tridiagonal form which did not converge to zero.

---

## ?hpev

*Computes all eigenvalues and,  
optionally, eigenvectors of a Hermitian  
matrix in packed storage.*

---

```
call chpev (jobz, uplo, n, ap, w, z, ldz, work, rwork, info)
call zhpev (jobz, uplo, n, ap, w, z, ldz, work, rwork, info)
```

### Discussion

This routine computes all the eigenvalues and, optionally, eigenvectors of a complex Hermitian matrix  $A$  in packed storage.

### Input Parameters

<i>jobz</i>	CHARACTER*1. Must be ' <b>N</b> ' or ' <b>V</b> '. If <i>job</i> = ' <b>N</b> ', then only eigenvalues are computed. If <i>job</i> = ' <b>V</b> ', then eigenvalues and eigenvectors are computed.
<i>uplo</i>	CHARACTER*1. Must be ' <b>U</b> ' or ' <b>L</b> '. If <i>uplo</i> = ' <b>U</b> ', <i>ap</i> stores the packed upper triangular part of $A$ . If <i>uplo</i> = ' <b>L</b> ', <i>ap</i> stores the packed lower triangular part of $A$ .
<i>n</i>	INTEGER. The order of the matrix $A$ ( $n \geq 0$ ).
<i>ap, work</i>	COMPLEX for <b>chpev</b> DOUBLE COMPLEX for <b>zhpev</b> . Arrays: <i>ap</i> (*) contains the packed upper or lower triangle of Hermitian matrix $A$ , as specified by <i>uplo</i> . The dimension of <i>ap</i> must be at least $\max(1, n*(n+1)/2)$ . <i>work</i> (*) is a workspace array, DIMENSION at least $\max(1, 2n-1)$ .

---

<i>ldz</i>	<b>INTEGER.</b> The leading dimension of the output array <i>z</i> . Constraints: if <i>jobz</i> = 'N', then <i>ldz</i> $\geq 1$ ; if <i>jobz</i> = 'V', then <i>ldz</i> $\geq \max(1, n)$ .
<i>rwork</i>	<b>REAL</b> for <i>chpev</i> <b>DOUBLE PRECISION</b> for <i>zhpev</i> . Workspace array, <b>DIMENSION</b> at least $\max(1, 3n-2)$ .

## Output Parameters

<i>w</i>	<b>REAL</b> for <i>chpev</i> <b>DOUBLE PRECISION</b> for <i>zhpev</i> . Array, <b>DIMENSION</b> at least $\max(1, n)$ . If <i>info</i> = 0, <i>w</i> contains the eigenvalues of the matrix <i>A</i> in ascending order.
<i>z</i>	<b>COMPLEX</b> for <i>chpev</i> <b>DOUBLE COMPLEX</b> for <i>zhpev</i> . Array <i>z</i> ( <i>ldz</i> , *). The second dimension of <i>z</i> must be at least $\max(1, n)$ . If <i>jobz</i> = 'V', then if <i>info</i> = 0, <i>z</i> contains the orthonormal eigenvectors of the matrix <i>A</i> , with the <i>i</i> -th column of <i>z</i> holding the eigenvector associated with <i>w(i)</i> . If <i>jobz</i> = 'N', then <i>z</i> is not referenced.
<i>ap</i>	On exit, this array is overwritten by the values generated during the reduction to tridiagonal form. The elements of the diagonal and the off-diagonal of the tridiagonal matrix overwrite the corresponding elements of <i>A</i> .
<i>info</i>	<b>INTEGER.</b> If <i>info</i> = 0, the execution is successful. If <i>info</i> = <i>-i</i> , the <i>i</i> th parameter had an illegal value. If <i>info</i> = <i>i</i> , then the algorithm failed to converge; <i>i</i> indicates the number of elements of an intermediate tridiagonal form which did not converge to zero.

---

## ?spevd

*Uses divide and conquer algorithm to compute all eigenvalues and (optionally) all eigenvectors of a real symmetric matrix held in packed storage.*

---

```
call sspevd ( job, uplo, n, ap, w, z, ldz, work, lwork, iwork, liwork, info)
call dspevd ( job, uplo, n, ap, w, z, ldz, work, lwork, iwork, liwork, info)
```

### Discussion

This routine computes all the eigenvalues, and optionally all the eigenvectors, of a real symmetric matrix  $A$  (held in packed storage). In other words, it can compute the spectral factorization of  $A$  as:  $A = Z\Lambda Z^T$ . Here  $\Lambda$  is a diagonal matrix whose diagonal elements are the eigenvalues  $\lambda_i$ , and  $Z$  is the orthogonal matrix whose columns are the eigenvectors  $z_i$ . Thus,

$$Az_i = \lambda_i z_i \text{ for } i = 1, 2, \dots, n.$$

If the eigenvectors are requested, then this routine uses a divide and conquer algorithm to compute eigenvalues and eigenvectors. However, if only eigenvalues are required, then it uses the Pal-Walker-Kahan variant of the  $QL$  or  $QR$  algorithm.

### Input Parameters

<i>job</i>	CHARACTER*1. Must be ' <b>N</b> ' or ' <b>V</b> '. If <i>job</i> = ' <b>N</b> ', then only eigenvalues are computed. If <i>job</i> = ' <b>V</b> ', then eigenvalues and eigenvectors are computed.
<i>uplo</i>	CHARACTER*1. Must be ' <b>U</b> ' or ' <b>L</b> '. If <i>uplo</i> = ' <b>U</b> ', <i>ap</i> stores the packed upper triangular part of $A$ . If <i>uplo</i> = ' <b>L</b> ', <i>ap</i> stores the packed lower triangular part of $A$ .
<i>n</i>	INTEGER. The order of the matrix $A$ ( <i>n</i> $\geq 0$ ).

---

<i>ap, work</i>	<b>REAL</b> for <b>sspevd</b> <b>DOUBLE PRECISION</b> for <b>dspevd</b>
Arrays:	
<i>ap</i> (*)	contains the packed upper or lower triangle of symmetric matrix A, as specified by <i>uplo</i> . The dimension of <i>ap</i> must be at least $\max(1, n*(n+1)/2)$
<i>work</i> (*) is a workspace array, <b>DIMENSION</b> at least <i>lwork</i> .	
<i>ldz</i>	<b>INTEGER</b> . The leading dimension of the output array <i>z</i> .
Constraints:	
	if <i>job</i> = 'N', then <i>ldz</i> $\geq 1$ ;
	if <i>job</i> = 'V', then <i>ldz</i> $\geq \max(1, n)$ .
<i>lwork</i>	<b>INTEGER</b> . The dimension of the array <i>work</i> .
Constraints:	
	if <i>n</i> $\leq 1$ , then <i>lwork</i> $\geq 1$ ;
	if <i>job</i> = 'N' and <i>n</i> $> 1$ , then <i>lwork</i> $\geq 2n$ ;
	if <i>job</i> = 'V' and <i>n</i> $> 1$ , then <i>lwork</i> $\geq 2n^2 + (5+2k)*n+1$ , where <i>k</i> is the smallest integer which satisfies $2^k \geq n$ .
<i>iwork</i>	<b>INTEGER</b> .
Workspace array, <b>DIMENSION</b> at least <i>liwork</i> .	
<i>liwork</i>	<b>INTEGER</b> . The dimension of the array <i>iwork</i> .
Constraints:	
	if <i>n</i> $\leq 1$ , then <i>liwork</i> $\geq 1$ ;
	if <i>job</i> = 'N' and <i>n</i> $> 1$ , then <i>liwork</i> $\geq 1$ ;
	if <i>job</i> = 'V' and <i>n</i> $> 1$ , then <i>liwork</i> $\geq 5n+2$ .

## Output Parameters

<i>w, z</i>	<b>REAL</b> for <b>sspevd</b> <b>DOUBLE PRECISION</b> for <b>dspevd</b>
Arrays:	
<i>w</i> (*), <b>DIMENSION</b> at least $\max(1, n)$ .	
If <i>info</i> = 0, contains the eigenvalues of the matrix A in ascending order. See also <i>info</i> .	
<i>z</i> ( <i>ldz</i> , *).	The second dimension of <i>z</i> must be: at least 1 if <i>job</i> = 'N';

at least  $\max(1, n)$  if  $job = 'V'$ .

If  $job = 'V'$ , then this array is overwritten by the orthogonal matrix  $Z$  which contains the eigenvectors of  $A$ . If  $job = 'N'$ , then  $z$  is not referenced.

*ap*

On exit, this array is overwritten by the values generated during the reduction to tridiagonal form. The elements of the diagonal and the off-diagonal of the tridiagonal matrix overwrite the corresponding elements of  $A$ .

*work(1)*

On exit, if  $lwork > 0$ , then *work(1)* returns the required minimal size of *lwork*.

*iwork(1)*

On exit, if  $liwork > 0$ , then *iwork(1)* returns the required minimal size of *liwork*.

*info*

INTEGER.

If *info* = 0, the execution is successful.

If *info* = *i*, then the algorithm failed to converge; *i* indicates the number of elements of an intermediate tridiagonal form which did not converge to zero.

If *info* = -*i*, the *i*th parameter had an illegal value.

## Application Notes

The computed eigenvalues and eigenvectors are exact for a matrix  $T + E$  such that  $\|E\|_2 = O(\epsilon) \|T\|_2$ , where  $\epsilon$  is the machine precision.

The complex analogue of this routine is [?hpevd](#).

See also [?syevd](#) for matrices held in full storage, and [?sbevd](#) for banded matrices.

## ?hpevd

*Uses divide and conquer algorithm to compute all eigenvalues and (optionally) all eigenvectors of a complex Hermitian matrix held in packed storage.*

```
call chpevd (job, uplo, n, ap, w, z, ldz, work, lwork, rwork,
             lrwork, iwork, liwork, info)
call zhpevd (job, uplo, n, ap, w, z, ldz, work, lwork, rwork,
             lrwork, iwork, liwork, info)
```

### Discussion

This routine computes all the eigenvalues, and optionally all the eigenvectors, of a complex Hermitian matrix  $A$  (held in packed storage). In other words, it can compute the spectral factorization of  $A$  as:  $A = Z\Lambda Z^H$ . Here  $\Lambda$  is a real diagonal matrix whose diagonal elements are the eigenvalues  $\lambda_i$ , and  $Z$  is the (complex) unitary matrix whose columns are the eigenvectors  $z_i$ . Thus,

$$Az_i = \lambda_i z_i \text{ for } i = 1, 2, \dots, n.$$

If the eigenvectors are requested, then this routine uses a divide and conquer algorithm to compute eigenvalues and eigenvectors. However, if only eigenvalues are required, then it uses the Pal-Walker-Kahan variant of the  $QL$  or  $QR$  algorithm.

### Input Parameters

<i>job</i>	CHARACTER*1. Must be ' <b>N</b> ' or ' <b>V</b> '. If <i>job</i> = ' <b>N</b> ', then only eigenvalues are computed. If <i>job</i> = ' <b>V</b> ', then eigenvalues and eigenvectors are computed.
<i>uplo</i>	CHARACTER*1. Must be ' <b>U</b> ' or ' <b>L</b> '.

	If $\text{uplo} = \text{'U'}$ , $\text{ap}$ stores the packed upper triangular part of $A$ . If $\text{uplo} = \text{'L'}$ , $\text{ap}$ stores the packed lower triangular part of $A$ .
$n$	<b>INTEGER.</b> The order of the matrix $A$ ( $n \geq 0$ ).
$\text{ap}, \text{work}$	<b>COMPLEX</b> for <code>chpevd</code> <b>DOUBLE COMPLEX</b> for <code>zhpevd</code> Arrays: $\text{ap}(\star)$ contains the packed upper or lower triangle of Hermitian matrix $A$ , as specified by $\text{uplo}$ . The dimension of $\text{ap}$ must be at least $\max(1, n^*(n+1)/2)$ $\text{work}(\star)$ is a workspace array, <b>DIMENSION</b> at least $\text{lwork}$ .
$\text{ldz}$	<b>INTEGER.</b> The leading dimension of the output array $\text{z}$ . Constraints: if $\text{job} = \text{'N'}$ , then $\text{ldz} \geq 1$ ; if $\text{job} = \text{'V'}$ , then $\text{ldz} \geq \max(1, n)$ .
$\text{lwork}$	<b>INTEGER.</b> The dimension of the array $\text{work}$ . Constraints: if $n \leq 1$ , then $\text{lwork} \geq 1$ ; if $\text{job} = \text{'N'}$ and $n > 1$ , then $\text{lwork} \geq n$ ; if $\text{job} = \text{'V'}$ and $n > 1$ , then $\text{lwork} \geq 2n$
$\text{rwork}$	<b>REAL</b> for <code>chpevd</code> <b>DOUBLE PRECISION</b> for <code>zhpevd</code> Workspace array, <b>DIMENSION</b> at least $\text{lrwork}$ .
$\text{lrwork}$	<b>INTEGER.</b> The dimension of the array $\text{rwork}$ . Constraints: if $n \leq 1$ , then $\text{lrwork} \geq 1$ ; if $\text{job} = \text{'N'}$ and $n > 1$ , then $\text{lrwork} \geq n$ ; if $\text{job} = \text{'V'}$ and $n > 1$ , then $\text{lrwork} \geq 3n^2 + (4+2k)*n+1$ , where $k$ is the smallest integer which satisfies $2^k \geq n$ .
$\text{iwork}$	<b>INTEGER.</b> Workspace array, <b>DIMENSION</b> at least $\text{liwork}$ .

*lwork*      **INTEGER.** The dimension of the array *iwork*.  
 Constraints:  
 if *n* ≤ 1, then *lwork* ≥ 1;  
 if *job* = 'N' and *n* > 1, then *lwork* ≥ 1;  
 if *job* = 'V' and *n* > 1, then *lwork* ≥ 5*n*+2.

## Output Parameters

<i>w</i>	<b>REAL</b> for <i>chpevd</i> <b>DOUBLE PRECISION</b> for <i>zhpevd</i> Array, <b>DIMENSION</b> at least max(1, <i>n</i> ). If <i>info</i> = 0, contains the eigenvalues of the matrix <i>A</i> in ascending order. See also <i>info</i> .
<i>z</i>	<b>COMPLEX</b> for <i>chpevd</i> <b>DOUBLE COMPLEX</b> for <i>zhpevd</i> Array, <b>DIMENSION</b> ( <i>ldz</i> , *). The second dimension of <i>z</i> must be: at least 1 if <i>job</i> = 'N'; at least max(1, <i>n</i> ) if <i>job</i> = 'V'. If <i>job</i> = 'V', then this array is overwritten by the unitary matrix <i>Z</i> which contains the eigenvectors of <i>A</i> . If <i>job</i> = 'N', then <i>z</i> is not referenced.
<i>ap</i>	On exit, this array is overwritten by the values generated during the reduction to tridiagonal form. The elements of the diagonal and the off-diagonal of the tridiagonal matrix overwrite the corresponding elements of <i>A</i> .
<i>work(1)</i>	On exit, if <i>lwork</i> > 0, then the real part of <i>work(1)</i> returns the required minimal size of <i>lwork</i> .
<i>rwork(1)</i>	On exit, if <i>lrwork</i> > 0, then <i>rwork(1)</i> returns the required minimal size of <i>lrwork</i> .
<i>iwork(1)</i>	On exit, if <i>liwork</i> > 0, then <i>iwork(1)</i> returns the required minimal size of <i>liwork</i> .
<i>info</i>	<b>INTEGER.</b> If <i>info</i> = 0, the execution is successful. If <i>info</i> = <i>i</i> , then the algorithm failed to converge; <i>i</i>

indicates the number of elements of an intermediate tridiagonal form which did not converge to zero.

If  $\text{info} = -i$ , the  $i$ th parameter had an illegal value.

### Application Notes

The computed eigenvalues and eigenvectors are exact for a matrix  $T + E$  such that  $\|E\|_2 = O(\epsilon) \|T\|_2$ , where  $\epsilon$  is the machine precision.

The real analogue of this routine is [?spevd](#).

See also [?heevd](#) for matrices held in full storage, and [?hbevd](#) for banded matrices.

## ?spevx

*Computes selected eigenvalues and, optionally, eigenvectors of a real symmetric matrix in packed storage.*

```
call sspevx (jobz, range, uplo, n, ap, vl, vu, il, iu, abstol,
             m, w, z, ldz, work, iwork, ifail, info)
call dspevx (jobz, range, uplo, n, ap, vl, vu, il, iu, abstol,
             m, w, z, ldz, work, iwork, ifail, info)
```

### Discussion

This routine computes selected eigenvalues and, optionally, eigenvectors of a real symmetric matrix  $A$  in packed storage. Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

### Input Parameters

<i>jobz</i>	CHARACTER*1. Must be ' <b>N</b> ' or ' <b>V</b> '. If <i>job</i> = ' <b>N</b> ', then only eigenvalues are computed. If <i>job</i> = ' <b>V</b> ', then eigenvalues and eigenvectors are computed.
<i>range</i>	CHARACTER*1. Must be ' <b>A</b> ' or ' <b>V</b> ' or ' <b>I</b> '. If <i>range</i> = ' <b>A</b> ', the routine computes all eigenvalues. If <i>range</i> = ' <b>V</b> ', the routine computes eigenvalues $\lambda_i$ in the half-open interval: $vl < \lambda_i \leq vu$ . If <i>range</i> = ' <b>I</b> ', the routine computes eigenvalues with indices <i>il</i> to <i>iu</i> .
<i>uplo</i>	CHARACTER*1. Must be ' <b>U</b> ' or ' <b>L</b> '. If <i>uplo</i> = ' <b>U</b> ', <i>ap</i> stores the packed upper triangular part of $A$ . If <i>uplo</i> = ' <b>L</b> ', <i>ap</i> stores the packed lower triangular part of $A$ .
<i>n</i>	INTEGER. The order of the matrix $A$ ( <i>n</i> $\geq 0$ ).

<i>ap, work</i>	<small>REAL for sspevx DOUBLE PRECISION for dspevx</small>
	<small>Arrays: <i>ap(*)</i> contains the packed upper or lower triangle of the symmetric matrix <math>A</math>, as specified by <i>uplo</i>. The dimension of <i>ap</i> must be at least <math>\max(1, n*(n+1)/2)</math>.</small>
	<small><i>work(*)</i> is a workspace array, <b>DIMENSION</b> at least <math>\max(1, 8n)</math>.</small>
<i>vl, vu</i>	<small>REAL for sspevx DOUBLE PRECISION for dspevx</small>
	<small>If <i>range</i> = 'V', the lower and upper bounds of the interval to be searched for eigenvalues. Constraint: <i>vl</i> &lt; <i>vu</i>. If <i>range</i> = 'A' or 'I', <i>vl</i> and <i>vu</i> are not referenced.</small>
<i>il, iu</i>	<small>INTEGER. If <i>range</i> = 'I', the indices in ascending order of the smallest and largest eigenvalues to be returned. Constraint: <math>1 \leq il \leq iu \leq n</math>, if <math>n &gt; 0</math>; <i>il</i>=1 and <i>iu</i>=0 if <math>n = 0</math>. If <i>range</i> = 'A' or 'V', <i>il</i> and <i>iu</i> are not referenced.</small>
<i>abstol</i>	<small>REAL for sspevx DOUBLE PRECISION for dspevx</small>
	<small>The absolute error tolerance to which each eigenvalue is required. See <i>Application notes</i> for details on error tolerance.</small>
<i>ldz</i>	<small>INTEGER. The leading dimension of the output array <i>z</i>. Constraints: if <i>jobz</i> = 'N', then <i>ldz</i> <math>\geq 1</math>; if <i>jobz</i> = 'V', then <i>ldz</i> <math>\geq \max(1, n)</math>.</small>
<i>iwork</i>	<small>INTEGER. Workspace array, <b>DIMENSION</b> at least <math>\max(1, 5n)</math>.</small>

## Output Parameters

*ap*

On exit, this array is overwritten by the values generated during the reduction to tridiagonal form. The elements of the diagonal and the off-diagonal of the tridiagonal matrix overwrite the corresponding elements of A.

*m*

**INTEGER**. The total number of eigenvalues found,  
 $0 \leq m \leq n$ . If *range* = 'A', *m* = *n*, and if *range* = 'I',  
 $m = iu - il + 1$ .

*w, z*

**REAL** for **sspevx**  
**DOUBLE PRECISION** for **dspevx**

Arrays:

*w(\*)*, **DIMENSION** at least  $\max(1, n)$ .

If *info* = 0, contains the selected eigenvalues of the matrix A in ascending order.

*z(ldz, \*)*. The second dimension of *z* must be at least  $\max(1, m)$ .

If *jobz* = 'V', then if *info* = 0, the first *m* columns of *z* contain the orthonormal eigenvectors of the matrix A corresponding to the selected eigenvalues, with the *i*-th column of *z* holding the eigenvector associated with *w(i)*. If an eigenvector fails to converge, then that column of *z* contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in *ifail*.

If *jobz* = 'N', then *z* is not referenced.

Note: you must ensure that at least  $\max(1, m)$  columns are supplied in the array *z*; if *range* = 'V', the exact value of *m* is not known in advance and an upper bound must be used.

*ifail*

**INTEGER**. Array, **DIMENSION** at least  $\max(1, n)$ .

If *jobz* = 'V', then if *info* = 0, the first *m* elements of *ifail* are zero; if *info* > 0, the *ifail* contains the indices the eigenvectors that failed to converge.

If *jobz* = 'N', then *ifail* is not referenced.

*info**INTEGER.*If *info* = 0, the execution is successful.If *info* = -*i*, the *i*th parameter had an illegal value.If *info* = *i*, then *i* eigenvectors failed to converge; their indices are stored in the array *ifail*.

## Application Notes

An approximate eigenvalue is accepted as converged when it is determined to lie in an interval [a,b] of width less than or equal to

*abstol* +  $\epsilon * \max(|a|, |b|)$ , where  $\epsilon$  is the machine precision. If *abstol* is less than or equal to zero, then  $\epsilon * \|T\|_1$  will be used in its place, where *T* is the tridiagonal matrix obtained by reducing *A* to tridiagonal form.

Eigenvalues will be computed most accurately when *abstol* is set to twice the underflow threshold  $2 * ?lamch('S')$ , not zero. If this routine returns with *info* > 0, indicating that some eigenvectors did not converge, try setting *abstol* to  $2 * ?lamch('S')$ .

## ?hpevx

*Computes selected eigenvalues and, optionally, eigenvectors of a Hermitian matrix in packed storage.*

```
call chpevx (jobz, range, uplo, n, ap, vl, vu, il, iu, abstol,
             m, w, z, ldz, work, rwork, iwork, ifail, info)
call zhpevx (jobz, range, uplo, n, ap, vl, vu, il, iu, abstol,
             m, w, z, ldz, work, rwork, iwork, ifail, info)
```

### Discussion

This routine computes selected eigenvalues and, optionally, eigenvectors of a complex Hermitian matrix  $A$  in packed storage. Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

### Input Parameters

<i>jobz</i>	CHARACTER*1. Must be ' <b>N</b> ' or ' <b>V</b> '. If <i>job</i> = ' <b>N</b> ', then only eigenvalues are computed. If <i>job</i> = ' <b>V</b> ', then eigenvalues and eigenvectors are computed.
<i>range</i>	CHARACTER*1. Must be ' <b>A</b> ' or ' <b>V</b> ' or ' <b>I</b> '. If <i>range</i> = ' <b>A</b> ', the routine computes all eigenvalues. If <i>range</i> = ' <b>V</b> ', the routine computes eigenvalues $\lambda_i$ in the half-open interval: $vl < \lambda_i \leq vu$ . If <i>range</i> = ' <b>I</b> ', the routine computes eigenvalues with indices <i>il</i> to <i>iu</i> .
<i>uplo</i>	CHARACTER*1. Must be ' <b>U</b> ' or ' <b>L</b> '. If <i>uplo</i> = ' <b>U</b> ', <i>ap</i> stores the packed upper triangular part of $A$ . If <i>uplo</i> = ' <b>L</b> ', <i>ap</i> stores the packed lower triangular part of $A$ .
<i>n</i>	INTEGER. The order of the matrix $A$ ( <i>n</i> $\geq 0$ ).

<i>ap</i> , <i>work</i>	COMPLEX for <code>chpevx</code> DOUBLE COMPLEX for <code>zhpevx</code>
	Arrays: <i>ap</i> (*) contains the packed upper or lower triangle of the Hermitian matrix <i>A</i> , as specified by <i>uplo</i> . The dimension of <i>ap</i> must be at least $\max(1, n*(n+1)/2)$ .
	<i>work</i> (*) is a workspace array, DIMENSION at least $\max(1, 2n)$ .
<i>vl</i> , <i>vu</i>	REAL for <code>chpevx</code> DOUBLE PRECISION for <code>zhpevx</code>
	If <i>range</i> = 'V', the lower and upper bounds of the interval to be searched for eigenvalues. Constraint: <i>vl</i> < <i>vu</i> .
	If <i>range</i> = 'A' or 'I', <i>vl</i> and <i>vu</i> are not referenced.
<i>il</i> , <i>iu</i>	INTEGER. If <i>range</i> = 'I', the indices in ascending order of the smallest and largest eigenvalues to be returned. Constraint: $1 \leq il \leq iu \leq n$ , if <i>n</i> > 0; <i>il</i> =1 and <i>iu</i> =0 if <i>n</i> = 0. If <i>range</i> = 'A' or 'V', <i>il</i> and <i>iu</i> are not referenced.
<i>abstol</i>	REAL for <code>chpevx</code> DOUBLE PRECISION for <code>zhpevx</code>
	The absolute error tolerance to which each eigenvalue is required. See <i>Application notes</i> for details on error tolerance.
<i>ldz</i>	INTEGER. The leading dimension of the output array <i>z</i> . Constraints: if <i>jobz</i> = 'N', then <i>ldz</i> $\geq 1$ ; if <i>jobz</i> = 'V', then <i>ldz</i> $\geq \max(1, n)$ .
<i>rwork</i>	REAL for <code>chpevx</code> DOUBLE PRECISION for <code>zhpevx</code>
	Workspace array, DIMENSION at least $\max(1, 7n)$ .
<i>iwork</i>	INTEGER. Workspace array, DIMENSION at least $\max(1, 5n)$ .

## Output Parameters

*ap*

On exit, this array is overwritten by the values generated during the reduction to tridiagonal form. The elements of the diagonal and the off-diagonal of the tridiagonal matrix overwrite the corresponding elements of A.

*m*

**INTEGER.** The total number of eigenvalues found,  
 $0 \leq m \leq n$ . If *range* = 'A', *m* = *n*, and if *range* = 'I',  
 $m = iu - il + 1$ .

*w*

**REAL** for *chpevx*  
**DOUBLE PRECISION** for *zhpevx*  
 Array, **DIMENSION** at least max(1, *n*). If *info* = 0,  
 contains the selected eigenvalues of the matrix A in  
 ascending order.

*z*

**COMPLEX** for *chpevx*  
**DOUBLE COMPLEX** for *zhpevx*  
 Array *z*(*ldz*, \*). The second dimension of *z* must be  
 at least max(1, *m*).  
 If *jobz* = 'V', then if *info* = 0, the first *m* columns of *z*  
 contain the orthonormal eigenvectors of the matrix A  
 corresponding to the selected eigenvalues, with the *i*-th  
 column of *z* holding the eigenvector associated with  
 $w(i)$ . If an eigenvector fails to converge, then that  
 column of *z* contains the latest approximation to the  
 eigenvector, and the index of the eigenvector is returned  
 in *ifail*.  
 If *jobz* = 'N', then *z* is not referenced.

Note: you must ensure that at least max(1, *m*) columns  
 are supplied in the array *z*; if *range* = 'V', the exact  
 value of *m* is not known in advance and an upper bound  
 must be used.

*ifail*

**INTEGER.** Array, **DIMENSION** at least max(1, *n*).  
 If *jobz* = 'V', then if *info* = 0, the first *m* elements of  
*ifail* are zero; if *info* > 0, the *ifail* contains the  
 indices of the eigenvectors that failed to converge.  
 If *jobz* = 'N', then *ifail* is not referenced.

*info**INTEGER.*If *info* = 0, the execution is successful.If *info* = -*i*, the *i*th parameter had an illegal value.If *info* = *i*, then *i* eigenvectors failed to converge; their indices are stored in the array *ifail*.

## Application Notes

An approximate eigenvalue is accepted as converged when it is determined to lie in an interval [a,b] of width less than or equal to

*abstol* +  $\epsilon * \max(|a|, |b|)$ , where  $\epsilon$  is the machine precision. If *abstol* is less than or equal to zero, then  $\epsilon * \|T\|_1$  will be used in its place, where *T* is the tridiagonal matrix obtained by reducing *A* to tridiagonal form.

Eigenvalues will be computed most accurately when *abstol* is set to twice the underflow threshold  $2 * ?lamch('S')$ , not zero. If this routine returns with *info* > 0, indicating that some eigenvectors did not converge, try setting *abstol* to  $2 * ?lamch('S')$ .

## ?sbev

*Computes all eigenvalues and, optionally, eigenvectors of a real symmetric band matrix.*

```
call ssbev (jobz, uplo, n, kd, ab, ldab, w, z, ldz, work, info)
call dsbev (jobz, uplo, n, kd, ab, ldab, w, z, ldz, work, info)
```

### Discussion

This routine computes all eigenvalues and, optionally, eigenvectors of a real symmetric band matrix  $A$ .

### Input Parameters

<i>jobz</i>	CHARACTER*1. Must be ' <b>N</b> ' or ' <b>V</b> '. If <i>jobz</i> = ' <b>N</b> ', then only eigenvalues are computed. If <i>jobz</i> = ' <b>V</b> ', then eigenvalues and eigenvectors are computed.
<i>uplo</i>	CHARACTER*1. Must be ' <b>U</b> ' or ' <b>L</b> '. If <i>uplo</i> = ' <b>U</b> ', <i>ab</i> stores the upper triangular part of $A$ . If <i>uplo</i> = ' <b>L</b> ', <i>ab</i> stores the lower triangular part of $A$ .
<i>n</i>	INTEGER. The order of the matrix $A$ ( <i>n</i> $\geq$ 0).
<i>kd</i>	INTEGER. The number of super- or sub-diagonals in $A$ ( <i>kd</i> $\geq$ 0).
<i>ab</i> , <i>work</i>	REAL for <b>ssbev</b> DOUBLE PRECISION for <b>dsbev</b> . Arrays: <i>ab</i> ( <i>ldab</i> , *) is an array containing either upper or lower triangular part of the symmetric matrix $A$ (as specified by <i>uplo</i> ) in band storage format. The second dimension of <i>ab</i> must be at least $\max(1, n)$ . <i>work</i> (*) is a workspace array. The dimension of <i>work</i> must be at least $\max(1, 3n-2)$ .

<i>ldab</i>	<b>INTEGER.</b> The leading dimension of <i>ab</i> ; must be at least <i>kd</i> +1.
<i>ldz</i>	<b>INTEGER.</b> The leading dimension of the output array <i>z</i> . Constraints: if <i>jobz</i> = 'N', then <i>ldz</i> $\geq 1$ ; if <i>jobz</i> = 'V', then <i>ldz</i> $\geq \max(1, n)$ .

## Output Parameters

<i>w, z</i>	<b>REAL</b> for <i>ssbev</i> <b>DOUBLE PRECISION</b> for <i>dsbev</i> Arrays: <i>w(*)</i> , <b>DIMENSION</b> at least $\max(1, n)$ . If <i>info</i> = 0, contains the eigenvalues of the matrix <i>A</i> in ascending order. <i>z (ldz, *)</i> . The second dimension of <i>z</i> must be at least $\max(1, n)$ . If <i>jobz</i> = 'V', then if <i>info</i> = 0, <i>z</i> contains the orthonormal eigenvectors of the matrix <i>A</i> , with the <i>i</i> -th column of <i>z</i> holding the eigenvector associated with <i>w(i)</i> . If <i>jobz</i> = 'N', then <i>z</i> is not referenced.
<i>ab</i>	On exit, this array is overwritten by the values generated during the reduction to tridiagonal form. If <i>uplo</i> = 'U', the first superdiagonal and the diagonal of the tridiagonal matrix <i>T</i> are returned in rows <i>kd</i> and <i>kd</i> +1 of <i>ab</i> , and if <i>uplo</i> = 'L', the diagonal and first subdiagonal of <i>T</i> are returned in the first two rows of <i>ab</i> .
<i>info</i>	<b>INTEGER.</b> If <i>info</i> = 0, the execution is successful. If <i>info</i> = <i>-i</i> , the <i>i</i> th parameter had an illegal value. If <i>info</i> = <i>i</i> , then the algorithm failed to converge; <i>i</i> indicates the number of elements of an intermediate tridiagonal form which did not converge to zero.

## ?hbev

*Computes all eigenvalues and, optionally, eigenvectors of a Hermitian band matrix.*

```
call chbev(jobz, uplo, n, kd, ab, ldab, w, z, ldz, work, rwork, info)
call zhbev(jobz, uplo, n, kd, ab, ldab, w, z, ldz, work, rwork, info)
```

### Discussion

This routine computes all eigenvalues and, optionally, eigenvectors of a complex Hermitian band matrix  $A$ .

### Input Parameters

<i>jobz</i>	CHARACTER*1. Must be ' <b>N</b> ' or ' <b>V</b> '. If <i>jobz</i> = ' <b>N</b> ', then only eigenvalues are computed. If <i>jobz</i> = ' <b>V</b> ', then eigenvalues and eigenvectors are computed.
<i>uplo</i>	CHARACTER*1. Must be ' <b>U</b> ' or ' <b>L</b> '. If <i>uplo</i> = ' <b>U</b> ', <i>ab</i> stores the upper triangular part of $A$ . If <i>uplo</i> = ' <b>L</b> ', <i>ab</i> stores the lower triangular part of $A$ .
<i>n</i>	INTEGER. The order of the matrix $A$ ( <i>n</i> $\geq$ 0).
<i>kd</i>	INTEGER. The number of super- or sub-diagonals in $A$ ( <i>kd</i> $\geq$ 0).
<i>ab</i> , <i>work</i>	COMPLEX for <b>chbev</b> DOUBLE COMPLEX for <b>zhbev</b> . Arrays: <i>ab</i> ( <i>ldab</i> , *) is an array containing either upper or lower triangular part of the Hermitian matrix $A$ (as specified by <i>uplo</i> ) in band storage format. The second dimension of <i>ab</i> must be at least $\max(1, n)$ . <i>work</i> (*) is a workspace array. The dimension of <i>work</i> must be at least $\max(1, n)$ .

<i>ldab</i>	<b>INTEGER.</b> The leading dimension of <i>ab</i> ; must be at least <i>kd</i> +1.
<i>ldz</i>	<b>INTEGER.</b> The leading dimension of the output array <i>z</i> . Constraints: if <i>jobz</i> = 'N', then <i>ldz</i> $\geq 1$ ; if <i>jobz</i> = 'V', then <i>ldz</i> $\geq \max(1, n)$ .
<i>rwork</i>	<b>REAL</b> for <i>chbev</i> <b>DOUBLE PRECISION</b> for <i>zhbev</i> Workspace array, <b>DIMENSION</b> at least $\max(1, 3n-2)$ .

## Output Parameters

<i>w</i>	<b>REAL</b> for <i>chbev</i> <b>DOUBLE PRECISION</b> for <i>zhbev</i> Array, <b>DIMENSION</b> at least $\max(1, n)$ . If <i>info</i> = 0, contains the eigenvalues in ascending order.
<i>z</i>	<b>COMPLEX</b> for <i>chbev</i> <b>DOUBLE COMPLEX</b> for <i>zhbev</i> . Array <i>z</i> ( <i>ldz</i> , *). The second dimension of <i>z</i> must be at least $\max(1, n)$ . If <i>jobz</i> = 'V', then if <i>info</i> = 0, <i>z</i> contains the orthonormal eigenvectors of the matrix <i>A</i> , with the <i>i</i> -th column of <i>z</i> holding the eigenvector associated with <i>w(i)</i> . If <i>jobz</i> = 'N', then <i>z</i> is not referenced.
<i>ab</i>	On exit, this array is overwritten by the values generated during the reduction to tridiagonal form. If <i>uplo</i> = 'U', the first superdiagonal and the diagonal of the tridiagonal matrix <i>T</i> are returned in rows <i>kd</i> and <i>kd</i> +1 of <i>ab</i> , and if <i>uplo</i> = 'L', the diagonal and first subdiagonal of <i>T</i> are returned in the first two rows of <i>ab</i> .
<i>info</i>	<b>INTEGER.</b> If <i>info</i> = 0, the execution is successful. If <i>info</i> = <i>-i</i> , the <i>i</i> th parameter had an illegal value. If <i>info</i> = <i>i</i> , then the algorithm failed to converge; <i>i</i> indicates the number of elements of an intermediate tridiagonal form which did not converge to zero.

## ?sbefd

*Computes all eigenvalues and (optionally) all eigenvectors of a real symmetric band matrix using divide and conquer algorithm.*

```
call ssbevd (job, uplo, n, kd, ab, ldab, w, z, ldz, work, lwork,
             iwork, liwork, info)
call dsbevd (job, uplo, n, kd, ab, ldab, w, z, ldz, work, lwork,
             iwork, liwork, info)
```

### Discussion

This routine computes all the eigenvalues, and optionally all the eigenvectors, of a real symmetric band matrix  $A$ . In other words, it can compute the spectral factorization of  $A$  as:

$$A = Z\Lambda Z^T$$

Here  $\Lambda$  is a diagonal matrix whose diagonal elements are the eigenvalues  $\lambda_i$ , and  $Z$  is the orthogonal matrix whose columns are the eigenvectors  $z_i$ . Thus,

$$Az_i = \lambda_i z_i \text{ for } i = 1, 2, \dots, n.$$

If the eigenvectors are requested, then this routine uses a divide and conquer algorithm to compute eigenvalues and eigenvectors. However, if only eigenvalues are required, then it uses the Pal-Walker-Kahan variant of the  $QL$  or  $QR$  algorithm.

### Input Parameters

<i>job</i>	CHARACTER*1. Must be ' <b>N</b> ' or ' <b>V</b> '. If <i>job</i> = ' <b>N</b> ', then only eigenvalues are computed. If <i>job</i> = ' <b>V</b> ', then eigenvalues and eigenvectors are computed.
<i>uplo</i>	CHARACTER*1. Must be ' <b>U</b> ' or ' <b>L</b> '. If <i>uplo</i> = ' <b>U</b> ', <i>ab</i> stores the upper triangular part of $A$ . If <i>uplo</i> = ' <b>L</b> ', <i>ab</i> stores the lower triangular part of $A$ .

<i>n</i>	<b>INTEGER.</b> The order of the matrix <i>A</i> ( <i>n</i> ≥ 0).
<i>kd</i>	<b>INTEGER.</b> The number of super- or sub-diagonals in <i>A</i> ( <i>kd</i> ≥ 0).
<i>ab, work</i>	<b>REAL</b> for <b>ssbevd</b> <b>DOUBLE PRECISION</b> for <b>dsbevd</b> . Arrays: <i>ab</i> ( <i>ldab</i> , *) is an array containing either upper or lower triangular part of the symmetric matrix <i>A</i> (as specified by <i>uplo</i> ) in band storage format. The second dimension of <i>ab</i> must be at least $\max(1, n)$ . <i>work</i> (*) is a workspace array. The dimension of <i>work</i> must be at least <i>lwork</i> .
<i>ldab</i>	<b>INTEGER.</b> The leading dimension of <i>ab</i> ; must be at least <i>kd</i> +1.
<i>ldz</i>	<b>INTEGER.</b> The leading dimension of the output array <i>z</i> . Constraints: if <i>job</i> = 'N', then <i>ldz</i> ≥ 1; if <i>job</i> = 'V', then <i>ldz</i> ≥ $\max(1, n)$ .
<i>lwork</i>	<b>INTEGER.</b> The dimension of the array <i>work</i> . Constraints: if <i>n</i> ≤ 1, then <i>lwork</i> ≥ 1; if <i>job</i> = 'N' and <i>n</i> > 1, then <i>lwork</i> ≥ $2n$ ; if <i>job</i> = 'V' and <i>n</i> > 1, then $iwork \geq 3n^2 + (4+2k) * n + 1$ , where <i>k</i> is the smallest integer which satisfies $2^k \geq n$ .
<i>iwork</i>	<b>INTEGER.</b> Workspace array, <b>DIMENSION</b> at least <i>liwork</i> .
<i>liwork</i>	<b>INTEGER.</b> The dimension of the array <i>iwork</i> . Constraints: if <i>n</i> ≤ 1, then <i>liwork</i> ≥ 1; if <i>job</i> = 'N' and <i>n</i> > 1, then <i>liwork</i> ≥ 1; if <i>job</i> = 'V' and <i>n</i> > 1, then <i>liwork</i> ≥ $5n + 2$ .

## Output Parameters

<i>w, z</i>	<b>REAL</b> for <b>ssbevd</b> <b>DOUBLE PRECISION</b> for <b>dsbevd</b> Arrays: <i>w(*)</i> , DIMENSION at least max(1, <i>n</i> ). If <i>info</i> = 0, contains the eigenvalues of the matrix <i>A</i> in ascending order. See also <i>info</i> . <i>z(ldz, *)</i> . The second dimension of <i>z</i> must be: at least 1 if <i>job</i> = 'N'; at least max(1, <i>n</i> ) if <i>job</i> = 'V'. If <i>job</i> = 'V', then this array is overwritten by the orthogonal matrix <i>Z</i> which contains the eigenvectors of <i>A</i> . The <i>i</i> th column of <i>Z</i> contains the eigenvector which corresponds to the eigenvalue <i>w(i)</i> . If <i>job</i> = 'N', then <i>z</i> is not referenced.
<i>ab</i>	On exit, this array is overwritten by the values generated during the reduction to tridiagonal form.
<i>work(1)</i>	On exit, if <i>lwork</i> > 0, then <i>work(1)</i> returns the required minimal size of <i>lwork</i> .
<i>iwork(1)</i>	On exit, if <i>liwork</i> > 0, then <i>iwork(1)</i> returns the required minimal size of <i>liwork</i> .
<i>info</i>	<b>INTEGER</b> . If <i>info</i> = 0, the execution is successful. If <i>info</i> = <i>i</i> , then the algorithm failed to converge; <i>i</i> indicates the number of elements of an intermediate tridiagonal form which did not converge to zero. If <i>info</i> = - <i>i</i> , the <i>i</i> th parameter had an illegal value.

## Application Notes

The computed eigenvalues and eigenvectors are exact for a matrix  $T + E$  such that  $\|E\|_2 = O(\epsilon) \|T\|_2$ , where  $\epsilon$  is the machine precision.

The complex analogue of this routine is [?hbevd](#).

See also [?syevd](#) for matrices held in full storage, and [?spevd](#) for matrices held in packed storage.

## ?hbenvd

*Computes all eigenvalues and  
(optionally) all eigenvectors of a  
complex Hermitian band matrix using  
divide and conquer algorithm.*

---

```
call chbevd ( job, uplo, n, kd, ab, ldab, w, z, ldz, work, lwork,
              rwork, lrwork, iwork, liwork, info)
call zhbevd ( job, uplo, n, kd, ab, ldab, w, z, ldz, work, lwork,
              rwork, lrwork, iwork, liwork, info)
```

### Discussion

This routine computes all the eigenvalues, and optionally all the eigenvectors, of a complex Hermitian band matrix  $A$ . In other words, it can compute the spectral factorization of  $A$  as:  $A = Z\Lambda Z^H$ .

Here  $\Lambda$  is a real diagonal matrix whose diagonal elements are the eigenvalues  $\lambda_i$ , and  $Z$  is the (complex) unitary matrix whose columns are the eigenvectors  $z_i$ . Thus,

$$Az_i = \lambda_i z_i \text{ for } i = 1, 2, \dots, n.$$

If the eigenvectors are requested, then this routine uses a divide and conquer algorithm to compute eigenvalues and eigenvectors. However, if only eigenvalues are required, then it uses the Pal-Walker-Kahan variant of the  $QL$  or  $QR$  algorithm.

### Input Parameters

<i>job</i>	CHARACTER*1. Must be ' <b>N</b> ' or ' <b>V</b> '. If <i>job</i> = ' <b>N</b> ', then only eigenvalues are computed. If <i>job</i> = ' <b>V</b> ', then eigenvalues and eigenvectors are computed.
<i>uplo</i>	CHARACTER*1. Must be ' <b>U</b> ' or ' <b>L</b> '. If <i>uplo</i> = ' <b>U</b> ', <i>ab</i> stores the upper triangular part of $A$ . If <i>uplo</i> = ' <b>L</b> ', <i>ab</i> stores the lower triangular part of $A$ .
<i>n</i>	INTEGER. The order of the matrix $A$ ( <i>n</i> $\geq 0$ ).

<i>kd</i>	<b>INTEGER.</b> The number of super- or sub-diagonals in <i>A</i> ( <i>kd</i> ≥ 0).
<i>ab</i> , <i>work</i>	<b>COMPLEX</b> for <i>chbevd</i> <b>DOUBLE COMPLEX</b> for <i>zhbevd</i> . Arrays: <i>ab</i> ( <i>ldab</i> , *) is an array containing either upper or lower triangular part of the Hermitian matrix <i>A</i> (as specified by <i>uplo</i> ) in band storage format. The second dimension of <i>ab</i> must be at least <i>n</i> . <i>work</i> (*) is a workspace array. The dimension of <i>work</i> must be at least <i>lwork</i> .
<i>ldab</i>	<b>INTEGER.</b> The leading dimension of <i>ab</i> ; must be at least <i>kd</i> +1.
<i>ldz</i>	<b>INTEGER.</b> The leading dimension of the output array <i>z</i> . Constraints: if <i>job</i> = 'N', then <i>ldz</i> ≥ 1; if <i>job</i> = 'V', then <i>ldz</i> ≥ max(1, <i>n</i> ).
<i>lwork</i>	<b>INTEGER.</b> The dimension of the array <i>work</i> . Constraints: if <i>n</i> ≤ 1, then <i>lwork</i> ≥ 1; if <i>job</i> = 'N' and <i>n</i> > 1, then <i>lwork</i> ≥ <i>n</i> ; if <i>job</i> = 'V' and <i>n</i> > 1, then <i>lwork</i> ≥ $2n^2$
<i>rwork</i>	<b>REAL</b> for <i>chbevd</i> <b>DOUBLE PRECISION</b> for <i>zhbevd</i> Workspace array, <b>DIMENSION</b> at least <i>lrwork</i> .
<i>lrwork</i>	<b>INTEGER.</b> The dimension of the array <i>rwork</i> . Constraints: if <i>n</i> ≤ 1, then <i>lrwork</i> ≥ 1; if <i>job</i> = 'N' and <i>n</i> > 1, then <i>lrwork</i> ≥ <i>n</i> ; if <i>job</i> = 'V' and <i>n</i> > 1, then <i>lrwork</i> ≥ $3n^2 + (4+2k) * n + 1$ , where <i>k</i> is the smallest integer which satisfies $2^k \geq n$ .
<i>iwork</i>	<b>INTEGER.</b> Workspace array, <b>DIMENSION</b> at least <i>liwork</i> .

*liwork*      **INTEGER.** The dimension of the array *iwork*.

Constraints:

if *job* = 'N' or *n* ≤ 1, then *liwork* ≥ 1;

if *job* = 'V' and *n* > 1, then *liwork* ≥ 5*n*+2.

## Output Parameters

<i>w</i>	<b>REAL</b> for <i>chbevd</i> <b>DOUBLE PRECISION</b> for <i>zhbevd</i> Array, <b>DIMENSION</b> at least max(1, <i>n</i> ). If <i>info</i> = 0, contains the eigenvalues of the matrix <i>A</i> in ascending order. See also <i>info</i> .
<i>z</i>	<b>COMPLEX</b> for <i>chbevd</i> <b>DOUBLE COMPLEX</b> for <i>zhbevd</i> Array, <b>DIMENSION</b> ( <i>ldz</i> , *). The second dimension of <i>z</i> must be: at least 1 if <i>job</i> = 'N'; at least max(1, <i>n</i> ) if <i>job</i> = 'V'. If <i>job</i> = 'V', then this array is overwritten by the unitary matrix <i>Z</i> which contains the eigenvectors of <i>A</i> . The <i>i</i> th column of <i>Z</i> contains the eigenvector which corresponds to the eigenvalue <i>w(i)</i> . If <i>job</i> = 'N', then <i>z</i> is not referenced.
<i>ab</i>	On exit, this array is overwritten by the values generated during the reduction to tridiagonal form.
<i>work(1)</i>	On exit, if <i>lwork</i> > 0, then the real part of <i>work(1)</i> returns the required minimal size of <i>lwork</i> .
<i>rwork(1)</i>	On exit, if <i>lrwork</i> > 0, then <i>rwork(1)</i> returns the required minimal size of <i>lrwork</i> .
<i>iwork(1)</i>	On exit, if <i>liwork</i> > 0, then <i>iwork(1)</i> returns the required minimal size of <i>liwork</i> .
<i>info</i>	<b>INTEGER.</b> If <i>info</i> = 0, the execution is successful. If <i>info</i> = <i>i</i> , then the algorithm failed to converge; <i>i</i>

indicates the number of elements of an intermediate tridiagonal form which did not converge to zero.  
If `info` = `-i`, the `i`th parameter had an illegal value.

### Application Notes

The computed eigenvalues and eigenvectors are exact for a matrix  $T + E$  such that  $\|E\|_2 = O(\epsilon) \|T\|_2$ , where  $\epsilon$  is the machine precision.

The real analogue of this routine is [?sbevd](#).

See also [?heevd](#) for matrices held in full storage, and [?hpevd](#) for matrices held in packed storage.

## ?sbevx

*Computes selected eigenvalues and,  
optionally, eigenvectors of a real  
symmetric band matrix.*

---

```
call ssbevx ( jobz, range, uplo, n, kd, ab, ldab, q, ldq, vl, vu, il,
              iu, abstol, m, w, z, ldz, work, iwork, ifail, info)
call dsbevx ( jobz, range, uplo, n, kd, ab, ldab, q, ldq, vl, vu, il,
              iu, abstol, m, w, z, ldz, work, iwork, ifail, info)
```

### Discussion

This routine computes selected eigenvalues and, optionally, eigenvectors of a real symmetric band matrix  $A$ . Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

### Input Parameters

<i>jobz</i>	CHARACTER*1. Must be ' <b>N</b> ' or ' <b>V</b> '. If <i>job</i> = ' <b>N</b> ', then only eigenvalues are computed. If <i>job</i> = ' <b>V</b> ', then eigenvalues and eigenvectors are computed.
<i>range</i>	CHARACTER*1. Must be ' <b>A</b> ' or ' <b>V</b> ' or ' <b>I</b> '. If <i>range</i> = ' <b>A</b> ', the routine computes all eigenvalues. If <i>range</i> = ' <b>V</b> ', the routine computes eigenvalues $\lambda_i$ in the half-open interval: $vl < \lambda_i \leq vu$ . If <i>range</i> = ' <b>I</b> ', the routine computes eigenvalues with indices <i>il</i> to <i>iu</i> .
<i>uplo</i>	CHARACTER*1. Must be ' <b>U</b> ' or ' <b>L</b> '. If <i>uplo</i> = ' <b>U</b> ', <i>ab</i> stores the upper triangular part of $A$ . If <i>uplo</i> = ' <b>L</b> ', <i>ab</i> stores the lower triangular part of $A$ .
<i>n</i>	INTEGER. The order of the matrix $A$ ( <i>n</i> $\geq 0$ ).
<i>kd</i>	INTEGER. The number of super- or sub-diagonals in $A$ ( <i>kd</i> $\geq 0$ ).

---

<i>ab</i> , <i>work</i>	<small>REAL for ssbevx DOUBLE PRECISION for dsbevx.</small>
	Arrays: <i>ab</i> ( <i>ldab</i> , *) is an array containing either upper or lower triangular part of the symmetric matrix <i>A</i> (as specified by <i>uplo</i> ) in band storage format. The second dimension of <i>ab</i> must be at least $\max(1, n)$ . <i>work</i> (*) is a workspace array. The dimension of <i>work</i> must be at least $\max(1, 7n)$ .
<i>ldab</i>	<small>INTEGER. The leading dimension of <i>ab</i>; must be at least <i>kd</i> + 1.</small>
<i>vl</i> , <i i="" vu<=""></i>	<small>REAL for ssbevx DOUBLE PRECISION for dsbevx.</small> If <i>range</i> = 'V', the lower and upper bounds of the interval to be searched for eigenvalues. Constraint: <i>vl</i> < <i>vu</i> . If <i>range</i> = 'A' or 'I', <i>vl</i> and <i>vu</i> are not referenced.
<i>il</i> , <i>iu</i>	<small>INTEGER. If <i>range</i> = 'I', the indices in ascending order of the smallest and largest eigenvalues to be returned. Constraint: <math>1 \leq il \leq iu \leq n</math>, if <i>n</i> &gt; 0; <i>il</i>=1 and <i>iu</i>=0 if <i>n</i> = 0. If <i>range</i> = 'A' or 'V', <i>il</i> and <i>iu</i> are not referenced.</small>
<i>abstol</i>	<small>REAL for chpevx DOUBLE PRECISION for zhpevx</small> The absolute error tolerance to which each eigenvalue is required. See <i>Application notes</i> for details on error tolerance.
<i>ldq</i> , <i>ldz</i>	<small>INTEGER. The leading dimensions of the output arrays <i>q</i> and <i>z</i>, respectively. Constraints: <math>ldq \geq 1</math>, <math>ldz \geq 1</math>; If <i>jobz</i> = 'V', then <math>ldq \geq \max(1, n)</math> and <math>ldz \geq \max(1, n)</math>.</small>
<i>iwork</i>	<small>INTEGER. Workspace array, DIMENSION at least <math>\max(1, 5n)</math>.</small>

---

## Output Parameters

*m* **INTEGER.** The total number of eigenvalues found,  
 $0 \leq m \leq n$ . If *range* = 'A', *m* = *n*, and if *range* = 'I',  
 $m = iu - il + 1$ .

*w, z* **REAL** for *ssbevx*  
**DOUBLE PRECISION** for *dsbevx*

Arrays:

*w(\*)*, **DIMENSION** at least  $\max(1, n)$ .

The first *m* elements of *w* contain the selected eigenvalues of the matrix *A* in ascending order.

*z(ldz, \*)*. The second dimension of *z* must be at least  $\max(1, m)$ .

If *jobz* = 'V', then if *info* = 0, the first *m* columns of *z* contain the orthonormal eigenvectors of the matrix *A* corresponding to the selected eigenvalues, with the *i*-th column of *z* holding the eigenvector associated with *w(i)*. If an eigenvector fails to converge, then that column of *z* contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in *ifail*.

If *jobz* = 'N', then *z* is not referenced.

Note: you must ensure that at least  $\max(1, m)$  columns are supplied in the array *z*; if *range* = 'V', the exact value of *m* is not known in advance and an upper bound must be used.

*ab* On exit, this array is overwritten by the values generated during the reduction to tridiagonal form. If *uplo* = 'U', the first superdiagonal and the diagonal of the tridiagonal matrix *T* are returned in rows *kd* and *kd*+1 of *ab*, and if *uplo* = 'L', the diagonal and first subdiagonal of *T* are returned in the first two rows of *ab*.

*ifail* **INTEGER.**  
Array, **DIMENSION** at least  $\max(1, n)$ .  
If *jobz* = 'V', then if *info* = 0, the first *m* elements of

*ifail* are zero; if *info* > 0, the *ifail* contains the indices the eigenvectors that failed to converge.  
If *jobz* = 'N', then *ifail* is not referenced.

*info*

INTEGER.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*th parameter had an illegal value.

If *info* = *i*, then *i* eigenvectors failed to converge;  
their indices are stored in the array *ifail*.

## Application Notes

An approximate eigenvalue is accepted as converged when it is determined to lie in an interval [a,b] of width less than or equal to

*abstol* +  $\epsilon * \max(|a|, |b|)$ , where  $\epsilon$  is the machine precision. If *abstol* is less than or equal to zero, then  $\epsilon * \|T\|_1$  will be used in its place, where *T* is the tridiagonal matrix obtained by reducing *A* to tridiagonal form.

Eigenvalues will be computed most accurately when *abstol* is set to twice the underflow threshold  $2 * ?lamch('S')$ , not zero. If this routine returns with *info* > 0, indicating that some eigenvectors did not converge, try setting *abstol* to  $2 * ?lamch('S')$ .

## ?hbevx

*Computes selected eigenvalues and, optionally, eigenvectors of a Hermitian band matrix.*

---

```
call chbevx ( jobz, range, uplo, n, kd, ab, ldab, q, ldq, vl, vu, il,
              iu, abstol, m, w, z, ldz, work, rwork, iwork, ifail, info)
call zhbevx ( jobz, range, uplo, n, kd, ab, ldab, q, ldq, vl, vu, il,
              iu, abstol, m, w, z, ldz, work, rwork, iwork, ifail, info)
```

### Discussion

This routine computes selected eigenvalues and, optionally, eigenvectors of a complex Hermitian band matrix  $A$ . Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

### Input Parameters

<i>jobz</i>	CHARACTER*1. Must be ' <b>N</b> ' or ' <b>V</b> '. If <i>job</i> = ' <b>N</b> ', then only eigenvalues are computed. If <i>job</i> = ' <b>V</b> ', then eigenvalues and eigenvectors are computed.
<i>range</i>	CHARACTER*1. Must be ' <b>A</b> ' or ' <b>V</b> ' or ' <b>I</b> '. If <i>range</i> = ' <b>A</b> ', the routine computes all eigenvalues. If <i>range</i> = ' <b>V</b> ', the routine computes eigenvalues $\lambda_i$ in the half-open interval: $vl < \lambda_i \leq vu$ . If <i>range</i> = ' <b>I</b> ', the routine computes eigenvalues with indices <i>il</i> to <i>iu</i> .
<i>uplo</i>	CHARACTER*1. Must be ' <b>U</b> ' or ' <b>L</b> '. If <i>uplo</i> = ' <b>U</b> ', <i>ab</i> stores the upper triangular part of $A$ . If <i>uplo</i> = ' <b>L</b> ', <i>ab</i> stores the lower triangular part of $A$ .
<i>n</i>	INTEGER. The order of the matrix $A$ ( <i>n</i> $\geq 0$ ).
<i>kd</i>	INTEGER. The number of super- or sub-diagonals in $A$ ( <i>kd</i> $\geq 0$ ).

---

<i>ab</i> , <i>work</i>	<small>COMPLEX for chbevx DOUBLE COMPLEX for zhbevx.</small>
	Arrays: <i>ab</i> ( <i>ldab</i> , *) is an array containing either upper or lower triangular part of the Hermitian matrix <i>A</i> (as specified by <i>uplo</i> ) in band storage format. The second dimension of <i>ab</i> must be at least $\max(1, n)$ . <i>work</i> (*) is a workspace array. The dimension of <i>work</i> must be at least $\max(1, n)$ .
<i>ldab</i>	<small>INTEGER. The leading dimension of <i>ab</i>; must be at least <i>kd</i> +1.</small>
<i>vl</i> , <i>vu</i>	<small>REAL for chbevx DOUBLE PRECISION for zhbevx.</small> If <i>range</i> = 'V', the lower and upper bounds of the interval to be searched for eigenvalues. Constraint: <i>vl</i> < <i>vu</i> . If <i>range</i> = 'A' or 'I', <i>vl</i> and <i>vu</i> are not referenced.
<i>il</i> , <i>iu</i>	<small>INTEGER. If <i>range</i> = 'I', the indices in ascending order of the smallest and largest eigenvalues to be returned. Constraint: <math>1 \leq il \leq iu \leq n</math>, if <i>n</i> &gt; 0; <i>il</i>=1 and <i>iu</i>=0 if <i>n</i> = 0. If <i>range</i> = 'A' or 'V', <i>il</i> and <i>iu</i> are not referenced.</small>
<i>abstol</i>	<small>REAL for chbevx DOUBLE PRECISION for zhbevx.</small> The absolute error tolerance to which each eigenvalue is required. See <i>Application notes</i> for details on error tolerance.
<i>ldq</i> , <i>ldz</i>	<small>INTEGER. The leading dimensions of the output arrays <i>q</i> and <i>z</i>, respectively. Constraints: <math>ldq \geq 1</math>, <math>ldz \geq 1</math>; If <i>jobz</i> = 'V', then <math>ldq \geq \max(1, n)</math> and <math>ldz \geq \max(1, n)</math>.</small>

<i>rwork</i>	<code>REAL</code> for <code>chbevx</code> <code>DOUBLE PRECISION</code> for <code>zhbevx</code>
	Workspace array, <code>DIMENSION</code> at least <code>max(1, 7n)</code> .
<i>iwork</i>	<code>INTEGER</code> . Workspace array, <code>DIMENSION</code> at least <code>max(1, 5n)</code> .

## Output Parameters

<i>m</i>	<code>INTEGER</code> . The total number of eigenvalues found, $0 \leq m \leq n$ . If <code>range = 'A'</code> , $m = n$ , and if <code>range = 'I'</code> , $m = iu - il + 1$ .
<i>w</i>	<code>REAL</code> for <code>chbevx</code> <code>DOUBLE PRECISION</code> for <code>zhbevx</code> Array, <code>DIMENSION</code> at least <code>max(1, n)</code> . The first <i>m</i> elements contain the selected eigenvalues of the matrix <i>A</i> in ascending order.
<i>z</i>	<code>COMPLEX</code> for <code>chbevx</code> <code>DOUBLE COMPLEX</code> for <code>zhbevx</code> . Array <code>z(ldz, *)</code> . The second dimension of <i>z</i> must be at least <code>max(1, m)</code> . If <code>jobz = 'V'</code> , then if <code>info = 0</code> , the first <i>m</i> columns of <i>z</i> contain the orthonormal eigenvectors of the matrix <i>A</i> corresponding to the selected eigenvalues, with the <i>i</i> -th column of <i>z</i> holding the eigenvector associated with <code>w(i)</code> . If an eigenvector fails to converge, then that column of <i>z</i> contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in <code>ifail</code> . If <code>jobz = 'N'</code> , then <i>z</i> is not referenced. Note: you must ensure that at least <code>max(1, m)</code> columns are supplied in the array <i>z</i> ; if <code>range = 'V'</code> , the exact value of <i>m</i> is not known in advance and an upper bound must be used.
<i>ab</i>	On exit, this array is overwritten by the values generated during the reduction to tridiagonal form. If <code>uplo = 'U'</code> , the first superdiagonal and the diagonal of the

tridiagonal matrix  $T$  are returned in rows  $kd$  and  $kd+1$  of  $ab$ , and if  $uplo = 'L'$ , the diagonal and first subdiagonal of  $T$  are returned in the first two rows of  $ab$ .

*ifail***INTEGER.**Array, **DIMENSION** at least  $\max(1, n)$ .If  $jobz = 'V'$ , then if  $info = 0$ , the first  $m$  elements of *ifail* are zero; if  $info > 0$ , the *ifail* contains the indices of the eigenvectors that failed to converge.If  $jobz = 'N'$ , then *ifail* is not referenced.*info***INTEGER.**If  $info = 0$ , the execution is successful.If  $info = -i$ , the  $i$ th parameter had an illegal value.If  $info = i$ , then  $i$  eigenvectors failed to converge; their indices are stored in the array *ifail*.

## Application Notes

An approximate eigenvalue is accepted as converged when it is determined to lie in an interval  $[a,b]$  of width less than or equal to

$abstol + \epsilon * \max(|a|, |b|)$ , where  $\epsilon$  is the machine precision. If *abstol* is less than or equal to zero, then  $\epsilon * \|T\|_1$  will be used in its place, where  $T$  is the tridiagonal matrix obtained by reducing  $A$  to tridiagonal form.

Eigenvalues will be computed most accurately when *abstol* is set to twice the underflow threshold  $2 * ?lamch('S')$ , not zero. If this routine returns with  $info > 0$ , indicating that some eigenvectors did not converge, try setting *abstol* to  $2 * ?lamch('S')$ .

---

## ?stev

*Computes all eigenvalues and,  
optionally, eigenvectors of a real  
symmetric tridiagonal matrix.*

---

```
call sstev (jobz, n, d, e, z, ldz, work, info)
call dstev (jobz, n, d, e, z, ldz, work, info)
```

### Discussion

This routine computes all eigenvalues and, optionally, eigenvectors of a real symmetric tridiagonal matrix  $A$ .

### Input Parameters

<i>jobz</i>	CHARACTER*1. Must be ' <b>N</b> ' or ' <b>V</b> '. If <i>jobz</i> = ' <b>N</b> ', then only eigenvalues are computed. If <i>jobz</i> = ' <b>V</b> ', then eigenvalues and eigenvectors are computed.
<i>n</i>	INTEGER. The order of the matrix $A$ ( <i>n</i> $\geq$ 0).
<i>d, e, work</i>	REAL for <b>sstev</b> DOUBLE PRECISION for <b>dstev</b> . Arrays: <i>d(*)</i> contains the <i>n</i> diagonal elements of the tridiagonal matrix $A$ . The dimension of <i>d</i> must be at least max(1, <i>n</i> ). <i>e(*)</i> contains the <i>n</i> -1 subdiagonal elements of the tridiagonal matrix $A$ . The dimension of <i>e</i> must be at least max(1, <i>n</i> ). The <i>n</i> th element of this array is used as workspace. <i>work(*)</i> is a workspace array. The dimension of <i>work</i> must be at least max(1, 2 <i>n</i> -2). If <i>jobz</i> = ' <b>N</b> ', <i>work</i> is not referenced.

*ldz*            **INTEGER.** The leading dimension of the output array *z*;  
*ldz*  $\geq 1$ . If *jobz* = 'V' then *ldz*  $\geq \max(1, n)$ .

### Output Parameters

<i>d</i>	On exit, if <i>info</i> = 0, contains the eigenvalues of the matrix <i>A</i> in ascending order.
<i>z</i>	<b>REAL</b> for <i>sstev</i> <b>DOUBLE PRECISION</b> for <i>dstev</i> Array, <b>DIMENSION</b> ( <i>ldz</i> , *). The second dimension of <i>z</i> must be at least $\max(1, n)$ . If <i>jobz</i> = 'V', then if <i>info</i> = 0, <i>z</i> contains the orthonormal eigenvectors of the matrix <i>A</i> , with the <i>i</i> -th column of <i>z</i> holding the eigenvector associated with the eigenvalue returned in <i>d(i)</i> . If <i>job</i> = 'N', then <i>z</i> is not referenced.
<i>e</i>	On exit, this array is overwritten with intermediate results.
<i>info</i>	<b>INTEGER.</b> If <i>info</i> = 0, the execution is successful. If <i>info</i> = <i>-i</i> , the <i>i</i> th parameter had an illegal value. If <i>info</i> = <i>i</i> , then the algorithm failed to converge; <i>i</i> elements of <i>e</i> did not converge to zero.

---

## ?stevd

*Computes all eigenvalues and  
(optionally) all eigenvectors of a real  
symmetric tridiagonal matrix using  
divide and conquer algorithm.*

---

```
call sstevd (job, n, d, e, z, ldz, work, lwork, iwork, liwork, info)
call dstevd (job, n, d, e, z, ldz, work, lwork, iwork, liwork, info)
```

### Discussion

This routine computes all the eigenvalues, and optionally all the eigenvectors, of a real symmetric tridiagonal matrix  $T$ . In other words, the routine can compute the spectral factorization of  $T$  as:  $T = Z\Lambda Z^T$ . Here  $\Lambda$  is a diagonal matrix whose diagonal elements are the eigenvalues  $\lambda_i$ , and  $Z$  is the orthogonal matrix whose columns are the eigenvectors  $z_i$ . Thus,

$$Tz_i = \lambda_i z_i \text{ for } i = 1, 2, \dots, n.$$

If the eigenvectors are requested, then this routine uses a divide and conquer algorithm to compute eigenvalues and eigenvectors. However, if only eigenvalues are required, then it uses the Pal-Walker-Kahan variant of the  $QL$  or  $QR$  algorithm.

There is no complex analogue of this routine.

### Input Parameters

<i>job</i>	CHARACTER*1. Must be ' <b>N</b> ' or ' <b>V</b> '. If <i>job</i> = ' <b>N</b> ', then only eigenvalues are computed. If <i>job</i> = ' <b>V</b> ', then eigenvalues and eigenvectors are computed.
<i>n</i>	INTEGER. The order of the matrix $T$ ( <i>n</i> $\geq 0$ ).
<i>d</i> , <i>e</i> , <i>work</i>	REAL for <b>sstevd</b> DOUBLE PRECISION for <b>dstevd</b> . Arrays:

$d(*)$  contains the  $n$  diagonal elements of the tridiagonal matrix  $T$ .

The dimension of  $d$  must be at least  $\max(1, n)$ .

$e(*)$  contains the  $n-1$  off-diagonal elements of  $T$ .

The dimension of  $e$  must be at least  $\max(1, n)$ . The  $n$ th element of this array is used as workspace.

$work(*)$  is a workspace array.

The dimension of  $work$  must be at least  $lwork$ .

$ldz$

**INTEGER**. The leading dimension of the output array  $z$ .

Constraints:

$ldz \geq 1$  if  $job = 'N'$ ;

$ldz \geq \max(1, n)$  if  $job = 'V'$ .

$lwork$

**INTEGER**. The dimension of the array  $work$ .

Constraints:

if  $job = 'N'$  or  $n \leq 1$ , then  $lwork \geq 1$ ;

if  $job = 'V'$  and  $n > 1$ , then

$lwork \geq 2n^2 + (3+2k)*n+1$ , where  $k$  is the smallest integer which satisfies  $2^k \geq n$ .

$iwork$

**INTEGER**.

Workspace array, **DIMENSION** at least  $liwork$ .

$liwork$

**INTEGER**. The dimension of the array  $iwork$ .

Constraints:

if  $job = 'N'$  or  $n \leq 1$ , then  $liwork \geq 1$ ;

if  $job = 'V'$  and  $n > 1$ , then  $liwork \geq 5n+2$ .

## Output Parameters

$d$

On exit, if  $info = 0$ , contains the eigenvalues of the matrix  $T$  in ascending order.

See also  $info$ .

$z$

**REAL** for  $sstevd$

**DOUBLE PRECISION** for  $dstevd$

Array, **DIMENSION**( $ldz, *$ ).

The second dimension of  $z$  must be:

at least 1 if  $job = 'N'$ ;

at least  $\max(1, n)$  if  $job = 'V'$ .

	If $\text{job} = \text{'V'}$ , then this array is overwritten by the orthogonal matrix $Z$ which contains the eigenvectors of $T$ . If $\text{job} = \text{'N'}$ , then $\text{z}$ is not referenced.
$e$	On exit, this array is overwritten with intermediate results.
$\text{work}(1)$	On exit, if $\text{lwork} > 0$ , then $\text{work}(1)$ returns the required minimal size of $\text{lwork}$ .
$iwork(1)$	On exit, if $\text{liwork} > 0$ , then $iwork(1)$ returns the required minimal size of $\text{liwork}$ .
$\text{info}$	<b>INTEGER.</b> If $\text{info} = 0$ , the execution is successful. If $\text{info} = i$ , then the algorithm failed to converge; $i$ indicates the number of elements of an intermediate tridiagonal form which did not converge to zero. If $\text{info} = -i$ , the $i$ th parameter had an illegal value.

## Application Notes

The computed eigenvalues and eigenvectors are exact for a matrix  $T + E$  such that  $\|E\|_2 = O(\varepsilon) \|T\|_2$ , where  $\varepsilon$  is the machine precision.

If  $\lambda_i$  is an exact eigenvalue, and  $\mu_i$  is the corresponding computed value, then

$$|\mu_i - \lambda_i| \leq c(n)\varepsilon \|T\|_2$$

where  $c(n)$  is a modestly increasing function of  $n$ .

If  $z_i$  is the corresponding exact eigenvector, and  $w_i$  is the corresponding computed vector, then the angle  $\theta(z_i, w_i)$  between them is bounded as follows:

$$\theta(z_i, w_i) \leq c(n)\varepsilon \|T\|_2 / \min_{i \neq j} |\lambda_i - \lambda_j|.$$

Thus the accuracy of a computed eigenvector depends on the gap between its eigenvalue and all the other eigenvalues.

## ?stevx

*Computes selected eigenvalues and eigenvectors of a real symmetric tridiagonal matrix.*

```
call sstevx ( jobz, range, n, d, e, vl, vu, il, iu, abstol, m, w, z,
              ldz, work, iwork, ifail, info)
call dstevx ( jobz, range, n, d, e, vl, vu, il, iu, abstol, m, w, z,
              ldz, work, iwork, ifail, info)
```

### Discussion

This routine computes selected eigenvalues and, optionally, eigenvectors of a real symmetric tridiagonal matrix  $A$ . Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

### Input Parameters

<i>jobz</i>	CHARACTER*1. Must be ' <b>N</b> ' or ' <b>V</b> '. If <i>job</i> = ' <b>N</b> ', then only eigenvalues are computed. If <i>job</i> = ' <b>V</b> ', then eigenvalues and eigenvectors are computed.
<i>range</i>	CHARACTER*1. Must be ' <b>A</b> ' or ' <b>V</b> ' or ' <b>I</b> '. If <i>range</i> = ' <b>A</b> ', the routine computes all eigenvalues. If <i>range</i> = ' <b>V</b> ', the routine computes eigenvalues $\lambda_i$ in the half-open interval: $vl < \lambda_i \leq vu$ . If <i>range</i> = ' <b>I</b> ', the routine computes eigenvalues with indices <i>il</i> to <i>iu</i> .
<i>n</i>	INTEGER. The order of the matrix $A$ ( <i>n</i> $\geq 0$ ).
<i>d</i> , <i>e</i> , <i>work</i>	REAL for <b>sstevx</b> DOUBLE PRECISION for <b>dstevx</b> . Arrays:

$d(*)$  contains the  $n$  diagonal elements of the tridiagonal matrix  $A$ .

The dimension of  $d$  must be at least  $\max(1, n)$ .

$e(*)$  contains the  $n-1$  subdiagonal elements of  $A$ .

The dimension of  $e$  must be at least  $\max(1, n)$ . The  $n$ th element of this array is used as workspace.

$work(*)$  is a workspace array.

The dimension of  $work$  must be at least  $\max(1, 5n)$ .

$vl, vu$

REAL for `sstevx`

DOUBLE PRECISION for `dstevx`.

If  $range = 'V'$ , the lower and upper bounds of the interval to be searched for eigenvalues.

Constraint:  $vl < vu$ .

If  $range = 'A'$  or ' $I$ ',  $vl$  and  $vu$  are not referenced.

$il, iu$

INTEGER.

If  $range = 'I'$ , the indices in ascending order of the smallest and largest eigenvalues to be returned.

Constraint:  $1 \leq il \leq iu \leq n$ , if  $n > 0$ ;  $il=1$  and  $iu=0$  if  $n = 0$ .

If  $range = 'A'$  or ' $V$ ',  $il$  and  $iu$  are not referenced.

$abstol$

REAL for `sstevx`

DOUBLE PRECISION for `dstevx`.

The absolute error tolerance to which each eigenvalue is required. See *Application notes* for details on error tolerance.

$ldz$

INTEGER. The leading dimensions of the output array  $z$ ;  $ldz \geq 1$ . If  $jobz = 'V'$ , then  $ldz \geq \max(1, n)$ .

$iwork$

INTEGER.

Workspace array, DIMENSION at least  $\max(1, 5n)$ .

## Output Parameters

*m*

**INTEGER.** The total number of eigenvalues found,  
 $0 \leq m \leq n$ . If *range* = 'A', *m* = *n*, and if *range* = 'I',  
 $m = iu - il + 1$ .

*w, z*

**REAL** for *sstevx*  
**DOUBLE PRECISION** for *dstevx*.

Arrays:

*w(\*)*, **DIMENSION** at least  $\max(1, n)$ .

The first *m* elements of *w* contain the selected eigenvalues of the matrix *A* in ascending order.

*z(ldz, \*)*. The second dimension of *z* must be at least  $\max(1, m)$ .

If *jobz* = 'V', then if *info* = 0, the first *m* columns of *z* contain the orthonormal eigenvectors of the matrix *A* corresponding to the selected eigenvalues, with the *i*-th column of *z* holding the eigenvector associated with *w(i)*. If an eigenvector fails to converge, then that column of *z* contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in *ifail*.

If *jobz* = 'N', then *z* is not referenced.

Note: you must ensure that at least  $\max(1, m)$  columns are supplied in the array *z*; if *range* = 'V', the exact value of *m* is not known in advance and an upper bound must be used.

*d, e*

On exit, these arrays may be multiplied by a constant factor chosen to avoid overflow or underflow in computing the eigenvalues.

*ifail*

**INTEGER.**

Array, **DIMENSION** at least  $\max(1, n)$ .

If *jobz* = 'V', then if *info* = 0, the first *m* elements of *ifail* are zero; if *info* > 0, the *ifail* contains the indices of the eigenvectors that failed to converge.

If *jobz* = 'N', then *ifail* is not referenced.

*info*

INTEGER.

If *info* = 0, the execution is successful.If *info* = -*i*, the *i*th parameter had an illegal value.If *info* = *i*, then *i* eigenvectors failed to converge; their indices are stored in the array *ifail*.

## Application Notes

An approximate eigenvalue is accepted as converged when it is determined to lie in an interval [a,b] of width less than or equal to

*abstol* +  $\epsilon * \max(|a|, |b|)$ , where  $\epsilon$  is the machine precision. If *abstol* is less than or equal to zero, then  $\epsilon * \|A\|_1$  will be used in its place.

Eigenvalues will be computed most accurately when *abstol* is set to twice the underflow threshold  $2 * ?lamch('S')$ , not zero. If this routine returns with *info* > 0, indicating that some eigenvectors did not converge, try setting *abstol* to  $2 * ?lamch('S')$ .

## ?stevr

*Computes selected eigenvalues and, optionally, eigenvectors of a real symmetric tridiagonal matrix using the Relatively Robust Representations.*

```
call sstevr ( jobz, range, n, d, e, vl, vu, il, iu, abstol, m, w, z,
              ldz, isuppz, work, lwork, iwork, liwork, info)
call dstevr ( jobz, range, n, d, e, vl, vu, il, iu, abstol, m, w, z,
              ldz, isuppz, work, lwork, iwork, liwork, info)
```

### Discussion

This routine computes selected eigenvalues and, optionally, eigenvectors of a real symmetric tridiagonal matrix  $T$ . Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

Whenever possible, `?stevr` calls `ssteqr/dstegr` to compute the eigenspectrum using Relatively Robust Representations. `?steqr` computes eigenvalues by the *dqds* algorithm, while orthogonal eigenvectors are computed from various "good"  $LDL^T$  representations (also known as Relatively Robust Representations). Gram-Schmidt orthogonalization is avoided as far as possible. More specifically, the various steps of the algorithm are as follows. For the  $i$ -th unreduced block of  $T$ ,

- (a) Compute  $T - \sigma_i = L_i D_i L_i^T$ , such that  $L_i D_i L_i^T$  is a relatively robust representation;
- (b) Compute the eigenvalues,  $\lambda_j$ , of  $L_i D_i L_i^T$  to high relative accuracy by the *dqds* algorithm;
- (c) If there is a cluster of close eigenvalues, "choose"  $\sigma_i$  close to the cluster, and go to step (a);
- (d) Given the approximate eigenvalue  $\lambda_j$  of  $L_i D_i L_i^T$ , compute the corresponding eigenvector by forming a rank-revealing twisted factorization.

The desired accuracy of the output can be specified by the input parameter *abstol*.

The routine `?stevr` calls `ssteqr/dsteqr` when the full spectrum is requested on machines which conform to the IEEE-754 floating point standard. `?stevr` calls `sstebz/dstebz` and `sstein/dstein` on non-IEEE machines and when partial spectrum requests are made.

## Input Parameters

<i>jobz</i>	CHARACTER*1. Must be ' <code>N</code> ' or ' <code>V</code> '. If <i>jobz</i> = ' <code>N</code> ', then only eigenvalues are computed. If <i>jobz</i> = ' <code>V</code> ', then eigenvalues and eigenvectors are computed.
<i>range</i>	CHARACTER*1. Must be ' <code>A</code> ' or ' <code>V</code> ' or ' <code>I</code> '. If <i>range</i> = ' <code>A</code> ', the routine computes all eigenvalues. If <i>range</i> = ' <code>V</code> ', the routine computes eigenvalues $\lambda_i$ in the half-open interval: $v_l < \lambda_i \leq v_u$ . If <i>range</i> = ' <code>I</code> ', the routine computes eigenvalues with indices <i>il</i> to <i>iu</i> .  For <i>range</i> = ' <code>V</code> ' or ' <code>I</code> ', <code>sstebz/dstebz</code> and <code>sstein/dstein</code> are called.
<i>n</i>	INTEGER. The order of the matrix <i>T</i> ( <i>n</i> $\geq 0$ ).
<i>d</i> , <i>e</i> , <i>work</i>	REAL for <code>sstevr</code> DOUBLE PRECISION for <code>dstevr</code> .  Arrays: <i>d(*)</i> contains the <i>n</i> diagonal elements of the tridiagonal matrix <i>T</i> . The dimension of <i>d</i> must be at least $\max(1, n)$ . <i>e(*)</i> contains the <i>n</i> -1 subdiagonal elements of <i>A</i> . The dimension of <i>e</i> must be at least $\max(1, n)$ . The <i>n</i> th element of this array is used as workspace. <i>work(lwork)</i> is a workspace array.

---

<i>vl, vu</i>	<small>REAL for <code>sstevr</code> DOUBLE PRECISION for <code>dstevr</code>.</small> If <i>range</i> = 'V', the lower and upper bounds of the interval to be searched for eigenvalues. Constraint: <i>vl</i> < <i>vu</i> . If <i>range</i> = 'A' or 'I', <i>vl</i> and <i>vu</i> are not referenced.
<i>il, iu</i>	<small>INTEGER. If <i>range</i> = 'I', the indices in ascending order of the smallest and largest eigenvalues to be returned. Constraint: <math>1 \leq il \leq iu \leq n</math>, if <i>n</i> &gt; 0; <i>il</i>=1 and <i>iu</i>=0 if <i>n</i> = 0. If <i>range</i> = 'A' or 'V', <i>il</i> and <i>iu</i> are not referenced.</small>
<i>abstol</i>	<small>REAL for <code>ssyevr</code> DOUBLE PRECISION for <code>dsyevr</code>. The absolute error tolerance to which each eigenvalue/eigenvector is required. If <i>jobz</i> = 'V', the eigenvalues and eigenvectors output have residual norms bounded by <i>abstol</i>, and the dot products between different eigenvectors are bounded by <i>abstol</i>. If <i>abstol</i> &lt; <i>n</i><math>\epsilon\ T\ _1</math>, then <i>n</i><math>\epsilon\ T\ _1</math> will be used in its place, where <math>\epsilon</math> is the machine precision. The eigenvalues are computed to an accuracy of <math>\epsilon\ T\ _1</math> irrespective of <i>abstol</i>. If high relative accuracy is important, set <i>abstol</i> to <code>?lamch('S')</code>.</small>
<i>ldz</i>	<small>INTEGER. The leading dimension of the output array <i>z</i>. Constraints: <math>ldz \geq 1</math> if <i>jobz</i> = 'N'; <math>ldz \geq \max(1, n)</math> if <i>jobz</i> = 'V'.</small>
<i>lwork</i>	<small>INTEGER. The dimension of the array <i>work</i>. Constraint: <i>lwork</i> <math>\geq \max(1, 20n)</math>.</small>
<i>iwork</i>	<small>INTEGER. Workspace array, DIMENSION (<i>liwork</i>).</small>
<i>liwork</i>	<small>INTEGER. The dimension of the array <i>iwork</i>, <i>lwork</i> <math>\geq \max(1, 10n)</math>.</small>

---

## Output Parameters

*m* **INTEGER.** The total number of eigenvalues found,  
 $0 \leq m \leq n$ . If *range* = 'A', *m* = *n*, and if *range* = 'I',  
*m* = *iu* - *il* + 1.

*w, z* **REAL** for **sstevr**  
**DOUBLE PRECISION** for **dstevr**.

Arrays:

*w(\*)*, **DIMENSION** at least  $\max(1, n)$ .

The first *m* elements of *w* contain the selected eigenvalues of the matrix *T* in ascending order.

*z(ldz, \*)*. The second dimension of *z* must be at least  $\max(1, m)$ .

If *jobz* = 'V', then if *info* = 0, the first *m* columns of *z* contain the orthonormal eigenvectors of the matrix *T* corresponding to the selected eigenvalues, with the *i*-th column of *z* holding the eigenvector associated with *w(i)*.

If *jobz* = 'N', then *z* is not referenced.

Note: you must ensure that at least  $\max(1, m)$  columns are supplied in the array *z*; if *range* = 'V', the exact value of *m* is not known in advance and an upper bound must be used.

*d, e* On exit, these arrays may be multiplied by a constant factor chosen to avoid overflow or underflow in computing the eigenvalues.

*isuppz* **INTEGER.**

Array, **DIMENSION** at least  $2 * \max(1, m)$ .

The support of the eigenvectors in *z*, i.e., the indices indicating the nonzero elements in *z*. The *i*-th eigenvector is nonzero only in elements *isuppz(2i-1)* through *isuppz(2i)*.

*work(1)* On exit, if *info* = 0, then *work(1)* returns the required minimal size of *lwork*.

*iwork(1)* On exit, if *info* = 0, then *iwork(1)* returns the required minimal size of *liwork*.

*info*

INTEGER.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*th parameter had an illegal value.

If *info* = *i*, an internal error has occurred.

### Application Notes

Normal execution of the routine `?steqr` may create NaNs and infinities and hence may abort due to a floating point exception in environments which do not handle NaNs and infinities in the IEEE standard default manner.

## Nonsymmetric Eigenproblems

This section describes LAPACK driver routines used for solving nonsymmetric eigenproblems. See also [computational routines](#) that can be called to solve these problems.

[Table 5-12](#) lists routines described in more detail below.

**Table 5-11 Driver Routines for Solving Nonsymmetric Eigenproblems**

Routine Name	Operation performed
<a href="#">?gees</a>	Computes the eigenvalues and Shur factorization of a general matrix, and orders the factorization so that selected eigenvalues are at the top left of the Shur form.
<a href="#">?geesx</a>	Computes the eigenvalues and Shur factorization of a general matrix, orders the factorization and computes reciprocal condition numbers.
<a href="#">?geev</a>	Computes the eigenvalues and left and right eigenvectors of a general matrix.
<a href="#">?geevx</a>	Computes the eigenvalues and left and right eigenvectors of a general matrix, with preliminary matrix balancing, and computes reciprocal condition numbers for the eigenvalues and right eigenvectors.

## ?gees

*Computes the eigenvalues and Shur factorization of a general matrix, and orders the factorization so that selected eigenvalues are at the top left of the Shur form.*

```
call sgees ( jobvs, sort, select, n, a, lda, sdim, wr, wi, vs, ldvs,
             work, lwork, bwork, info)
call dgees ( jobvs, sort, select, n, a, lda, sdim, wr, wi, vs, ldvs,
             work, lwork, bwork, info)
call cgees ( jobvs, sort, select, n, a, lda, sdim, w, vs, ldvs,
             work, lwork, rwork, bwork, info)
call zgees ( jobvs, sort, select, n, a, lda, sdim, w, vs, ldvs,
             work, lwork, rwork, bwork, info)
```

## Discussion

This routine computes for an  $n$ -by- $n$  real/complex nonsymmetric matrix  $A$ , the eigenvalues, the real Schur form  $T$ , and, optionally, the matrix of Schur vectors  $Z$ . This gives the Schur factorization  $A = ZTZ^H$ .

Optionally, it also orders the eigenvalues on the diagonal of the real-Schur/Shur form so that selected eigenvalues are at the top left. The leading columns of  $Z$  then form an orthonormal basis for the invariant subspace corresponding to the selected eigenvalues.

A real matrix is in real-Schur form if it is upper quasi-triangular with 1-by-1 and 2-by-2 blocks. 2-by-2 blocks will be standardized in the form

$$\begin{pmatrix} a & b \\ c & a \end{pmatrix}$$

where  $b*c < 0$ . The eigenvalues of such a block are  $a \pm \sqrt{bc}$ .

A complex matrix is in Schur form if it is upper triangular.

## Input Parameters

<i>jobvs</i>	CHARACTER*1. Must be ' <b>N</b> ' or ' <b>V</b> '. If <i>jobvs</i> = ' <b>N</b> ', then Shur vectors are not computed. If <i>jobvs</i> = ' <b>V</b> ', then Shur vectors are computed.
<i>sort</i>	CHARACTER*1. Must be ' <b>N</b> ' or ' <b>S</b> '. Specifies whether or not to order the eigenvalues on the diagonal of the Schur form. If <i>sort</i> = ' <b>N</b> ', then eigenvalues are not ordered. If <i>sort</i> = ' <b>S</b> ', eigenvalues are ordered (see <i>select</i> ).
<i>select</i>	LOGICAL FUNCTION of two REAL arguments for real flavors. LOGICAL FUNCTION of one COMPLEX argument for complex flavors. <i>select</i> must be declared EXTERNAL in the calling subroutine. If <i>sort</i> = ' <b>S</b> ', <i>select</i> is used to select eigenvalues to sort to the top left of the Shur form. If <i>sort</i> = ' <b>N</b> ', <i>select</i> is not referenced.

*For real flavors:*

An eigenvalue  $\text{wr}(j) + \sqrt{-1} * \text{wi}(j)$  is selected if  $\text{select}(\text{wr}(j), \text{wi}(j))$  is true; that is, if either one of a complex conjugate pair of eigenvalues is selected, then both complex eigenvalues are selected. Note that a selected complex eigenvalue may no longer satisfy  $\text{select}(\text{wr}(j), \text{wi}(j)) = .\text{TRUE}.$  after ordering, since ordering may change the value of complex eigenvalues (especially if the eigenvalue is ill-conditioned); in this case  $\text{info}$  may be set to  $n+2$  (see  $\text{info}$  below).

*For complex flavors:*

An eigenvalue  $w(j)$  is selected if  $\text{select}(w(j))$  is true.

***n*** **INTEGER.** The order of the matrix  $A$  ( $n \geq 0$ ).

***a, work*** **REAL** for **sgees**  
**DOUBLE PRECISION** for **dgees**  
**COMPLEX** for **cgees**  
**DOUBLE COMPLEX** for **zgees**.

**Arrays:**

***a( lda, \* )*** is an array containing the  $n$ -by- $n$  matrix  $A$ . The second dimension of ***a*** must be at least  $\max(1, n)$ .

***work( lwork )*** is a workspace array.

***lda*** **INTEGER.** The first dimension of the array ***a***. Must be at least  $\max(1, n)$ .

***ldvs*** **INTEGER.** The leading dimension of the output array ***vs***. Constraints:  
 $ldvs \geq 1$  ;  
 $ldvs \geq \max(1, n)$  if  $\text{jobvs} = 'V'$ .

***lwork*** **INTEGER.** The dimension of the array ***work***. Constraint:

$lwork \geq \max(1, 3n)$  for real flavors;  
 $lwork \geq \max(1, 2n)$  for complex flavors.

***rwork*** **REAL** for **cgees**  
**DOUBLE PRECISION** for **zgees**  
 Workspace array, **DIMENSION** at least  $\max(1, n)$ . Used in complex flavors only.

---

<i>bwork</i>	<b>LOGICAL.</b> Workspace array, <b>DIMENSION</b> at least $\max(1, n)$ . Not referenced if <i>sort</i> = 'N'.
<b>Output Parameters</b>	
<i>a</i>	On exit, this array is overwritten by the real-Shur/Shur form $T$ .
<i>sdim</i>	<b>INTEGER.</b> If <i>sort</i> = 'N', <i>sdim</i> = 0. If <i>sort</i> = 'S', <i>sdim</i> is equal to the number of eigenvalues (after sorting) for which <i>select</i> is true. Note that for real flavors complex conjugate pairs for which <i>select</i> is true for either eigenvalue count as 2.
<i>wr, wi</i>	<b>REAL</b> for <b>sgees</b> <b>DOUBLE PRECISION</b> for <b>dgees</b> Arrays, <b>DIMENSION</b> at least $\max(1, n)$ each. Contain the real and imaginary parts, respectively, of the computed eigenvalues, in the same order that they appear on the diagonal of the output real-Shur form $T$ . Complex conjugate pairs of eigenvalues appear consecutively with the eigenvalue having positive imaginary part first.
<i>w</i>	<b>COMPLEX</b> for <b>cgees</b> <b>DOUBLE COMPLEX</b> for <b>zgees</b> . Array, <b>DIMENSION</b> at least $\max(1, n)$ . Contains the computed eigenvalues. The eigenvalues are stored in the same order as they appear on the diagonal of the output Shur form $T$ .
<i>vs</i>	<b>REAL</b> for <b>sgees</b> <b>DOUBLE PRECISION</b> for <b>dgees</b> <b>COMPLEX</b> for <b>cgees</b> <b>DOUBLE COMPLEX</b> for <b>zgees</b> . Array <i>vs</i> ( <i>ldvs</i> , *); the second dimension of <i>vs</i> must be at least $\max(1, n)$ .

If  $\text{jobvs} = \text{'V'}$ ,  $\text{vs}$  contains the orthogonal/unitary matrix  $Z$  of Schur vectors.

If  $\text{jobvs} = \text{'N'}$ ,  $\text{vs}$  is not referenced.

$\text{work}(1)$

On exit, if  $\text{info} = 0$ , then  $\text{work}(1)$  returns the required minimal size of  $\text{lwork}$ .

$\text{info}$

**INTEGER.**

If  $\text{info} = 0$ , the execution is successful.

If  $\text{info} = -i$ , the  $i$ th parameter had an illegal value.

If  $\text{info} = i$ , and

$i \leq n$  :

the  $QR$  algorithm failed to compute all the eigenvalues; elements  $1:\text{ilo}-1$  and  $i+1:n$  of  $\text{wr}$  and  $\text{wi}$  (for real flavors) or  $\text{w}$  (for complex flavors) contain those eigenvalues which have converged; if  $\text{jobvs} = \text{'V'}$ ,  $\text{vs}$  contains the matrix which reduces  $A$  to its partially converged Schur form;

$i = n+1$  :

the eigenvalues could not be reordered because some eigenvalues were too close to separate (the problem is very ill-conditioned);

$i = n+2$  :

after reordering, roundoff changed values of some complex eigenvalues so that leading eigenvalues in the Schur form no longer satisfy  $\text{select} = .\text{TRUE}..$ . This could also be caused by underflow due to scaling.

## Application Notes

If you are in doubt how much workspace to supply for the array  $\text{work}$ , use a generous value of  $\text{lwork}$  for the first run. On exit, examine  $\text{work}(1)$  and use this value for subsequent runs.

## ?geesx

*Computes the eigenvalues and Schur factorization of a general matrix, orders the factorization and computes reciprocal condition numbers.*

```
call sgeesx(jobvs, sort, select, sense, n, a, lda, sdim, wr, wi, vs,
            ldvs, rconde, rcondv, work, lwork, iwork, liwork, bwork, info)
call dgeesx(jobvs, sort, select, sense, n, a, lda, sdim, wr, wi, vs,
            ldvs, rconde, rcondv, work, lwork, iwork, liwork, bwork, info)
call cgeesx(jobvs, sort, select, sense, n, a, lda, sdim, w, vs,
            ldvs, rconde, rcondv, work, lwork, rwork, bwork, info)
call zgeesx(jobvs, sort, select, sense, n, a, lda, sdim, w, vs,
            ldvs, rconde, rcondv, work, lwork, rwork, bwork, info)
```

### Discussion

This routine computes for an  $n$ -by- $n$  real/complex nonsymmetric matrix  $A$ , the eigenvalues, the real-Schur/Shur form  $T$ , and, optionally, the matrix of Schur vectors  $Z$ . This gives the Schur factorization  $A = TZ^H$ .

Optionally, it also orders the eigenvalues on the diagonal of the real-Schur/Shur form so that selected eigenvalues are at the top left; computes a reciprocal condition number for the average of the selected eigenvalues (*rconde*); and computes a reciprocal condition number for the right invariant subspace corresponding to the selected eigenvalues (*rcondv*). The leading columns of  $Z$  form an orthonormal basis for this invariant subspace.

For further explanation of the reciprocal condition numbers *rconde* and *rcondv*, see [[LUG](#)], Section 4.10 (where these quantities are called *s* and *sep* respectively).

A real matrix is in real-Schur form if it is upper quasi-triangular with 1-by-1 and 2-by-2 blocks. 2-by-2 blocks will be standardized in the form

$$\begin{pmatrix} a & b \\ c & a \end{pmatrix}$$

where  $b*c < 0$ . The eigenvalues of such a block are  $a \pm \sqrt{bc}$ .

A complex matrix is in Schur form if it is upper triangular.

### Input Parameters

<i>jobvs</i>	CHARACTER*1. Must be 'N' or 'V'. If <i>jobvs</i> = 'N', then Shur vectors are not computed. If <i>jobvs</i> = 'V', then Shur vectors are computed.
<i>sort</i>	CHARACTER*1. Must be 'N' or 'S'. Specifies whether or not to order the eigenvalues on the diagonal of the Schur form. If <i>sort</i> = 'N', then eigenvalues are not ordered. If <i>sort</i> = 'S', eigenvalues are ordered (see <i>select</i> ).
<i>select</i>	LOGICAL FUNCTION of two REAL arguments for real flavors. LOGICAL FUNCTION of one COMPLEX argument for complex flavors.  <i>select</i> must be declared EXTERNAL in the calling subroutine. If <i>sort</i> = 'S', <i>select</i> is used to select eigenvalues to sort to the top left of the Schur form. If <i>sort</i> = 'N', <i>select</i> is not referenced. <i>For real flavors:</i> An eigenvalue $wr(j) + \sqrt{-1} * wi(j)$ is selected if <i>select(wr(j), wi(j))</i> is true; that is, if either one of a complex conjugate pair of eigenvalues is selected, then both complex eigenvalues are selected. Note that a selected complex eigenvalue may no longer satisfy <i>select(wr(j), wi(j)) = .TRUE.</i> after ordering, since ordering may change the value of complex eigenvalues (especially if the eigenvalue is ill-conditioned); in this case <i>info</i> may be set to <i>n+2</i> (see <i>info</i> below). <i>For complex flavors:</i> An eigenvalue <i>w(j)</i> is selected if <i>select(w(j))</i> is true.

---

<i>sense</i>	<b>CHARACTER*1.</b> Must be ' <b>N</b> ', ' <b>E</b> ', ' <b>V</b> ', or ' <b>B</b> '. Determines which reciprocal condition number are computed.  If <i>sense</i> = ' <b>N</b> ', none are computed; If <i>sense</i> = ' <b>E</b> ', computed for average of selected eigenvalues only; If <i>sense</i> = ' <b>V</b> ', computed for selected right invariant subspace only; If <i>sense</i> = ' <b>B</b> ', computed for both.  If <i>sense</i> is ' <b>E</b> ', ' <b>V</b> ', or ' <b>B</b> ', then <i>sort</i> must equal ' <b>S</b> '.
<i>n</i>	<b>INTEGER.</b> The order of the matrix <i>A</i> ( <i>n</i> ≥ 0).
<i>a, work</i>	<b>REAL</b> for <i>sgeesx</i> <b>DOUBLE PRECISION</b> for <i>dgeesx</i> <b>COMPLEX</b> for <i>cgeesx</i> <b>DOUBLE COMPLEX</b> for <i>zgeesx</i> .  Arrays: <i>a( lda, * )</i> is an array containing the <i>n</i> -by- <i>n</i> matrix <i>A</i> . The second dimension of <i>a</i> must be at least $\max(1, n)$ . <i>work(lwork)</i> is a workspace array.
<i>lda</i>	<b>INTEGER.</b> The first dimension of the array <i>a</i> . Must be at least $\max(1, n)$ .
<i>ldvs</i>	<b>INTEGER.</b> The leading dimension of the output array <i>vs</i> . Constraints: $ldvs \geq 1$ ; $ldvs \geq \max(1, n)$ if <i>jobvs</i> = ' <b>V</b> '.
<i>lwork</i>	<b>INTEGER.</b> The dimension of the array <i>work</i> . Constraint: $lwork \geq \max(1, 3n)$ for real flavors; $lwork \geq \max(1, 2n)$ for complex flavors.  Also, if <i>sense</i> = ' <b>E</b> ', ' <b>V</b> ', or ' <b>B</b> ', then $lwork \geq n+2*sdim*(n-sdim)$ for real flavors; $lwork \geq 2*sdim*(n-sdim)$ for complex flavors;

where  $sdim$  is the number of selected eigenvalues computed by this routine. Note that  
 $2 * sdim * (n - sdim) \leq n * n / 2$ .

For good performance,  $lwork$  must generally be larger.

*iwork*

INTEGER.

Workspace array, DIMENSION ( $liwork$ ). Used in real flavors only. Not referenced if  $sense = 'N'$  or ' $E$ '.

*liwork*

INTEGER. The dimension of the array *iwork*. Used in real flavors only. Constraint:

$liwork \geq 1$ ;

if  $sense = 'V'$  or ' $B$ ',  $liwork \geq sdim * (n - sdim)$ .

*rwork*

REAL for *cgeesx*

DOUBLE PRECISION for *zgeesx*

Workspace array, DIMENSION at least max(1, *n*). Used in complex flavors only.

*bwork*

LOGICAL.

Workspace array, DIMENSION at least max(1, *n*). Not referenced if  $sort = 'N'$ .

## Output Parameters

*a*

On exit, this array is overwritten by the real-Shur/Shur form  $T$ .

*sdim*

INTEGER.

If  $sort = 'N'$ ,  $sdim = 0$ .

If  $sort = 'S'$ ,  $sdim$  is equal to the number of eigenvalues (after sorting) for which *select* is true. Note that for real flavors complex conjugate pairs for which *select* is true for either eigenvalue count as 2.

*wr*, *wi*

REAL for *sgeesx*

DOUBLE PRECISION for *dgeesx*

Arrays, DIMENSION at least max (1, *n*) each.

Contain the real and imaginary parts, respectively, of the computed eigenvalues, in the same order that they appear on the diagonal of the output real-Shur form  $T$ .

Complex conjugate pairs of eigenvalues appear consecutively with the eigenvalue having positive imaginary part first.

*w***COMPLEX** for `cgeesx`**DOUBLE COMPLEX** for `zgeesx`.Array, **DIMENSION** at least  $\max(1, n)$ .Contains the computed eigenvalues. The eigenvalues are stored in the same order as they appear on the diagonal of the output Shur form  $T$ .*vs***REAL** for `sgeesx`**DOUBLE PRECISION** for `dgeesx`**COMPLEX** for `cgeesx`**DOUBLE COMPLEX** for `zgeesx`.Array  $vs(1:ldvs, :)$ ; the second dimension of *vs* must be at least  $\max(1, n)$ .If  $jobvs = 'V'$ , *vs* contains the orthogonal/unitary matrix  $Z$  of Shur vectors.If  $jobvs = 'N'$ , *vs* is not referenced.*rconde, rcondv***REAL** for single precision flavors**DOUBLE PRECISION** for double precision flavors.If *sense* = 'E' or 'B', *rconde* contains the reciprocal condition number for the average of the selected eigenvalues. If *sense* = 'N' or 'V', *rconde* is not referenced.If *sense* = 'V' or 'B', *rcondv* contains the reciprocal condition number for the selected right invariant subspace. If *sense* = 'N' or 'E', *rcondv* is not referenced.*work(1)*On exit, if *info*=0, then *work(1)* returns the required minimal size of *lwork*.*info***INTEGER**.If *info* = 0, the execution is successful.If *info* = *-i*, the *i*th parameter had an illegal value.

If  $\text{info} = i$ , and  
 $i \leq n$  :

the  $QR$  algorithm failed to compute all the eigenvalues; elements  $1:\text{ilo}-1$  and  $i+1:n$  of  $\text{wr}$  and  $\text{wi}$  (for real flavors) or  $\text{w}$  (for complex flavors) contain those eigenvalues which have converged; if  $\text{jobvs} = 'V'$ ,  $\text{vs}$  contains the transformation which reduces  $A$  to its partially converged Schur form;

$i = n+1$  :

the eigenvalues could not be reordered because some eigenvalues were too close to separate (the problem is very ill-conditioned);

$i = n+2$  :

after reordering, roundoff changed values of some complex eigenvalues so that leading eigenvalues in the Schur form no longer satisfy  $\text{select} = .\text{TRUE}..$ . This could also be caused by underflow due to scaling.

### Application Notes

If you are in doubt how much workspace to supply for the array  $\text{work}$ , use a generous value of  $\text{lwork}$  for the first run. On exit, examine  $\text{work}(1)$  and use this value for subsequent runs.

## ?geev

*Computes the eigenvalues and left and right eigenvectors of a general matrix.*

```
call sgeev ( jobvl, jobvr, n, a, lda, wr, wi, vl, ldvl, vr, ldvr,
            work, lwork, info)
call dgeev ( jobvl, jobvr, n, a, lda, wr, wi, vl, ldvl, vr, ldvr,
            work, lwork, info)
call cgeev ( jobvl, jobvr, n, a, lda, w, vl, ldvl, vr, ldvr, work,
            lwork, rwork, info)
call zgeev ( jobvl, jobvr, n, a, lda, w, vl, ldvl, vr, ldvr, work,
            lwork, rwork, info)
```

### Discussion

This routine computes for an  $n$ -by- $n$  real/complex nonsymmetric matrix  $A$ , the eigenvalues and, optionally, the left and/or right eigenvectors. The right eigenvector  $v(j)$  of  $A$  satisfies

$$A * v(j) = \lambda(j) * v(j)$$

where  $\lambda(j)$  is its eigenvalue.

The left eigenvector  $u(j)$  of  $A$  satisfies

$$u(j)^H * A = \lambda(j) * u(j)^H$$

where  $u(j)^H$  denotes the conjugate transpose of  $u(j)$ .

The computed eigenvectors are normalized to have Euclidean norm equal to 1 and largest component real.

### Input Parameters

*jobvl*      CHARACTER\*1. Must be '**N**' or '**V**'.  
 If *jobvl* = '**N**', then left eigenvectors of  $A$  are not computed.  
 If *jobvl* = '**V**', then left eigenvectors of  $A$  are computed.

<i>jobvr</i>	CHARACTER*1. Must be ' <b>N</b> ' or ' <b>V</b> '. If <i>jobvr</i> = ' <b>N</b> ', then right eigenvectors of <i>A</i> are not computed. If <i>jobvr</i> = ' <b>V</b> ', then right eigenvectors of <i>A</i> are computed.
<i>n</i>	INTEGER. The order of the matrix <i>A</i> ( <i>n</i> ≥ 0).
<i>a, work</i>	REAL for <b>sgeev</b> DOUBLE PRECISION for <b>dgeev</b> COMPLEX for <b>cgeev</b> DOUBLE COMPLEX for <b>zgeev</b> . Arrays: <i>a</i> ( <i>lda</i> ,*) is an array containing the <i>n</i> -by- <i>n</i> matrix <i>A</i> . The second dimension of <i>a</i> must be at least max(1, <i>n</i> ). <i>work</i> ( <i>lwork</i> ) is a workspace array.
<i>lda</i>	INTEGER. The first dimension of the array <i>a</i> . Must be at least max(1, <i>n</i> ).
<i>ldvl, ldvr</i>	INTEGER. The leading dimensions of the output arrays <i>vl</i> and <i>vr</i> , respectively. Constraints: <i>ldvl</i> ≥ 1 ; <i>ldvr</i> ≥ 1. If <i>jobvl</i> = ' <b>V</b> ', <i>ldvl</i> ≥ max(1, <i>n</i> ) ; If <i>jobvr</i> = ' <b>V</b> ', <i>ldvr</i> ≥ max(1, <i>n</i> ).
<i>lwork</i>	INTEGER. The dimension of the array <i>work</i> . Constraint: <i>lwork</i> ≥ max(1, 3 <i>n</i> ) , and if <i>jobvl</i> = ' <b>V</b> ' or <i>jobvr</i> = ' <b>V</b> ' , <i>lwork</i> ≥ max(1, 4 <i>n</i> ) (for real flavors); <i>lwork</i> ≥ max(1, 2 <i>n</i> ) (for complex flavors). For good performance, <i>lwork</i> must generally be larger.
<i>rwork</i>	REAL for <b>cgeev</b> DOUBLE PRECISION for <b>zgeev</b> Workspace array, DIMENSION at least max(1, 2 <i>n</i> ). Used in complex flavors only.

## Output Parameters

**a** On exit, this array is overwritten by intermediate results.

**wr, wi** **REAL** for **sgeev**  
**DOUBLE PRECISION** for **dgeev**

Arrays, **DIMENSION** at least max (1, **n**) each.

Contain the real and imaginary parts, respectively, of the computed eigenvalues. Complex conjugate pairs of eigenvalues appear consecutively with the eigenvalue having positive imaginary part first.

**w** **COMPLEX** for **cgeev**  
**DOUBLE COMPLEX** for **zgeev**.  
Array, **DIMENSION** at least max(1,**n**).  
Contains the computed eigenvalues.

**vl, vr** **REAL** for **sgeev**  
**DOUBLE PRECISION** for **dgeev**  
**COMPLEX** for **cgeev**  
**DOUBLE COMPLEX** for **zgeev**.

Arrays:

**vl( ldvl, \* )**; the second dimension of **vl** must be at least max(1, **n**).

If **jobvl = 'V'**, the left eigenvectors  $u(j)$  are stored one after another in the columns of **vl**, in the same order as their eigenvalues. If **jobvl = 'N'**, **vl** is not referenced.

*For real flavors:*

If the j-th eigenvalue is real, then  $u(j) = \text{vl}(:,j)$ , the j-th column of **vl**. If the j-th and (j+1)-st eigenvalues form a complex conjugate pair, then  $u(j) = \text{vl}(:,j) + i^* \text{vl}(:,j+1)$  and  $u(j+1) = \text{vl}(:,j) - i^* \text{vl}(:,j+1)$ , where  $i = \sqrt{-1}$ .

*For complex flavors:*

$u(j) = \text{vl}(:,j)$ , the j-th column of **vl**.

**vr( ldvr, \* )**; the second dimension of **vr** must be at least max(1, **n**).

If **jobvr = 'V'**, the right eigenvectors  $v(j)$  are stored one after another in the columns of **vr**, in the same order as their eigenvalues. If **jobvr = 'N'**, **vr** is not referenced.

*For real flavors:*

If the j-th eigenvalue is real, then  $v(j) = \text{vr}(:,j)$ , the j-th column of  $\text{vr}$ . If the j-th and (j+1)-st eigenvalues form a complex conjugate pair, then  $v(j) = \text{vr}(:,j) + i * \text{vr}(:,j+1)$  and  $v(j+1) = \text{vr}(:,j) - i * \text{vr}(:,j+1)$ , where  $i = \sqrt{-1}$ .

*For complex flavors:*

$v(j) = \text{vr}(:,j)$ , the j-th column of  $\text{vr}$ .

`work(1)`

On exit, if `info`=0, then `work(1)` returns the required minimal size of `lwork`.

`info`

`INTEGER`.

If `info`=0, the execution is successful.

If `info`=-*i*, the *i*th parameter had an illegal value.

If `info`=*i*, the QR algorithm failed to compute all the eigenvalues, and no eigenvectors have been computed; elements *i*+1:*n* of `wr` and `wi` (for real flavors) or `w` (for complex flavors) contain those eigenvalues which have converged.

## Application Notes

If you are in doubt how much workspace to supply for the array `work`, use a generous value of `lwork` for the first run. On exit, examine `work(1)` and use this value for subsequent runs.

## ?geevx

*Computes the eigenvalues and left and right eigenvectors of a general matrix, with preliminary matrix balancing, and computes reciprocal condition numbers for the eigenvalues and right eigenvectors.*

```
call sgeevx ( balanc, jobvl, jobvr, sense, n, a, lda, wr, wi, vl,
              ldvl, vr, ldvr, ilo, ihi, scale, abnrm, rconde,
              rcondv, work, lwork, iwork, info)
call dgeevx ( balanc, jobvl, jobvr, sense, n, a, lda, wr, wi, vl,
              ldvl, vr, ldvr, ilo, ihi, scale, abnrm, rconde,
              rcondv, work, lwork, iwork, info)
call cgeevx ( balanc, jobvl, jobvr, sense, n, a, lda, w, vl, ldvl,
              vr, ldvr, ilo, ihi, scale, abnrm, rconde, rcondv,
              work, lwork, rwork, info)
call zgeevx ( balanc, jobvl, jobvr, sense, n, a, lda, w, vl, ldvl,
              vr, ldvr, ilo, ihi, scale, abnrm, rconde, rcondv,
              work, lwork, rwork, info)
```

### Discussion

This routine computes for an  $n$ -by- $n$  real/complex nonsymmetric matrix  $A$ , the eigenvalues and, optionally, the left and/or right eigenvectors.

Optionally also, it computes a balancing transformation to improve the conditioning of the eigenvalues and eigenvectors (*ilo*, *ihi*, *scale*, and *abnrm*), reciprocal condition numbers for the eigenvalues (*rconde*), and reciprocal condition numbers for the right eigenvectors (*rcondv*).

The right eigenvector  $v(j)$  of  $A$  satisfies

$$A^* v(j) = \lambda(j)^* v(j)$$

where  $\lambda(j)$  is its eigenvalue.

The left eigenvector  $u(j)$  of  $A$  satisfies

$$u(j)^H A = \lambda(j)^* u(j)^H$$

where  $u(j)^H$  denotes the conjugate transpose of  $u(j)$ .

The computed eigenvectors are normalized to have Euclidean norm equal to 1 and largest component real.

Balancing a matrix means permuting the rows and columns to make it more nearly upper triangular, and applying a diagonal similarity transformation  $D A D^{-1}$ , where  $D$  is a diagonal matrix, to make its rows and columns closer in norm and the condition numbers of its eigenvalues and eigenvectors smaller. The computed reciprocal condition numbers correspond to the balanced matrix.

Permuting rows and columns will not change the condition numbers in exact arithmetic) but diagonal scaling will. For further explanation of balancing, see [[LUG](#)], Section 4.10.

## Input Parameters

*balanc*      CHARACTER\*1. Must be '**N**', '**P**', '**S**', or '**B**'.  
Indicates how the input matrix should be diagonally scaled and/or permuted to improve the conditioning of its eigenvalues.

If *balanc* = '**N**', do not diagonally scale or permute;  
If *balanc* = '**P**', perform permutations to make the matrix more nearly upper triangular. Do not diagonally scale;

If *balanc* = '**S**', Diagonally scale the matrix, i.e. replace  $A$  by  $D A D^{-1}$ , where  $D$  is a diagonal matrix chosen to make the rows and columns of  $A$  more equal in norm. Do not permute;

If *balanc* = '**B**', both diagonally scale and permute  $A$ .  
Computed reciprocal condition numbers will be for the matrix after balancing and/or permuting. Permuting does not change condition numbers (in exact arithmetic), but balancing does.

*jobvl*      CHARACTER\*1. Must be '**N**' or '**V**'.  
If *jobvl* = '**N**', left eigenvectors of  $A$  are not computed;  
If *jobvl* = '**V**', left eigenvectors of  $A$  are computed.  
If *sense* = '**E**' or '**B**', then *jobvl* must be '**V**'.

<i>jobvr</i>	CHARACTER*1. Must be ' <b>N</b> ' or ' <b>V</b> '. If <i>jobvr</i> = ' <b>N</b> ', right eigenvectors of <i>A</i> are not computed; If <i>jobvr</i> = ' <b>V</b> ', right eigenvectors of <i>A</i> are computed. If <i>sense</i> = ' <b>E</b> ' or ' <b>B</b> ', then <i>jobvr</i> must be ' <b>V</b> '.
<i>sense</i>	CHARACTER*1. Must be ' <b>N</b> ', ' <b>E</b> ', ' <b>V</b> ', or ' <b>B</b> '. Determines which reciprocal condition number are computed.  If <i>sense</i> = ' <b>N</b> ', none are computed; If <i>sense</i> = ' <b>E</b> ', computed for eigenvalues only; If <i>sense</i> = ' <b>V</b> ', computed for right eigenvectors only; If <i>sense</i> = ' <b>B</b> ', computed for eigenvalues and right eigenvectors.  If <i>sense</i> is ' <b>E</b> ' or ' <b>B</b> ', both left and right eigenvectors must also be computed ( <i>jobvl</i> = ' <b>V</b> ' and <i>jobvr</i> = ' <b>V</b> ').
<i>n</i>	INTEGER. The order of the matrix <i>A</i> ( <i>n</i> ≥ 0).
<i>a, work</i>	REAL for <i>sggevx</i> DOUBLE PRECISION for <i>dgeevx</i> COMPLEX for <i>cgeevx</i> DOUBLE COMPLEX for <i>zgeevx</i> .  Arrays: <i>a</i> ( <i>lda,*</i> ) is an array containing the <i>n</i> -by- <i>n</i> matrix <i>A</i> . The second dimension of <i>a</i> must be at least max(1, <i>n</i> ). <i>work</i> ( <i>lwork</i> ) is a workspace array.
<i>lda</i>	INTEGER. The first dimension of the array <i>a</i> . Must be at least max(1, <i>n</i> ).
<i>ldvl, ldvr</i>	INTEGER. The leading dimensions of the output arrays <i>vl</i> and <i>vr</i> , respectively. Constraints: <i>ldvl</i> ≥ 1 ; <i>ldvr</i> ≥ 1. If <i>jobvl</i> = ' <b>V</b> ', <i>ldvl</i> ≥ max(1, <i>n</i> ) ; If <i>jobvr</i> = ' <b>V</b> ', <i>ldvr</i> ≥ max(1, <i>n</i> ).
<i>lwork</i>	INTEGER. The dimension of the array <i>work</i> . <i>For real flavors:</i> If <i>sense</i> = ' <b>N</b> ' or ' <b>E</b> ', <i>lwork</i> ≥ max(1, 2 <i>n</i> ) , and

if  $\text{jobvl} = \text{'V'}$  or  $\text{jobvr} = \text{'V'}$ ,  $\text{lwork} \geq 3n$ ;  
 If  $\text{sense} = \text{'V'}$  or  $\text{'B'}$ ,  $\text{lwork} \geq n(n+6)$ .  
 For good performance,  $\text{lwork}$  must generally be larger.

*For complex flavors:*

If  $\text{sense} = \text{'N'}$  or  $\text{'E'}$ ,  $\text{lwork} \geq \max(1, 2n)$ ;  
 If  $\text{sense} = \text{'V'}$  or  $\text{'B'}$ ,  $\text{lwork} \geq n^2 + 2n$ .

For good performance,  $\text{lwork}$  must generally be larger.

*rwork*

**REAL** for **cgeevx**

**DOUBLE PRECISION** for **zgeevx**

Workspace array, **DIMENSION** at least  $\max(1, 2n)$ . Used in complex flavors only.

*iwork*

**INTEGER**.

Workspace array, **DIMENSION** at least  $\max(1, 2n-2)$ .

Used in real flavors only. Not referenced if  $\text{sense} = \text{'N'}$  or  $\text{'E'}$ .

## Output Parameters

*a*

On exit, this array is overwritten. If  $\text{jobvl} = \text{'V'}$  or  $\text{jobvr} = \text{'V'}$ , it contains the real-Shur/Shur form of the balanced version of the input matrix  $A$ .

*wr, wi*

**REAL** for **sggev**

**DOUBLE PRECISION** for **dgeev**

Arrays, **DIMENSION** at least max (1,  $n$ ) each.

Contain the real and imaginary parts, respectively, of the computed eigenvalues. Complex conjugate pairs of eigenvalues appear consecutively with the eigenvalue having positive imaginary part first.

*w*

**COMPLEX** for **cgeev**

**DOUBLE COMPLEX** for **zgeev**.

Array, **DIMENSION** at least  $\max(1, n)$ .

Contains the computed eigenvalues.

*vl, vr*

**REAL** for **sggev**

**DOUBLE PRECISION** for **dgeev**

**COMPLEX** for **cgeev**

**DOUBLE COMPLEX** for **zgeev**.

Arrays:

`vl( ldvl, * )`; the second dimension of `vl` must be at least  $\max(1, n)$ .

If `jobvl = 'V'`, the left eigenvectors  $u(j)$  are stored one after another in the columns of `vl`, in the same order as their eigenvalues. If `jobvl = 'N'`, `vl` is not referenced.

*For real flavors:*

If the j-th eigenvalue is real, then  $u(j) = \text{vl}(:,j)$ , the j-th column of `vl`. If the j-th and (j+1)-st eigenvalues form a complex conjugate pair, then  $u(j) = \text{vl}(:,j) + i^* \text{vl}(:,j+1)$  and  $u(j+1) = \text{vl}(:,j) - i^* \text{vl}(:,j+1)$ , where  $i = \sqrt{-1}$ .

*For complex flavors:*

$u(j) = \text{vl}(:,j)$ , the j-th column of `vl`.

`vr( ldvr, * )`; the second dimension of `vr` must be at least  $\max(1, n)$ .

If `jobvr = 'V'`, the right eigenvectors  $v(j)$  are stored one after another in the columns of `vr`, in the same order as their eigenvalues. If `jobvr = 'N'`, `vr` is not referenced.

*For real flavors:*

If the j-th eigenvalue is real, then  $v(j) = \text{vr}(:,j)$ , the j-th column of `vr`. If the j-th and (j+1)-st eigenvalues form a complex conjugate pair, then  $v(j) = \text{vr}(:,j) + i^* \text{vr}(:,j+1)$  and  $v(j+1) = \text{vr}(:,j) - i^* \text{vr}(:,j+1)$ , where  $i = \sqrt{-1}$ .

*For complex flavors:*

$v(j) = \text{vr}(:,j)$ , the j-th column of `vr`.

`ilo, ihi`

`INTEGER`.

`ilo` and `ihi` are integer values determined when  $A$  was balanced.

The balanced  $A(i,j) = 0$  if  $i > j$  and  $j = 1, \dots, \text{ilo}-1$  or  $i = \text{ihi}+1, \dots, n$ .

If `balanc = 'N'` or `'S'`, `ilo = 1` and `ihi = n`.

`scale`

`REAL` for single-precision flavors

`DOUBLE PRECISION` for double-precision flavors.

Array, `DIMENSION` at least  $\max(1, n)$ .

Details of the permutations and scaling factors applied

when balancing  $A$ . If  $P(j)$  is the index of the row and column interchanged with row and column  $j$ , and  $D(j)$  is the scaling factor applied to row and column  $j$ , then

$$\begin{aligned} \text{scale}(j) &= P(j), \quad \text{for } j = 1, \dots, \text{ilo}-1 \\ &= D(j), \quad \text{for } j = \text{ilo}, \dots, \text{ihi} \\ &= P(j) \quad \text{for } j = \text{ihi}+1, \dots, n. \end{aligned}$$

The order in which the interchanges are made is  $n$  to  $\text{ihi}+1$ , then 1 to  $\text{ilo}-1$ .

*abnrm*

**REAL** for single-precision flavors

**DOUBLE PRECISION** for double-precision flavors.

The one-norm of the balanced matrix (the maximum of the sum of absolute values of elements of any column).

*rconde, rcondv*

**REAL** for single precision flavors

**DOUBLE PRECISION** for double precision flavors.

Arrays, **DIMENSION** at least  $\max(1, n)$  each.

*rconde*( $j$ ) is the reciprocal condition number of the  $j$ -th eigenvalue.

*rcondv*( $j$ ) is the reciprocal condition number of the  $j$ -th right eigenvector.

*work(1)*

On exit, if  $\text{info} = 0$ , then *work(1)* returns the required minimal size of *lwork*.

*info*

**INTEGER**.

If  $\text{info} = 0$ , the execution is successful.

If  $\text{info} = -i$ , the  $i$ th parameter had an illegal value.

If  $\text{info} = i$ , the *QR* algorithm failed to compute all the eigenvalues, and no eigenvectors or condition numbers have been computed; elements  $1:\text{ilo}-1$  and  $i+1:n$  of *wr* and *wi* (for real flavors) or *w* (for complex flavors) contain eigenvalues which have converged.

## Application Notes

If you are in doubt how much workspace to supply for the array *work*, use a generous value of *lwork* for the first run. On exit, examine *work(1)* and use this value for subsequent runs.

## Singular Value Decomposition

This section describes LAPACK driver routines used for solving singular value problems. See also [computational routines](#) that can be called to solve these problems.

[Table 5-12](#) lists routines described in more detail below.

**Table 5-12 Driver Routines for Singular Value Decomposition**

Routine Name	Operation performed
<a href="#">?gesvd</a>	Computes the singular value decomposition of a general rectangular matrix.
<a href="#">?gesdd</a>	Computes the singular value decomposition of a general rectangular matrix using a divide and conquer method.
<a href="#">?gsvd</a>	Computes the generalized singular value decomposition of a pair of general rectangular matrices.

---

## ?gesvd

*Computes the singular value decomposition of a general rectangular matrix.*

---

```
call sgesvd ( jobu, jobvt, m, n, a, lda, s, u, ldu, vt, ldvt,
              work, lwork, info)
call dgesvd ( jobu, jobvt, m, n, a, lda, s, u, ldu, vt, ldvt,
              work, lwork, info)
call cgesvd ( jobu, jobvt, m, n, a, lda, s, u, ldu, vt, ldvt,
              work, lwork, rwork, info)
call zgesvd ( jobu, jobvt, m, n, a, lda, s, u, ldu, vt, ldvt,
              work, lwork, rwork, info)
```

### Discussion

This routine computes the singular value decomposition (SVD) of a real/complex  $m$ -by- $n$  matrix  $A$ , optionally computing the left and/or right singular vectors. The SVD is written

$$A = U \Sigma V^H$$

where  $\Sigma$  is an  $m$ -by- $n$  matrix which is zero except for its  $\min(m,n)$  diagonal elements,  $U$  is an  $m$ -by- $m$  orthogonal/unitary matrix, and  $V$  is an  $n$ -by- $n$  orthogonal/unitary matrix. The diagonal elements of  $\Sigma$  are the singular values of  $A$ ; they are real and non-negative, and are returned in descending order. The first  $\min(m,n)$  columns of  $U$  and  $V$  are the left and right singular vectors of  $A$ .

Note that the routine returns  $V^H$ , not  $V$ .

## Input Parameters

<i>jobu</i>	CHARACTER*1. Must be ' <b>A</b> ', ' <b>S</b> ', ' <b>O</b> ', or ' <b>N</b> '. Specifies options for computing all or part of the matrix $U$ .  If <i>jobu</i> = ' <b>A</b> ', all $m$ columns of $U$ are returned in the array <i>u</i> ; if <i>jobu</i> = ' <b>S</b> ', the first $\min(m,n)$ columns of $U$ (the left singular vectors) are returned in the array <i>u</i> ; if <i>jobu</i> = ' <b>O</b> ', the first $\min(m,n)$ columns of $U$ (the left singular vectors) are overwritten on the array <i>a</i> ; if <i>jobu</i> = ' <b>N</b> ', no columns of $U$ (no left singular vectors) are computed.
<i>jobvt</i>	CHARACTER*1. Must be ' <b>A</b> ', ' <b>S</b> ', ' <b>O</b> ', or ' <b>N</b> '. Specifies options for computing all or part of the matrix $V^H$ .  If <i>jobvt</i> = ' <b>A</b> ', all $n$ rows of $V^H$ are returned in the array <i>vt</i> ; if <i>jobvt</i> = ' <b>S</b> ', the first $\min(m,n)$ rows of $V^H$ (the right singular vectors) are returned in the array <i>vt</i> ; if <i>jobvt</i> = ' <b>O</b> ', the first $\min(m,n)$ rows of $V^H$ (the right singular vectors) are overwritten on the array <i>a</i> ; if <i>jobvt</i> = ' <b>N</b> ', no rows of $V^H$ (no right singular vectors) are computed.
<i>jobvt</i> and <i>jobu</i>	cannot both be ' <b>O</b> '.
<i>m</i>	INTEGER. The number of rows of the matrix $A$ ( $m \geq 0$ ).
<i>n</i>	INTEGER. The number of columns in $A$ ( $n \geq 0$ ).

<i>a, work</i>	<b>REAL</b> for <code>sgevd</code> <b>DOUBLE PRECISION</b> for <code>dgevd</code> <b>COMPLEX</b> for <code>cgevd</code> <b>DOUBLE COMPLEX</b> for <code>zgevd</code> . Arrays: <i>a( lda, * )</i> is an array containing the <i>m</i> -by- <i>n</i> matrix <i>A</i> . The second dimension of <i>a</i> must be at least $\max(1, n)$ . <i>work( lwork )</i> is a workspace array.
<i>lda</i>	<b>INTEGER</b> . The first dimension of the array <i>a</i> . Must be at least $\max(1, m)$ .
<i>ldu, ldvt</i>	<b>INTEGER</b> . The leading dimensions of the output arrays <i>u</i> and <i>vt</i> , respectively. Constraints: $ldu \geq 1 ; ldvt \geq 1$ . If <i>jobu</i> = 'S' or 'A', <i>ldu</i> $\geq m$ ; If <i>jobvt</i> = 'A', <i>ldvt</i> $\geq n$ ; If <i>jobvt</i> = 'S', <i>ldvt</i> $\geq \min(m, n)$ .
<i>lwork</i>	<b>INTEGER</b> . The dimension of the array <i>work</i> ; <i>lwork</i> $\geq 1$ . Constraints: $lwork \geq \max(3 * \min(m, n) + \max(m, n), 5 * \min(m, n))$ (for real flavors); $lwork \geq 2 * \min(m, n) + \max(m, n)$ (for complex flavors). For good performance, <i>lwork</i> must generally be larger.
<i>rwork</i>	<b>REAL</b> for <code>cgevd</code> <b>DOUBLE PRECISION</b> for <code>zgevd</code> Workspace array, <b>DIMENSION</b> at least $\max(1, 5 * \min(m, n))$ . Used in complex flavors only.

## Output Parameters

<i>a</i>	On exit, If <i>jobu</i> = 'O', <i>a</i> is overwritten with the first $\min(m, n)$ columns of <i>U</i> (the left singular vectors, stored columnwise); If <i>jobvt</i> = 'O', <i>a</i> is overwritten with the first $\min(m, n)$
----------	---

rows of  $V^H$  (the right singular vectors, stored rowwise);  
 If  $\text{jobu} \neq 'O'$  and  $\text{jobvt} \neq 'O'$ , the contents of  $a$  are destroyed.

*s*

**REAL** for single precision flavors

**DOUBLE PRECISION** for double precision flavors.

Array, **DIMENSION** at least  $\max(1, \min(m, n))$ .

Contains the singular values of  $A$  sorted so that  
 $s(i) \geq s(i+1)$ .

*u, vt*

**REAL** for **s gesvd**

**DOUBLE PRECISION** for **d gesvd**

**COMPLEX** for **c gesvd**

**DOUBLE COMPLEX** for **z gesvd**.

Arrays:

*u*(*lDU*, \*); the second dimension of *u* must be at least  $\max(1, m)$  if  $\text{jobu} = 'A'$ , and at least  $\max(1, \min(m, n))$  if  $\text{jobu} = 'S'$ .

If  $\text{jobu} = 'A'$ , *u* contains the *m*-by-*m* orthogonal/unitary matrix *U*.

If  $\text{jobu} = 'S'$ , *u* contains the first  $\min(m, n)$  columns of *U* (the left singular vectors, stored columnwise).

If  $\text{jobu} = 'N'$  or '*O*', *u* is not referenced.

*vt*(*ldvt*, \*); the second dimension of *vt* must be at least  $\max(1, n)$ .

If  $\text{jobvt} = 'A'$ , *vt* contains the *n*-by-*n* orthogonal/unitary matrix *V*<sup>H</sup>.

If  $\text{jobvt} = 'S'$ , *vt* contains the first  $\min(m, n)$  rows of *V*<sup>H</sup> (the right singular vectors, stored rowwise).

If  $\text{jobvt} = 'N'$  or '*O*', *vt* is not referenced.

*work*

On exit, if *info* = 0, then *work(1)* returns the required minimal size of *lwork*.

For real flavors:

If *info* > 0, *work*(2: $\min(m, n)$ ) contains the un converged superdiagonal elements of an upper bidiagonal matrix *B* whose diagonal is in *s* (not

necessarily sorted).  $B$  satisfies  $A = \mathbf{u} * B * \mathbf{v}^T$ , so it has the same singular values as  $A$ , and singular vectors related by  $\mathbf{u}$  and  $\mathbf{v}^T$ .

*rwork*

On exit (for complex flavors), if  $\text{info} > 0$ ,  
 $\text{rwork}(1:\min(m,n)-1)$  contains the unconverged superdiagonal elements of an upper bidiagonal matrix  $B$  whose diagonal is in  $\mathbf{s}$  (not necessarily sorted).  $B$  satisfies  $A = \mathbf{u} * B * \mathbf{v}^T$ , so it has the same singular values as  $A$ , and singular vectors related by  $\mathbf{u}$  and  $\mathbf{v}^T$ .

*info*

INTEGER.

If  $\text{info} = 0$ , the execution is successful.

If  $\text{info} = -i$ , the  $i$ th parameter had an illegal value.

If  $\text{info} = i$ , then if `?bdsqr` did not converge,  $i$  specifies how many superdiagonals of the intermediate bidiagonal form  $B$  did not converge to zero.

## Application Notes

If you are in doubt how much workspace to supply for the array *work*, use a generous value of *lwork* for the first run. On exit, examine *work(1)* and use this value for subsequent runs.

---

## ?gesdd

*Computes the singular value decomposition of a general rectangular matrix using a divide and conquer method.*

---

```
call sgesdd ( jobz, m, n, a, lda, s, u, ldu, vt, ldvt,
              work, lwork, iwork, info)
call dgesdd ( jobz, m, n, a, lda, s, u, ldu, vt, ldvt,
              work, lwork, iwork, info)
call cgesdd ( jobz, m, n, a, lda, s, u, ldu, vt, ldvt,
              work, lwork, rwork, iwork, info)
call zgesdd ( jobz, m, n, a, lda, s, u, ldu, vt, ldvt,
              work, lwork, rwork, iwork, info)
```

### Discussion

This routine computes the singular value decomposition (SVD) of a real/complex  $m$ -by- $n$  matrix  $A$ , optionally computing the left and/or right singular vectors. If singular vectors are desired, it uses a divide and conquer algorithm.

The SVD is written

$$A = U \Sigma V^H$$

where  $\Sigma$  is an  $m$ -by- $n$  matrix which is zero except for its  $\min(m,n)$  diagonal elements,  $U$  is an  $m$ -by- $m$  orthogonal/unitary matrix, and  $V$  is an  $n$ -by- $n$  orthogonal/unitary matrix. The diagonal elements of  $\Sigma$  are the singular values of  $A$ ; they are real and non-negative, and are returned in descending order. The first  $\min(m,n)$  columns of  $U$  and  $V$  are the left and right singular vectors of  $A$ .

Note that the routine returns  $V^H$ , not  $V$ .

### Input Parameters

<i>jobz</i>	<b>CHARACTER*1.</b> Must be ' <b>A</b> ', ' <b>S</b> ', ' <b>O</b> ', or ' <b>N</b> '. Specifies options for computing all or part of the matrix $U$ .
-------------	---

If  $\text{jobz} = \text{'A'}$ , all  $m$  columns of  $U$  and all  $n$  rows of  $V^T$  are returned in the arrays  $u$  and  $vt$ ;  
 if  $\text{jobz} = \text{'S'}$ , the first  $\min(m, n)$  columns of  $U$  and the first  $\min(m, n)$  rows of  $V^T$  are returned in the arrays  $u$  and  $vt$ ;  
 if  $\text{jobz} = \text{'O'}$ , then  
   if  $m \geq n$ , the first  $n$  columns of  $U$  are overwritten on the array  $a$  and all rows of  $V^T$  are returned in the array  $vt$ ;  
   if  $m < n$ , all columns of  $U$  are returned in the array  $u$  and the first  $m$  rows of  $V^T$  are overwritten in the array  $vt$ ;  
 if  $\text{jobz} = \text{'N'}$ , no columns of  $U$  or rows of  $V^T$  are computed.

$m$  INTEGER. The number of rows of the matrix  $A$  ( $m \geq 0$ ).

$n$  INTEGER. The number of columns in  $A$  ( $n \geq 0$ ).

$a, work$  REAL for `sgetrf`  
 DOUBLE PRECISION for `dgetrf`  
 COMPLEX for `cgetrf`  
 DOUBLE COMPLEX for `zgetrf`.

Arrays:

$a(1:m, *)$  is an array containing the  $m$ -by- $n$  matrix  $A$ .  
 The second dimension of  $a$  must be at least  $\max(1, n)$ .

$work(1:lwork)$  is a workspace array.

$lda$  INTEGER. The first dimension of the array  $a$ .  
 Must be at least  $\max(1, m)$ .

$lDU, ldvt$  INTEGER. The leading dimensions of the output arrays  $u$  and  $vt$ , respectively. Constraints:  
 $lDU \geq 1$ ;  $ldvt \geq 1$ .  
 If  $\text{jobz} = \text{'S'}$  or  $\text{'A'}$ , or  $\text{jobz} = \text{'O'}$  and  $m < n$ ,  
 then  $lDU \geq m$ ;  
 If  $\text{jobz} = \text{'A'}$  or  $\text{jobz} = \text{'O'}$  and  $m \geq n$ ,  
 then  $ldvt \geq n$ ;  
 If  $\text{jobz} = \text{'S'}$ ,  $ldvt \geq \min(m, n)$ .

<i>lwork</i>	<b>INTEGER.</b> The dimension of the array <i>work</i> ; <i>lwork</i> $\geq 1$ . See <i>Application Notes</i> for the suggested value of <i>lwork</i> .
<i>rwork</i>	<b>REAL</b> for <i>cgesdd</i> <b>DOUBLE PRECISION</b> for <i>zgesdd</i> Workspace array, <b>DIMENSION</b> at least $\max(1, 5 * \min(m, n))$ if <i>jobz</i> = 'N'. Otherwise, the dimension of <i>rwork</i> must be at least $5 * (\min(m, n))^2 +$ $7 * \min(m, n)$ . This array is used in complex flavors only.
<i>iwork</i>	<b>INTEGER.</b> Workspace array, <b>DIMENSION</b> at least $\max(1, 8 * \min(m, n))$ .

## Output Parameters

<i>a</i>	On exit: If <i>jobz</i> = 'O', then if <i>m</i> $\geq n, a is overwritten with thefirst n columns of U (the left singular vectors, storedcolumnwise). If m < n, a is overwritten with the first mrows of VT (the right singular vectors, stored rowwise);If jobz \neq 'O', the contents of a are destroyed.$
<i>s</i>	<b>REAL</b> for single precision flavors <b>DOUBLE PRECISION</b> for double precision flavors. Array, <b>DIMENSION</b> at least $\max(1, \min(m, n))$ . Contains the singular values of <i>A</i> sorted so that <i>s</i> (i) $\geq s(i+1).$
<i>u</i> , <i>vt</i>	<b>REAL</b> for <i>s gesdd</i> <b>DOUBLE PRECISION</b> for <i>d gesdd</i> <b>COMPLEX</b> for <i>c gesdd</i> <b>DOUBLE COMPLEX</b> for <i>z gesdd</i> . Arrays: <i>u</i> ( <i>ldu</i> , *); the second dimension of <i>u</i> must be at least $\max(1, m)$ if <i>jobz</i> = 'A' or <i>jobz</i> = 'O' and <i>m</i> < <i>n</i> . If <i>jobz</i> = 'S', the second dimension of <i>u</i> must be at least $\max(1, \min(m, n))$ .

If  $\text{jobz} = \text{'A'}$  or  $\text{jobz} = \text{'O'}$  and  $m < n$ ,  $u$  contains the  $m$ -by- $m$  orthogonal/unitary matrix  $U$ .

If  $\text{jobz} = \text{'S'}$ ,  $u$  contains the first  $\min(m, n)$  columns of  $U$  (the left singular vectors, stored columnwise).

If  $\text{jobz} = \text{'O'}$  and  $m \geq n$ , or  $\text{jobz} = \text{'N'}$ ,  $u$  is not referenced.

$\text{vt}(1:\text{ldvt}, :)$ ; the second dimension of  $\text{vt}$  must be at least  $\max(1, n)$ .

If  $\text{jobz} = \text{'A'}$  or  $\text{jobz} = \text{'O'}$  and  $m \geq n$ ,  $\text{vt}$  contains the  $n$ -by- $n$  orthogonal/unitary matrix  $V^T$ .

If  $\text{jobz} = \text{'S'}$ ,  $\text{vt}$  contains the first  $\min(m, n)$  rows of  $V^T$  (the right singular vectors, stored rowwise).

If  $\text{jobz} = \text{'O'}$  and  $m < n$ , or  $\text{jobz} = \text{'N'}$ ,  $\text{vt}$  is not referenced.

$\text{work}(1)$  On exit, if  $\text{info} = 0$ , then  $\text{work}(1)$  returns the required minimal size of  $\text{lwork}$ .

$\text{info}$  INTEGER.

If  $\text{info} = 0$ , the execution is successful.

If  $\text{info} = -i$ , the  $i$ th parameter had an illegal value.

If  $\text{info} = i$ , then  $\text{?bdsdc}$  did not converge, updating process failed.

## Application Notes

For real flavors:

If  $\text{jobz} = \text{'N'}$ ,  $\text{lwork} \geq 5 * \min(m, n)$ ;

If  $\text{jobz} = \text{'O'}$ ,  $\text{lwork} \geq 5 * (\min(m, n))^2 + \max(m, n) + 9 * \min(m, n)$ ;

If  $\text{jobz} = \text{'S'}$  or  $\text{'A'}$ ,  $\text{lwork} \geq 4 * (\min(m, n))^2 + \max(m, n) + 9 * \min(m, n)$ ;

For complex flavors:

If  $\text{jobz} = \text{'N'}$ ,  $\text{lwork} \geq 2 * \min(m, n) + \max(m, n)$ ;

If  $\text{jobz} = \text{'O'}$ ,  $\text{lwork} \geq 2 * (\min(m, n))^2 + \max(m, n) + 2 * \min(m, n)$ ;

If  $\text{jobz} = \text{'S'}$  or  $\text{'A'}$ ,  $\text{lwork} \geq (\min(m, n))^2 + \max(m, n) + 2 * \min(m, n)$ ;

For good performance,  $\text{lwork}$  should generally be larger.

If you are in doubt how much workspace to supply for the array  $\text{work}$ , use a generous value of  $\text{lwork}$  for the first run. On exit, examine  $\text{work}(1)$  and use this value for subsequent runs.

---

## ?ggsvd

*Computes the generalized singular value decomposition of a pair of general rectangular matrices.*

---

```
call sggsvd ( jobu, jobv, jobq, m, n, p, k, l, a, lda, b, ldb, alpha,
              beta, u, ldu, v, ldv, q, ldq, work, iwork, info)
call dggsvd ( jobu, jobv, jobq, m, n, p, k, l, a, lda, b, ldb, alpha,
              beta, u, ldu, v, ldv, q, ldq, work, iwork, info)
call cggsvd ( jobu, jobv, jobq, m, n, p, k, l, a, lda, b, ldb, alpha,
              beta, u, ldu, v, ldv, q, ldq, work, rwork, iwork, info)
call zggsvd ( jobu, jobv, jobq, m, n, p, k, l, a, lda, b, ldb, alpha,
              beta, u, ldu, v, ldv, q, ldq, work, rwork, iwork, info)
```

### Discussion

This routine computes the generalized singular value decomposition (GSVD) of an  $m$ -by- $n$  real/complex matrix  $A$  and  $p$ -by- $n$  real/complex matrix  $B$ :

$$U^H A Q = D_1 * (0 \ R), \quad V^H B Q = D_2 * (0 \ R),$$

where  $U$ ,  $V$  and  $Q$  are orthogonal/unitary matrices.

Let  $k+1$  = the effective numerical rank of the matrix  $(A^H, B^H)^H$ , then  $R$  is a  $(k+1)$ -by- $(k+1)$  nonsingular upper triangular matrix,  $D_1$  and  $D_2$  are  $m$ -by- $(k+1)$  and  $p$ -by- $(k+1)$  "diagonal" matrices and of the following structures, respectively:

If  $m-k-1 \geq 0$ ,

$$D_1 = \begin{matrix} & k & l \\ & I & 0 \\ l & 0 & C \\ m-k-l & 0 & 0 \end{matrix}$$

$$D_2 = \begin{matrix} & k & l \\ l & 0 & S \\ p-l & 0 & 0 \end{matrix}$$

$$( \begin{matrix} 0 & R \end{matrix} ) = \begin{matrix} n-k-l & k & l \\ k & \begin{pmatrix} 0 & R_{11} & R_{12} \\ 0 & 0 & R_{22} \end{pmatrix} \\ l \end{matrix}$$

where

$$\begin{aligned} C &= \text{diag}(\alpha(k+1), \dots, \alpha(k+l)) \\ S &= \text{diag}(\beta(k+1), \dots, \beta(k+l)) \\ C^2 + S^2 &= I \end{aligned}$$

$R$  is stored in  $a(1:k+1, n-k-l+1:n)$  on exit.

If  $m-k-l < 0$ ,

$$\begin{matrix} k & m-k & k+l-m \\ D_1 = & \begin{matrix} I & 0 & 0 \\ 0 & C & 0 \end{matrix} \\ m-k & & \end{matrix}$$

$$D_2 = \begin{matrix} k & m-k & k+l-m \\ m-k & \begin{matrix} 0 & S & 0 \\ 0 & 0 & I \\ p-l & 0 & 0 \end{matrix} \\ k+l-m & & \end{matrix}$$

$$( \begin{matrix} 0 & R \end{matrix} ) = \begin{matrix} n-k-l & k & m-k & k+l-m \\ k & \begin{pmatrix} 0 & R_{11} & R_{12} & R_{13} \\ 0 & 0 & R_{22} & R_{23} \\ 0 & 0 & 0 & R_{33} \end{pmatrix} \\ m-k \\ k+l-m \end{matrix}$$

where

$$\begin{aligned} C &= \text{diag}(\alpha(k+1), \dots, \alpha(m)), \\ S &= \text{diag}(\beta(k+1), \dots, \beta(m)), \\ C^2 + S^2 &= I \end{aligned}$$

On exit,  $\begin{pmatrix} R_{11}R_{12}R_{13} \\ 0 & R_{22}R_{23} \end{pmatrix}$  is stored in  $a(1:m, n-k-l+1:n)$  and  $R_{33}$  is stored in  $b(m-k+l, n+m-k-l+1:n)$ .

The routine computes  $C$ ,  $S$ ,  $R$ , and optionally the orthogonal/unitary transformation matrices  $U$ ,  $V$  and  $Q$ .

In particular, if  $B$  is an  $n$ -by- $n$  nonsingular matrix, then the GSVD of  $A$  and  $B$  implicitly gives the SVD of  $AB^{-1}$ :

$$AB^{-1} = U(D_1 D_2^{-1})V^H.$$

If  $(A^H, B^H)^H$  has orthonormal columns, then the GSVD of  $A$  and  $B$  is also equal to the CS decomposition of  $A$  and  $B$ . Furthermore, the GSVD can be used to derive the solution of the eigenvalue problem:

$$A^H A x = \lambda B^H B x.$$

## Input Parameters

<i>jobu</i>	CHARACTER*1. Must be ' <b>U</b> ' or ' <b>N</b> '. If <i>jobu</i> = ' <b>U</b> ', orthogonal/unitary matrix $U$ is computed. If <i>jobu</i> = ' <b>N</b> ', $U$ is not computed.
<i>jobv</i>	CHARACTER*1. Must be ' <b>V</b> ' or ' <b>N</b> '. If <i>jobv</i> = ' <b>V</b> ', orthogonal/unitary matrix $V$ is computed. If <i>jobv</i> = ' <b>N</b> ', $V$ is not computed.
<i>jobq</i>	CHARACTER*1. Must be ' <b>Q</b> ' or ' <b>N</b> '. If <i>jobq</i> = ' <b>Q</b> ', orthogonal/unitary matrix $Q$ is computed. If <i>jobq</i> = ' <b>N</b> ', $Q$ is not computed.
<i>m</i>	INTEGER. The number of rows of the matrix $A$ ( $m \geq 0$ ).
<i>n</i>	INTEGER. The number of columns of the matrices $A$ and $B$ ( $n \geq 0$ ).
<i>p</i>	INTEGER. The number of rows of the matrix $B$ ( $p \geq 0$ ).
<i>a</i> , <i>b</i> , <i>work</i>	REAL for <b>sggsvd</b> DOUBLE PRECISION for <b>dggsvd</b> COMPLEX for <b>cggsvd</b> DOUBLE COMPLEX for <b>zggsvd</b> .

Arrays:

$a(1da, *)$  contains the  $m$ -by- $n$  matrix  $A$ .

The second dimension of  $a$  must be at least  $\max(1, n)$ .

$b(1db, *)$  contains the  $p$ -by- $n$  matrix  $B$ .

The second dimension of  $b$  must be at least  $\max(1, n)$ .

$work(*)$  is a workspace array. The dimension of  $work$  must be at least  $\max(3n, m, p) + n$ .

$1da$	INTEGER. The first dimension of $a$ ; at least $\max(1, m)$ .
$1db$	INTEGER. The first dimension of $b$ ; at least $\max(1, p)$ .
$1du$	INTEGER. The first dimension of the array $u$ . $1du \geq \max(1, m)$ if $jobu = 'U'$ ; $1du \geq 1$ otherwise.
$1dv$	INTEGER. The first dimension of the array $v$ . $1dv \geq \max(1, p)$ if $jobv = 'V'$ ; $1dv \geq 1$ otherwise.
$1dq$	INTEGER. The first dimension of the array $q$ . $1dq \geq \max(1, n)$ if $jobq = 'Q'$ ; $1dq \geq 1$ otherwise.
$iwork$	INTEGER. Workspace array, DIMENSION at least $\max(1, n)$ .
$rwork$	REAL for $cggsvd$ DOUBLE PRECISION for $zggsvd$ . Workspace array, DIMENSION at least $\max(1, 2n)$ . Used in complex flavors only.

## Output Parameters

$k, l$	INTEGER. On exit, $k$ and $l$ specify the dimension of the subblocks. The sum $k+l$ is equal to the effective numerical rank of $(A^H, B^{H^H})$ .
$a$	On exit, $a$ contains the triangular matrix $R$ or part of $R$ .
$b$	On exit, $b$ contains part of the triangular matrix $R$ if $m-k-l < 0$ .
$alpha, beta$	REAL for single-precision flavors DOUBLE PRECISION for double-precision flavors. Arrays, DIMENSION at least $\max(1, n)$ each. Contain the generalized singular value pairs of $A$ and $B$ :

*alpha*(1:*k*) = 1,  
*beta*(1:*k*) = 0,  
 and if *m-k-l* ≥ 0,  
*alpha*(*k+1:k+1*) = *C*,  
*beta*(*k+1:k+1*) = *S*,  
 or if *m-k-l* < 0,  
*alpha*(*k+1:m*) = *C*, *alpha*(*m+1:k+1*) = 0  
*beta*(*k+1:m*) = *S*, *beta*(*m+1:k+1*) = 1

and

*alpha*(*k+l+1:n*) = 0  
*beta*(*k+l+1:n*) = 0.

*u, v, q*

REAL for sggsvd  
 DOUBLE PRECISION for dggsvd  
 COMPLEX for cggsvd  
 DOUBLE COMPLEX for zggsvd.

Arrays:

*u*(*ldu*, \*); the second dimension of *u* must be at least max(1, *m*).  
 If *jobu* = 'U', *u* contains the *m*-by-*m* orthogonal/unitary matrix *U*.

If *jobu* = 'N', *u* is not referenced.

*v*(*ldv*, \*); the second dimension of *v* must be at least max(1, *p*).  
 If *jobv* = 'V', *v* contains the *p*-by-*p* orthogonal/unitary matrix *V*.

If *jobv* = 'N', *v* is not referenced.

*q*(*ldq*, \*); the second dimension of *q* must be at least max(1, *n*).  
 If *jobq* = 'Q', *q* contains the *n*-by-*n* orthogonal/unitary matrix *Q*.  
 If *jobq* = 'N', *q* is not referenced.

*iwork*

On exit, *iwork* stores the sorting information.

*info*

INTEGER.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*th parameter had an illegal value.

If *info* = 1, the Jacobi-type procedure failed to converge. For further details, see subroutine [?tgsja](#).

## Generalized Symmetric Definite Eigenproblems

This section describes LAPACK driver routines used for solving generalized symmetric definite eigenproblems. See also [computational routines](#) that can be called to solve these problems.

[Table 5-13](#) lists routines described in more detail below.

**Table 5-13 Driver Routines for Solving Generalized Symmetric Definite Eigenproblems**

Routine Name	Operation performed
<a href="#">?sygv /?hegv</a>	Computes all eigenvalues and, optionally, eigenvectors of a real / complex generalized symmetric /Hermitian definite eigenproblem.
<a href="#">?sygvd /?hegvd</a>	Computes all eigenvalues and, optionally, eigenvectors of a real / complex generalized symmetric /Hermitian definite eigenproblem. If eigenvectors are desired, it uses a divide and conquer method.
<a href="#">?sygvx /?hegvx</a>	Computes selected eigenvalues and, optionally, eigenvectors of a real / complex generalized symmetric /Hermitian definite eigenproblem.
<a href="#">?spgv /?hpgv</a>	Computes all eigenvalues and, optionally, eigenvectors of a real / complex generalized symmetric /Hermitian definite eigenproblem with matrices in packed storage.
<a href="#">?spgvd /?hpgvd</a>	Computes all eigenvalues and, optionally, eigenvectors of a real / complex generalized symmetric /Hermitian definite eigenproblem with matrices in packed storage. If eigenvectors are desired, it uses a divide and conquer method.
<a href="#">?spgvx /?hpgvx</a>	Computes selected eigenvalues and, optionally, eigenvectors of a real / complex generalized symmetric /Hermitian definite eigenproblem with matrices in packed storage.
<a href="#">?sbgv /?hbgy</a>	Computes all eigenvalues and, optionally, eigenvectors of a real / complex generalized symmetric /Hermitian definite eigenproblem with banded matrices.
<a href="#">?sbgvd /?hbgyd</a>	Computes all eigenvalues and, optionally, eigenvectors of a real / complex generalized symmetric /Hermitian definite eigenproblem with banded matrices. If eigenvectors are desired, it uses a divide and conquer method.
<a href="#">?sbgvx /?hbgyx</a>	Computes selected eigenvalues and, optionally, eigenvectors of a real / complex generalized symmetric /Hermitian definite eigenproblem with banded matrices.

---

## ?sygv

*Computes all eigenvalues and, optionally, eigenvectors of a real generalized symmetric definite eigenproblem.*

---

```
call ssygv ( itype, jobz, uplo, n, a, lda, b, ldb, w, work,  
          lwork, info )  
call dsygv ( itype, jobz, uplo, n, a, lda, b, ldb, w, work,  
          lwork, info )
```

### Discussion

This routine computes all the eigenvalues, and optionally, the eigenvectors of a real generalized symmetric-definite eigenproblem, of the form

$$Ax = \lambda Bx, \quad ABx = \lambda x, \text{ or } BAx = \lambda x.$$

Here  $A$  and  $B$  are assumed to be symmetric and  $B$  is also positive definite.

### Input Parameters

<i>itype</i>	<b>INTEGER.</b> Must be 1 or 2 or 3. Specifies the problem type to be solved: if <i>itype</i> = 1, the problem type is $Ax = \lambda Bx$ ; if <i>itype</i> = 2, the problem type is $ABx = \lambda x$ ; if <i>itype</i> = 3, the problem type is $BAx = \lambda x$ .
<i>jobz</i>	<b>CHARACTER*1.</b> Must be ' <b>N</b> ' or ' <b>V</b> '. If <i>jobz</i> = ' <b>N</b> ', then compute eigenvalues only. If <i>jobz</i> = ' <b>V</b> ', then compute eigenvalues and eigenvectors.
<i>uplo</i>	<b>CHARACTER*1.</b> Must be ' <b>U</b> ' or ' <b>L</b> '. If <i>uplo</i> = ' <b>U</b> ', arrays <i>a</i> and <i>b</i> store the upper triangles of $A$ and $B$ ; If <i>uplo</i> = ' <b>L</b> ', arrays <i>a</i> and <i>b</i> store the lower triangles of $A$ and $B$ .
<i>n</i>	<b>INTEGER.</b> The order of the matrices $A$ and $B$ ( <i>n</i> $\geq 0$ ).

*a, b, work*    **REAL** for **ssygv**  
**DOUBLE PRECISION** for **dsygv**.

Arrays:

*a( lda, \* )* contains the upper or lower triangle of the symmetric matrix  $A$ , as specified by *uplo*.

The second dimension of *a* must be at least  $\max(1, n)$ .

*b( ldb, \* )* contains the upper or lower triangle of the symmetric positive definite matrix  $B$ , as specified by *uplo*.

The second dimension of *b* must be at least  $\max(1, n)$ .

*work( lwork )* is a workspace array.

*lda*    **INTEGER**. The first dimension of *a*; at least  $\max(1, n)$ .

*ldb*    **INTEGER**. The first dimension of *b*; at least  $\max(1, n)$ .

*lwork*    **INTEGER**. The dimension of the array *work*;  
 $lwork \geq \max(1, 3n-1)$ .

See *Application Notes* for the suggested value of *lwork*.

## Output Parameters

*a*    On exit, if *jobz* = 'V', then if *info* = 0, *a* contains the matrix  $Z$  of eigenvectors. The eigenvectors are normalized as follows:

if *itype* = 1 or 2,  $Z^T B Z = I$ ;  
if *itype* = 3,  $Z^T B^{-1} Z = I$ ;

If *jobz* = 'N', then on exit the upper triangle (if *uplo* = 'U') or the lower triangle (if *uplo* = 'L') of  $A$ , including the diagonal, is destroyed.

*b*    On exit, if *info*  $\leq n$ , the part of *b* containing the matrix is overwritten by the triangular factor  $U$  or  $L$  from the Cholesky factorization  $B = U^T U$  or  $B = L L^T$ .

*w*    **REAL** for **ssygv**  
**DOUBLE PRECISION** for **dsygv**.  
Array, **DIMENSION** at least  $\max(1, n)$ .  
If *info* = 0, contains the eigenvalues in ascending order.

<i>work(1)</i>	On exit, if <i>info</i> =0, then <i>work(1)</i> returns the required minimal size of <i>lwork</i> .
<i>info</i>	<i>INTEGER.</i> If <i>info</i> =0, the execution is successful. If <i>info</i> =- <i>i</i> , the <i>i</i> th argument had an illegal value. If <i>info</i> >0, <i>spotrf/dpotrf</i> and <i>ssyev/dsyev</i> returned an error code: If <i>info</i> = <i>i</i> ≤ <i>n</i> , <i>ssyev/dsyev</i> failed to converge, and <i>i</i> off-diagonal elements of an intermediate tridiagonal did not converge to zero; If <i>info</i> = <i>n+i</i> , for 1≤ <i>i</i> ≤ <i>n</i> , then the leading minor of order <i>i</i> of <i>B</i> is not positive-definite. The factorization of <i>B</i> could not be completed and no eigenvalues or eigenvectors were computed.

### Application Notes

For optimum performance use *lwork*≥(nb+2)\**n*, where *nb* is the blocksize for *ssytrd/dsytrd* returned by *ilaenv*.  
If you are in doubt how much workspace to supply for the array *work*, use a generous value of *lwork* for the first run. On exit, examine *work(1)* and use this value for subsequent runs.

## ?hegv

*Computes all eigenvalues and, optionally, eigenvectors of a complex generalized Hermitian definite eigenproblem.*

---

```
call chegv ( itype, jobz, uplo, n, a, lda, b, ldb, w, work,
            lwork, rwork, info )
call zhegv ( itype, jobz, uplo, n, a, lda, b, ldb, w, work,
            lwork, rwork, info )
```

### Discussion

This routine computes all the eigenvalues, and optionally, the eigenvectors of a complex generalized Hermitian-definite eigenproblem, of the form

$$Ax = \lambda Bx, \quad ABx = \lambda x, \text{ or } BAx = \lambda x.$$

Here  $A$  and  $B$  are assumed to be Hermitian and  $B$  is also positive definite.

### Input Parameters

<i>itype</i>	<b>INTEGER.</b> Must be 1 or 2 or 3. Specifies the problem type to be solved: if <i>itype</i> = 1, the problem type is $Ax = \lambda Bx$ ; if <i>itype</i> = 2, the problem type is $ABx = \lambda x$ ; if <i>itype</i> = 3, the problem type is $BAx = \lambda x$ .
<i>jobz</i>	<b>CHARACTER*1.</b> Must be ' <b>N</b> ' or ' <b>V</b> '. If <i>jobz</i> = ' <b>N</b> ', then compute eigenvalues only. If <i>jobz</i> = ' <b>V</b> ', then compute eigenvalues and eigenvectors.
<i>uplo</i>	<b>CHARACTER*1.</b> Must be ' <b>U</b> ' or ' <b>L</b> '. If <i>uplo</i> = ' <b>U</b> ', arrays <i>a</i> and <i>b</i> store the upper triangles of $A$ and $B$ ; If <i>uplo</i> = ' <b>L</b> ', arrays <i>a</i> and <i>b</i> store the lower triangles of $A$ and $B$ .
<i>n</i>	<b>INTEGER.</b> The order of the matrices $A$ and $B$ ( <i>n</i> $\geq 0$ ).

<i>a, b, work</i>	<small>COMPLEX for <code>chevg</code> DOUBLE COMPLEX for <code>zhegv</code>.</small>
Arrays:	
<i>a( lda, * )</i>	contains the upper or lower triangle of the Hermitian matrix $A$ , as specified by <code>uplo</code> . The second dimension of <i>a</i> must be at least $\max(1, n)$ .
<i>b( ldb, * )</i>	contains the upper or lower triangle of the Hermitian positive definite matrix $B$ , as specified by <code>uplo</code> . The second dimension of <i>b</i> must be at least $\max(1, n)$ .
<i>work( lwork )</i>	is a workspace array.
<i>lda</i>	<small>INTEGER. The first dimension of <i>a</i>; at least <math>\max(1, n)</math>.</small>
<i>ldb</i>	<small>INTEGER. The first dimension of <i>b</i>; at least <math>\max(1, n)</math>.</small>
<i>lwork</i>	<small>INTEGER. The dimension of the array <i>work</i>; <math>lwork \geq \max(1, 2n-1)</math>. See <i>Application Notes</i> for the suggested value of <i>lwork</i>.</small>
<i>rwork</i>	<small>REAL for <code>chevg</code> DOUBLE PRECISION for <code>zhegv</code>. Workspace array, DIMENSION at least <math>\max(1, 3n-2)</math>.</small>

## Output Parameters

<i>a</i>	On exit, if <code>jobz = 'V'</code> , then if <code>info = 0</code> , <i>a</i> contains the matrix $Z$ of eigenvectors. The eigenvectors are normalized as follows: if <code>itype = 1</code> or <code>2</code> , $Z^H B Z = I$ ; if <code>itype = 3</code> , $Z^H B^{-1} Z = I$ ; If <code>jobz = 'N'</code> , then on exit the upper triangle (if <code>uplo = 'U'</code> ) or the lower triangle (if <code>uplo = 'L'</code> ) of $A$ , including the diagonal, is destroyed.
<i>b</i>	On exit, if <code>info \leq n</code> , the part of <i>b</i> containing the matrix is overwritten by the triangular factor $U$ or $L$ from the Cholesky factorization $B = U^H U$ or $B = L L^H$ .

<i>w</i>	<small>REAL for <code>chegv</code> DOUBLE PRECISION for <code>zhegv</code>.</small>
	Array, <small>DIMENSION</small> at least $\max(1, n)$ . If <i>info</i> = 0, contains the eigenvalues in ascending order.
<i>work(1)</i>	On exit, if <i>info</i> = 0, then <i>work(1)</i> returns the required minimal size of <i>lwork</i> .
<i>info</i>	<small>INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = <math>-i</math>, the <i>i</i>th argument had an illegal value. If <i>info</i> &gt; 0, <code>cpotrf/zpotrf</code> and <code>cheev/zheev</code> returned an error code:  If <i>info</i> = <math>i \leq n</math>, <code>cheev/zheev</code> failed to converge, and <i>i</i> off-diagonal elements of an intermediate tridiagonal did not converge to zero; If <i>info</i> = <math>n + i</math>, for <math>1 \leq i \leq n</math>, then the leading minor of order <i>i</i> of <i>B</i> is not positive-definite. The factorization of <i>B</i> could not be completed and no eigenvalues or eigenvectors were computed.</small>

### Application Notes

For optimum performance use  $lwork \geq (nb+1)*n$ , where *nb* is the blocksize for `chetrdf/zhetrd` returned by `ilaenv`.  
If you are in doubt how much workspace to supply for the array *work*, use a generous value of *lwork* for the first run. On exit, examine *work(1)* and use this value for subsequent runs.

---

## ?sygvd

*Computes all eigenvalues and, optionally, eigenvectors of a real generalized symmetric definite eigenproblem. If eigenvectors are desired, it uses a divide and conquer method.*

---

```
call ssygvd ( itype, jobz, uplo, n, a, lda, b, ldb, w, work,
              lwork, iwork, liwork, info )
call dsygvd ( itype, jobz, uplo, n, a, lda, b, ldb, w, work,
              lwork, iwork, liwork, info )
```

### Discussion

This routine computes all the eigenvalues, and optionally, the eigenvectors of a real generalized symmetric-definite eigenproblem, of the form

$$Ax = \lambda Bx, \quad ABx = \lambda x, \text{ or } BAx = \lambda x.$$

Here  $A$  and  $B$  are assumed to be symmetric and  $B$  is also positive definite.

If eigenvectors are desired, it uses a divide and conquer algorithm.

### Input Parameters

<i>itype</i>	<b>INTEGER.</b> Must be 1 or 2 or 3. Specifies the problem type to be solved: if <i>itype</i> = 1, the problem type is $Ax = \lambda Bx$ ; if <i>itype</i> = 2, the problem type is $ABx = \lambda x$ ; if <i>itype</i> = 3, the problem type is $BAx = \lambda x$ .
<i>jobz</i>	<b>CHARACTER*1.</b> Must be ' <b>N</b> ' or ' <b>V</b> '. If <i>jobz</i> = ' <b>N</b> ', then compute eigenvalues only. If <i>jobz</i> = ' <b>V</b> ', then compute eigenvalues and eigenvectors.

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. If <i>uplo</i> = 'U', arrays <i>a</i> and <i>b</i> store the upper triangles of <i>A</i> and <i>B</i> ; If <i>uplo</i> = 'L', arrays <i>a</i> and <i>b</i> store the lower triangles of <i>A</i> and <i>B</i> .
<i>n</i>	INTEGER. The order of the matrices <i>A</i> and <i>B</i> ( <i>n</i> $\geq$ 0).
<i>a</i> , <i>b</i> , <i>work</i>	REAL for <i>ssygvd</i> DOUBLE PRECISION for <i>dsygvd</i> . Arrays: <i>a( lda, * )</i> contains the upper or lower triangle of the symmetric matrix <i>A</i> , as specified by <i>uplo</i> . The second dimension of <i>a</i> must be at least $\max(1, n)$ . <i>b( ldb, * )</i> contains the upper or lower triangle of the symmetric positive definite matrix <i>B</i> , as specified by <i>uplo</i> . The second dimension of <i>b</i> must be at least $\max(1, n)$ . <i>work( lwork )</i> is a workspace array.
<i>lda</i>	INTEGER. The first dimension of <i>a</i> ; at least $\max(1, n)$ .
<i>ldb</i>	INTEGER. The first dimension of <i>b</i> ; at least $\max(1, n)$ .
<i>lwork</i>	INTEGER. The dimension of the array <i>work</i> . Constraints: If <i>n</i> $\leq$ 1, <i>lwork</i> $\geq$ 1; If <i>jobz</i> = 'N' and <i>n</i> > 1, <i>lwork</i> $\geq 2n+1$ ; If <i>jobz</i> = 'V' and <i>n</i> > 1, <i>lwork</i> $\geq 2n^2+6n+1$ .
<i>iwork</i>	INTEGER. Workspace array, DIMENSION ( <i>liwork</i> ). Constraints: If <i>n</i> $\leq$ 1, <i>liwork</i> $\geq$ 1; If <i>jobz</i> = 'N' and <i>n</i> > 1, <i>liwork</i> $\geq 1$ ; If <i>jobz</i> = 'V' and <i>n</i> > 1, <i>liwork</i> $\geq 5n+3$ .
<i>liwork</i>	

## Output Parameters

- a* On exit, if  $\text{jobz} = \text{'V'}$ , then if  $\text{info} = 0$ , *a* contains the matrix  $Z$  of eigenvectors. The eigenvectors are normalized as follows:  
 if  $\text{itype} = 1$  or  $2$ ,  $Z^T B Z = I$ ;  
 if  $\text{itype} = 3$ ,  $Z^T B^{-1} Z = I$ ;  
 If  $\text{jobz} = \text{'N'}$ , then on exit the upper triangle (if  $\text{uplo} = \text{'U'}$ ) or the lower triangle (if  $\text{uplo} = \text{'L'}$ ) of *A*, including the diagonal, is destroyed.
- b* On exit, if  $\text{info} \leq n$ , the part of *b* containing the matrix is overwritten by the triangular factor *U* or *L* from the Cholesky factorization  $B = U^T U$  or  $B = L L^T$ .
- w* REAL for `ssygvd`  
 DOUBLE PRECISION for `dsygvd`.  
 Array, DIMENSION at least  $\max(1, n)$ .  
 If  $\text{info} = 0$ , contains the eigenvalues in ascending order.
- work(1)* On exit, if  $\text{info} = 0$ , then *work(1)* returns the required minimal size of *lwork*.
- iwork(1)* On exit, if  $\text{info} = 0$ , then *iwork(1)* returns the required minimal size of *liwork*.
- info* INTEGER.  
 If  $\text{info} = 0$ , the execution is successful.  
 If  $\text{info} = -i$ , the *i*th argument had an illegal value.  
 If  $\text{info} > 0$ , `spotrf/dpotrf` and `ssyev/dsyev` returned an error code:  
   If  $\text{info} = i \leq n$ , `ssyev/dsyev` failed to converge, and *i* off-diagonal elements of an intermediate tridiagonal did not converge to zero;  
   If  $\text{info} = n + i$ , for  $1 \leq i \leq n$ , then the leading minor of order *i* of *B* is not positive-definite. The factorization of *B* could not be completed and no eigenvalues or eigenvectors were computed.

## ?hegvd

*Computes all eigenvalues and, optionally, eigenvectors of a complex generalized Hermitian definite eigenproblem. If eigenvectors are desired, it uses a divide and conquer method.*

```
call chegvd ( itype, jobz, uplo, n, a, lda, b, ldb, w, work,
              lwork, rwork, lrwork, iwork, liwork, info )
call zhegvd ( itype, jobz, uplo, n, a, lda, b, ldb, w, work,
              lwork, rwork, lrwork, iwork, liwork, info )
```

### Discussion

This routine computes all the eigenvalues, and optionally, the eigenvectors of a complex generalized Hermitian-definite eigenproblem, of the form  
 $Ax = \lambda Bx$ ,  $ABx = \lambda x$ , or  $BAx = \lambda x$ .

Here  $A$  and  $B$  are assumed to be Hermitian and  $B$  is also positive definite.  
If eigenvectors are desired, it uses a divide and conquer algorithm.

### Input Parameters

*itype*            INTEGER. Must be 1 or 2 or 3.  
Specifies the problem type to be solved:  
if *itype* = 1, the problem type is  $Ax = \lambda Bx$ ;  
if *itype* = 2, the problem type is  $ABx = \lambda x$ ;  
if *itype* = 3, the problem type is  $BAx = \lambda x$ .

*jobz*            CHARACTER\*1. Must be '**N**' or '**V**'.  
If *jobz* = '**N**', then compute eigenvalues only.  
If *jobz* = '**V**', then compute eigenvalues and eigenvectors.

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. If <i>uplo</i> = 'U', arrays <i>a</i> and <i>b</i> store the upper triangles of <i>A</i> and <i>B</i> ; If <i>uplo</i> = 'L', arrays <i>a</i> and <i>b</i> store the lower triangles of <i>A</i> and <i>B</i> .
<i>n</i>	INTEGER. The order of the matrices <i>A</i> and <i>B</i> ( <i>n</i> ≥ 0).
<i>a</i> , <i>b</i> , <i>work</i>	COMPLEX for <b>chegvd</b> DOUBLE COMPLEX for <b>zhegvd</b> . Arrays: <i>a</i> ( <i>lda</i> ,*) contains the upper or lower triangle of the Hermitian matrix <i>A</i> , as specified by <i>uplo</i> . The second dimension of <i>a</i> must be at least max(1, <i>n</i> ). <i>b</i> ( <i>ldb</i> ,*) contains the upper or lower triangle of the Hermitian positive definite matrix <i>B</i> , as specified by <i>uplo</i> . The second dimension of <i>b</i> must be at least max(1, <i>n</i> ). <i>work</i> ( <i>lwork</i> ) is a workspace array.
<i>lda</i>	INTEGER. The first dimension of <i>a</i> ; at least max(1, <i>n</i> ).
<i>ldb</i>	INTEGER. The first dimension of <i>b</i> ; at least max(1, <i>n</i> ).
<i>lwork</i>	INTEGER. The dimension of the array <i>work</i> . Constraints: If <i>n</i> ≤ 1, <i>lwork</i> ≥ 1; If <i>jobz</i> = 'N' and <i>n</i> > 1, <i>lwork</i> ≥ <i>n</i> +1; If <i>jobz</i> = 'V' and <i>n</i> > 1, <i>lwork</i> ≥ <i>n</i> <sup>2</sup> +2 <i>n</i> .
<i>rwork</i>	REAL for <b>chegvd</b> DOUBLE PRECISION for <b>zhegvd</b> . Workspace array, DIMENSION ( <i>lrwork</i> ). .
<i>lrwork</i>	INTEGER. The dimension of the array <i>rwork</i> . Constraints: If <i>n</i> ≤ 1, <i>lrwork</i> ≥ 1; If <i>jobz</i> = 'N' and <i>n</i> > 1, <i>lrwork</i> ≥ <i>n</i> ; If <i>jobz</i> = 'V' and <i>n</i> > 1, <i>lrwork</i> ≥ 2 <i>n</i> <sup>2</sup> +5 <i>n</i> +1 .
<i>iwork</i>	INTEGER. Workspace array, DIMENSION ( <i>liwork</i> ). .

*liwork*      **INTEGER.** The dimension of the array *iwork*.

Constraints:

If  $n \leq 1$ ,  $liwork \geq 1$ ;

If  $jobz = 'N'$  and  $n > 1$ ,  $liwork \geq 1$ ;

If  $jobz = 'V'$  and  $n > 1$ ,  $liwork \geq 5n + 3$ .

## Output Parameters

*a*      On exit, if  $jobz = 'V'$ , then if  $info = 0$ , *a* contains the matrix *Z* of eigenvectors. The eigenvectors are

normalized as follows:

if  $itype = 1$  or  $2$ ,  $Z^H B Z = I$ ;

if  $itype = 3$ ,  $Z^H B^{-1} Z = I$ ;

If  $jobz = 'N'$ , then on exit the upper triangle (if  $uplo = 'U'$ ) or the lower triangle (if  $uplo = 'L'$ ) of *A*, including the diagonal, is destroyed.

*b*      On exit, if  $info \leq n$ , the part of *b* containing the matrix is overwritten by the triangular factor *U* or *L* from the Cholesky factorization  $B = U^H U$  or  $B = L L^H$ .

*w*      **REAL** for *chegvd*

**DOUBLE PRECISION** for *zhegvd*.

Array, **DIMENSION** at least  $\max(1, n)$ .

If  $info = 0$ , contains the eigenvalues in ascending order.

*work(1)*      On exit, if  $info = 0$ , then *work(1)* returns the required minimal size of *lwork*.

*rwork(1)*      On exit, if  $info = 0$ , then *rwork(1)* returns the required minimal size of *lrwork*.

*iwork(1)*      On exit, if  $info = 0$ , then *iwork(1)* returns the required minimal size of *liwork*.

*info*      **INTEGER.**

If  $info = 0$ , the execution is successful.

If  $info = -i$ , the *i*th argument had an illegal value.

If  $info > 0$ , *cpotrf/zpotrf* and *cheev/zheev* returned an error code:

If  $\text{info} = i \leq n$ , `cheev/zheev` failed to converge, and  $i$  off-diagonal elements of an intermediate tridiagonal did not converge to zero;  
If  $\text{info} = n + i$ , for  $1 \leq i \leq n$ , then the leading minor of order  $i$  of  $B$  is not positive-definite. The factorization of  $B$  could not be completed and no eigenvalues or eigenvectors were computed.

## ?sygvx

*Computes selected eigenvalues and, optionally, eigenvectors of a real generalized symmetric definite eigenproblem.*

```
call ssygvx(itype, jobz, range, uplo, n, a, lda, b, ldb, vl, vu, il,
            iu, abstol, m, w, z, ldz, work, lwork, iwork, ifail, info)
call dsygvx(itype, jobz, range, uplo, n, a, lda, b, ldb, vl, vu, il,
            iu, abstol, m, w, z, ldz, work, lwork, iwork, ifail, info)
```

### Discussion

This routine computes selected eigenvalues, and optionally, the eigenvectors of a real generalized symmetric-definite eigenproblem, of the form

$$Ax = \lambda Bx, \quad ABx = \lambda x, \text{ or } BAx = \lambda x.$$

Here  $A$  and  $B$  are assumed to be symmetric and  $B$  is also positive definite. Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

### Input Parameters

<i>itype</i>	INTEGER. Must be 1 or 2 or 3. Specifies the problem type to be solved: if <i>itype</i> = 1, the problem type is $Ax = \lambda Bx$ ; if <i>itype</i> = 2, the problem type is $ABx = \lambda x$ ; if <i>itype</i> = 3, the problem type is $BAx = \lambda x$ .
<i>jobz</i>	CHARACTER*1. Must be ' <b>N</b> ' or ' <b>V</b> '. If <i>jobz</i> = ' <b>N</b> ', then compute eigenvalues only. If <i>jobz</i> = ' <b>V</b> ', then compute eigenvalues and eigenvectors.
<i>range</i>	CHARACTER*1. Must be ' <b>A</b> ' or ' <b>V</b> ' or ' <b>I</b> '.

If `range = 'A'`, the routine computes all eigenvalues.

If `range = 'V'`, the routine computes eigenvalues  $\lambda_i$  in the half-open interval:  $vl < \lambda_i \leq vu$ .

If `range = 'I'`, the routine computes eigenvalues with indices `il` to `iu`.

`uplo`

CHARACTER\*1. Must be '`U`' or '`L`'.

If `uplo = 'U'`, arrays `a` and `b` store the upper triangles of `A` and `B`;

If `uplo = 'L'`, arrays `a` and `b` store the lower triangles of `A` and `B`.

`n`

INTEGER. The order of the matrices `A` and `B` (`n`  $\geq 0$ ).

`a, b, work`

REAL for `ssygvx`

DOUBLE PRECISION for `dsygvx`.

Arrays:

`a( lda, * )` contains the upper or lower triangle of the symmetric matrix `A`, as specified by `uplo`.

The second dimension of `a` must be at least `max(1, n)`.

`b( ldb, * )` contains the upper or lower triangle of the symmetric positive definite matrix `B`, as specified by `uplo`.

The second dimension of `b` must be at least `max(1, n)`.

`work( lwork )` is a workspace array.

`lda`

INTEGER. The first dimension of `a`; at least `max(1, n)`.

`ldb`

INTEGER. The first dimension of `b`; at least `max(1, n)`.

`vl, vu`

REAL for `ssygvx`

DOUBLE PRECISION for `dsygvx`.

If `range = 'V'`, the lower and upper bounds of the interval to be searched for eigenvalues.

Constraint: `vl < vu`.

If `range = 'A'` or '`I`', `vl` and `vu` are not referenced.

---

<i>il, iu</i>	<b>INTEGER.</b> If <i>range</i> = 'I', the indices in ascending order of the smallest and largest eigenvalues to be returned. Constraint: $1 \leq il \leq iu \leq n$ , if <i>n</i> > 0; <i>il</i> =1 and <i>iu</i> =0 if <i>n</i> = 0. If <i>range</i> = 'A' or 'V', <i>il</i> and <i>iu</i> are not referenced.
<i>abstol</i>	<b>REAL</b> for <i>ssygvx</i> <b>DOUBLE PRECISION</b> for <i>dsygvx</i> . The absolute error tolerance for the eigenvalues. See <i>Application Notes</i> for more information.
<i>ldz</i>	<b>INTEGER.</b> The leading dimension of the output array <i>z</i> . Constraints: $ldz \geq 1$ ; if <i>jobz</i> = 'V', <i>ldz</i> $\geq \max(1, n)$ .
<i>lwork</i>	<b>INTEGER.</b> The dimension of the array <i>work</i> ; $lwork \geq \max(1, 8n)$ . See <i>Application Notes</i> for the suggested value of <i>lwork</i> .
<i>iwork</i>	<b>INTEGER.</b> Workspace array, <b>DIMENSION</b> at least $\max(1, 5n)$ .

## Output Parameters

<i>a</i>	On exit, the upper triangle (if <i>uplo</i> = 'U') or the lower triangle (if <i>uplo</i> = 'L') of <i>A</i> , including the diagonal, is overwritten.
<i>b</i>	On exit, if $info \leq n$ , the part of <i>b</i> containing the matrix is overwritten by the triangular factor <i>U</i> or <i>L</i> from the Cholesky factorization $B = U^T U$ or $B = L L^T$ .
<i>m</i>	<b>INTEGER.</b> The total number of eigenvalues found, $0 \leq m \leq n$ . If <i>range</i> = 'A', <i>m</i> = <i>n</i> , and if <i>range</i> = 'I', <i>m</i> = <i>iu</i> - <i>il</i> +1.
<i>w, z</i>	<b>REAL</b> for <i>ssygvx</i> <b>DOUBLE PRECISION</b> for <i>dsygvx</i> . Arrays:

$w(*)$ , **DIMENSION** at least  $\max(1, n)$ .

The first  $m$  elements of  $w$  contain the selected eigenvalues in ascending order.

$z(ldz, *)$ . The second dimension of  $z$  must be at least  $\max(1, m)$ .

If  $jobz = 'V'$ , then if  $info = 0$ , the first  $m$  columns of  $z$  contain the orthonormal eigenvectors of the matrix  $A$  corresponding to the selected eigenvalues, with the  $i$ -th column of  $z$  holding the eigenvector associated with  $w(i)$ . The eigenvectors are normalized as follows:

if  $itype = 1$  or  $2$ ,  $Z^T B Z = I$ ;  
if  $itype = 3$ ,  $Z^T B^{-1} Z = I$ ;

If  $jobz = 'N'$ , then  $z$  is not referenced.

If an eigenvector fails to converge, then that column of  $z$  contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in  $ifail$ .

Note: you must ensure that at least  $\max(1, m)$  columns are supplied in the array  $z$ ; if  $range = 'V'$ , the exact value of  $m$  is not known in advance and an upper bound must be used.

**work(1)**

On exit, if  $info = 0$ , then **work(1)** returns the required minimal size of **lwork**.

**ifail**

**INTEGER**.

Array, **DIMENSION** at least  $\max(1, n)$ .

If  $jobz = 'V'$ , then if  $info = 0$ , the first  $m$  elements of **ifail** are zero; if  $info > 0$ , the **ifail** contains the indices of the eigenvectors that failed to converge.

If  $jobz = 'N'$ , then **ifail** is not referenced.

**info**

**INTEGER**.

If  $info = 0$ , the execution is successful.

If  $info = -i$ , the  $i$ th argument had an illegal value.

If  $info > 0$ , **spotrf/dpotrf** and **ssyevx/dsyevx** returned an error code:

If  $\text{info} = i \leq n$ , `ssyevx/dsyevx` failed to converge, and  $i$  eigenvectors failed to converge. Their indices are stored in the array `ifail`;  
If  $\text{info} = n + i$ , for  $1 \leq i \leq n$ , then the leading minor of order  $i$  of  $B$  is not positive-definite. The factorization of  $B$  could not be completed and no eigenvalues or eigenvectors were computed.

### Application Notes

An approximate eigenvalue is accepted as converged when it is determined to lie in an interval  $[a,b]$  of width less than or equal to

$\text{abstol} + \varepsilon * \max(|a|,|b|)$ , where  $\varepsilon$  is the machine precision. If `abstol` is less than or equal to zero, then  $\varepsilon * \|T\|_1$  will be used in its place, where  $T$  is the tridiagonal matrix obtained by reducing  $A$  to tridiagonal form.

Eigenvalues will be computed most accurately when `abstol` is set to twice the underflow threshold  $2 * ?lamch('S')$ , not zero. If this routine returns with `info > 0`, indicating that some eigenvectors did not converge, try setting `abstol` to  $2 * ?lamch('S')$ .

For optimum performance use  $\text{lwork} \geq (nb+3)*n$ , where  $nb$  is the blocksize for `ssytrd/dsytrd` returned by `ilaenv`.

If you are in doubt how much workspace to supply for the array `work`, use a generous value of `lwork` for the first run. On exit, examine `work(1)` and use this value for subsequent runs.

---

## ?hegvx

*Computes selected eigenvalues and, optionally, eigenvectors of a complex generalized Hermitian definite eigenproblem.*

---

```
call chegvx ( itype, jobz, range, uplo, n, a, lda, b, ldb, vl, vu,
il, iu, abstol, m, w, z, ldz, work, lwork, rwork,
iwork, ifail, info)
call zhegvx ( itype, jobz, range, uplo, n, a, lda, b, ldb, vl, vu,
il, iu, abstol, m, w, z, ldz, work, lwork, rwork,
iwork, ifail, info)
```

### Discussion

This routine computes selected eigenvalues, and optionally, the eigenvectors of a complex generalized Hermitian-definite eigenproblem, of the form  
 $Ax = \lambda Bx$ ,  $ABx = \lambda x$ , or  $BAx = \lambda x$ .

Here  $A$  and  $B$  are assumed to be Hermitian and  $B$  is also positive definite. Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

### Input Parameters

<i>itype</i>	INTEGER. Must be 1 or 2 or 3. Specifies the problem type to be solved: if <i>itype</i> = 1, the problem type is $Ax = \lambda Bx$ ; if <i>itype</i> = 2, the problem type is $ABx = \lambda x$ ; if <i>itype</i> = 3, the problem type is $BAx = \lambda x$ .
<i>jobz</i>	CHARACTER*1. Must be ' <b>N</b> ' or ' <b>V</b> '. If <i>jobz</i> = ' <b>N</b> ', then compute eigenvalues only. If <i>jobz</i> = ' <b>V</b> ', then compute eigenvalues and eigenvectors.
<i>range</i>	CHARACTER*1. Must be ' <b>A</b> ' or ' <b>V</b> ' or ' <b>I</b> '.

If `range = 'A'`, the routine computes all eigenvalues.  
 If `range = 'V'`, the routine computes eigenvalues  $\lambda_i$  in  
 the half-open interval:  $vl < \lambda_i \leq vu$ .  
 If `range = 'I'`, the routine computes eigenvalues with  
 indices `il` to `iu`.

<code>uplo</code>	CHARACTER*1. Must be ' <code>U</code> ' or ' <code>L</code> '. If <code>uplo = 'U'</code> , arrays <code>a</code> and <code>b</code> store the upper triangles of <code>A</code> and <code>B</code> ; If <code>uplo = 'L'</code> , arrays <code>a</code> and <code>b</code> store the lower triangles of <code>A</code> and <code>B</code> .
<code>n</code>	INTEGER. The order of the matrices <code>A</code> and <code>B</code> ( $n \geq 0$ ).
<code>a, b, work</code>	COMPLEX for <code>chegvx</code> DOUBLE COMPLEX for <code>zhegvx</code> . Arrays: <code>a( lda, * )</code> contains the upper or lower triangle of the Hermitian matrix <code>A</code> , as specified by <code>uplo</code> . The second dimension of <code>a</code> must be at least <code>max(1, n)</code> . <code>b( ldb, * )</code> contains the upper or lower triangle of the Hermitian positive definite matrix <code>B</code> , as specified by <code>uplo</code> . The second dimension of <code>b</code> must be at least <code>max(1, n)</code> . <code>work( lwork )</code> is a workspace array.
<code>lda</code>	INTEGER. The first dimension of <code>a</code> ; at least <code>max(1, n)</code> .
<code>ldb</code>	INTEGER. The first dimension of <code>b</code> ; at least <code>max(1, n)</code> .
<code>vl, vu</code>	REAL for <code>chegvx</code> DOUBLE PRECISION for <code>zhegvx</code> . If <code>range = 'V'</code> , the lower and upper bounds of the interval to be searched for eigenvalues. Constraint: $vl < vu$ . If <code>range = 'A'</code> or ' <code>I</code> ', <code>vl</code> and <code>vu</code> are not referenced.

<i>il, iu</i>	<b>INTEGER.</b> If <i>range</i> = 'I', the indices in ascending order of the smallest and largest eigenvalues to be returned. Constraint: $1 \leq il \leq iu \leq n$ , if <i>n</i> > 0; <i>il</i> =1 and <i>iu</i> =0 if <i>n</i> = 0.
	If <i>range</i> = 'A' or 'V', <i>il</i> and <i>iu</i> are not referenced.
<i>abstol</i>	<b>REAL</b> for <i>chegvx</i> <b>DOUBLE PRECISION</b> for <i>zhegvx</i> . The absolute error tolerance for the eigenvalues. See <i>Application Notes</i> for more information.
<i>ldz</i>	<b>INTEGER.</b> The leading dimension of the output array <i>z</i> . Constraints: $ldz \geq 1$ ; if <i>jobz</i> = 'V', <i>ldz</i> $\geq \max(1, n)$ .
<i>lwork</i>	<b>INTEGER.</b> The dimension of the array <i>work</i> ; $lwork \geq \max(1, 2n-1)$ . See <i>Application Notes</i> for the suggested value of <i>lwork</i> .
<i>rwork</i>	<b>REAL</b> for <i>chegvx</i> <b>DOUBLE PRECISION</b> for <i>zhegvx</i> . Workspace array, <b>DIMENSION</b> at least $\max(1, 7n)$ .
<i>iwork</i>	<b>INTEGER.</b> Workspace array, <b>DIMENSION</b> at least $\max(1, 5n)$ .

## Output Parameters

<i>a</i>	On exit, the upper triangle (if <i>uplo</i> = 'U') or the lower triangle (if <i>uplo</i> = 'L') of <i>A</i> , including the diagonal, is overwritten.
<i>b</i>	On exit, if $info \leq n$ , the part of <i>b</i> containing the matrix is overwritten by the triangular factor <i>U</i> or <i>L</i> from the Cholesky factorization $B = U^H U$ or $B = L L^H$ .
<i>m</i>	<b>INTEGER.</b> The total number of eigenvalues found, $0 \leq m \leq n$ . If <i>range</i> = 'A', <i>m</i> = <i>n</i> , and if <i>range</i> = 'I', <i>m</i> = <i>iu</i> - <i>il</i> +1.

---

<i>w</i>	<small>REAL for <code>chegvx</code> DOUBLE PRECISION for <code>zhegvx</code>.</small>
	<small>Array, DIMENSION at least <math>\max(1, n)</math>. The first <math>m</math> elements of <i>w</i> contain the selected eigenvalues in ascending order.</small>
<i>z</i>	<small>COMPLEX for <code>chegvx</code> DOUBLE COMPLEX for <code>zhegvx</code>.</small>
	<small>Array <math>z(ldz, *)</math>. The second dimension of <i>z</i> must be at least <math>\max(1, m)</math>. If <math>jobz = 'V'</math>, then if <i>info</i> = 0, the first <math>m</math> columns of <i>z</i> contain the orthonormal eigenvectors of the matrix <i>A</i> corresponding to the selected eigenvalues, with the <math>i</math>-th column of <i>z</i> holding the eigenvector associated with <i>w(i)</i>. The eigenvectors are normalized as follows: if <i>itype</i> = 1 or 2, <math>Z^H B Z = I</math>; if <i>itype</i> = 3, <math>Z^H B^{-1} Z = I</math>;</small>
	<small>If <math>jobz = 'N'</math>, then <i>z</i> is not referenced. If an eigenvector fails to converge, then that column of <i>z</i> contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in <i>ifail</i>. Note: you must ensure that at least <math>\max(1, m)</math> columns are supplied in the array <i>z</i>; if <i>range</i> = 'V', the exact value of <math>m</math> is not known in advance and an upper bound must be used.</small>
<i>work(1)</i>	<small>On exit, if <i>info</i> = 0, then <i>work(1)</i> returns the required minimal size of <i>lwork</i>.</small>
<i>ifail</i>	<small>INTEGER. Array, DIMENSION at least <math>\max(1, n)</math>. If <math>jobz = 'V'</math>, then if <i>info</i> = 0, the first <math>m</math> elements of <i>ifail</i> are zero; if <i>info</i> &gt; 0, the <i>ifail</i> contains the indices of the eigenvectors that failed to converge. If <math>jobz = 'N'</math>, then <i>ifail</i> is not referenced.</small>

---

*info*

INTEGER.

If *info* = 0, the execution is successful.If *info* = -*i*, the *i*th argument had an illegal value.If *info* > 0, **cpotrf/zpotrf** and **cheevx/zheevx** returned an error code:If *info* = *i* ≤ *n*, **cheevx/zheevx** failed to converge, and *i* eigenvectors failed to converge. Their indices are stored in the array *ifail*;If *info* = *n* + *i*, for  $1 \leq i \leq n$ , then the leading minor of order *i* of *B* is not positive-definite. The factorization of *B* could not be completed and no eigenvalues or eigenvectors were computed.

## Application Notes

An approximate eigenvalue is accepted as converged when it is determined to lie in an interval [a,b] of width less than or equal to

*abstol* +  $\epsilon * \max(|a|, |b|)$ , where  $\epsilon$  is the machine precision. If *abstol* is less than or equal to zero, then  $\epsilon * \|T\|_1$  will be used in its place, where *T* is the tridiagonal matrix obtained by reducing *A* to tridiagonal form.

Eigenvalues will be computed most accurately when *abstol* is set to twice the underflow threshold  $2 * ?lamch('S')$ , not zero. If this routine returns with *info* > 0, indicating that some eigenvectors did not converge, try setting *abstol* to  $2 * ?lamch('S')$ .

For optimum performance use *lwork* ≥ (*nb*+1)\**n*, where *nb* is the blocksize for **chetrd/zhetrd** returned by **ilaenv**.

If you are in doubt how much workspace to supply for the array *work*, use a generous value of *lwork* for the first run. On exit, examine *work(1)* and use this value for subsequent runs.

## ?spgv

*Computes all eigenvalues and, optionally, eigenvectors of a real generalized symmetric definite eigenproblem with matrices in packed storage.*

```
call sspgv ( itype, jobz, uplo, n, ap, bp, w, z, ldz, work, info )
call dspgv ( itype, jobz, uplo, n, ap, bp, w, z, ldz, work, info )
```

### Discussion

This routine computes all the eigenvalues, and optionally, the eigenvectors of a real generalized symmetric-definite eigenproblem, of the form

$$Ax = \lambda Bx, \quad ABx = \lambda x, \text{ or } BAx = \lambda x.$$

Here  $A$  and  $B$  are assumed to be symmetric, stored in packed format, and  $B$  is also positive definite.

### Input Parameters

- itype*      **INTEGER.** Must be 1 or 2 or 3.  
Specifies the problem type to be solved:  
if *itype* = 1, the problem type is  $Ax = \lambda Bx$ ;  
if *itype* = 2, the problem type is  $ABx = \lambda x$ ;  
if *itype* = 3, the problem type is  $BAx = \lambda x$ .
- jobz*      **CHARACTER\*1.** Must be '**N**' or '**V**'.  
If *jobz* = '**N**', then compute eigenvalues only.  
If *jobz* = '**V**', then compute eigenvalues and eigenvectors.
- uplo*      **CHARACTER\*1.** Must be '**U**' or '**L**'.  
If *uplo* = '**U**', arrays *ap* and *bp* store the upper triangles of  $A$  and  $B$ ;  
If *uplo* = '**L**', arrays *ap* and *bp* store the lower triangles of  $A$  and  $B$ .
- n*      **INTEGER.** The order of the matrices  $A$  and  $B$  ( $n \geq 0$ ).

*ap, bp, work*    **REAL** for **sspgv**  
**DOUBLE PRECISION** for **dspgv**.

Arrays:

*ap(\*)* contains the packed upper or lower triangle of the symmetric matrix  $A$ , as specified by *uplo*. The dimension of *ap* must be at least  $\max(1, n*(n+1)/2)$ .

*bp(\*)* contains the packed upper or lower triangle of the symmetric matrix  $B$ , as specified by *uplo*. The dimension of *bp* must be at least  $\max(1, n*(n+1)/2)$ .

*work(\*)* is a workspace array, **DIMENSION** at least  $\max(1, 3n)$ .

*ldz*              **INTEGER**. The leading dimension of the output array *z*; *ldz*  $\geq 1$ . If *jobz* = 'V', *ldz*  $\geq \max(1, n)$ .

## Output Parameters

*ap*              On exit, the contents of *ap* are overwritten.

*bp*              On exit, contains the triangular factor  $U$  or  $L$  from the Cholesky factorization  $B = U^T U$  or  $B = L L^T$ , in the same storage format as  $B$ .

*w, z*            **REAL** for **sspgv**  
**DOUBLE PRECISION** for **dspgv**.

Arrays:

*w(\*)*, **DIMENSION** at least  $\max(1, n)$ .  
If *info* = 0, contains the eigenvalues in ascending order.

*z(ldz, \*)*. The second dimension of *z* must be at least  $\max(1, n)$ .  
If *jobz* = 'V', then if *info* = 0, *z* contains the matrix  $Z$  of eigenvectors. The eigenvectors are normalized as follows:  
if *itype* = 1 or 2,  $Z^T B Z = I$ ;  
if *itype* = 3,  $Z^T B^{-1} Z = I$ ;

If *jobz* = 'N', then *z* is not referenced.

*info*

INTEGER.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*th argument had an illegal value.

If *info* > 0, **spptrf/dpptrf** and **sspev/dspev** returned an error code:

If *info* = *i* ≤ *n*, **sspev/dspev** failed to converge, and *i* off-diagonal elements of an intermediate tridiagonal did not converge to zero;

If *info* = *n* + *i*, for  $1 \leq i \leq n$ , then the leading minor of order *i* of *B* is not positive-definite. The factorization of *B* could not be completed and no eigenvalues or eigenvectors were computed.

---

## ?hpgv

*Computes all eigenvalues and, optionally, eigenvectors of a complex generalized Hermitian definite eigenproblem with matrices in packed storage.*

---

```
call chpgv ( itype, jobz, uplo, n, ap, bp, w, z, ldz, work, rwork,
            info )
call zhpgv ( itype, jobz, uplo, n, ap, bp, w, z, ldz, work, rwork,
            info )
```

### Discussion

This routine computes all the eigenvalues, and optionally, the eigenvectors of a complex generalized Hermitian-definite eigenproblem, of the form

$$Ax = \lambda Bx, \quad ABx = \lambda x, \text{ or } BAx = \lambda x.$$

Here  $A$  and  $B$  are assumed to be Hermitian, stored in packed format, and  $B$  is also positive definite.

### Input Parameters

<i>itype</i>	<b>INTEGER.</b> Must be 1 or 2 or 3. Specifies the problem type to be solved: if <i>itype</i> = 1, the problem type is $Ax = \lambda Bx$ ; if <i>itype</i> = 2, the problem type is $ABx = \lambda x$ ; if <i>itype</i> = 3, the problem type is $BAx = \lambda x$ .
<i>jobz</i>	<b>CHARACTER*1.</b> Must be ' <b>N</b> ' or ' <b>V</b> '. If <i>jobz</i> = ' <b>N</b> ', then compute eigenvalues only. If <i>jobz</i> = ' <b>V</b> ', then compute eigenvalues and eigenvectors.
<i>uplo</i>	<b>CHARACTER*1.</b> Must be ' <b>U</b> ' or ' <b>L</b> '. If <i>uplo</i> = ' <b>U</b> ', arrays <i>ap</i> and <i>bp</i> store the upper triangles of $A$ and $B$ ; If <i>uplo</i> = ' <b>L</b> ', arrays <i>ap</i> and <i>bp</i> store the lower triangles of $A$ and $B$ .

---

<i>n</i>	<b>INTEGER.</b> The order of the matrices <i>A</i> and <i>B</i> ( <i>n</i> $\geq 0$ ).
<i>ap</i> , <i>bp</i> , <i>work</i>	<b>COMPLEX</b> for <b>chpgv</b> <b>DOUBLE COMPLEX</b> for <b>zhpgv</b> . Arrays: <i>ap</i> (*) contains the packed upper or lower triangle of the Hermitian matrix <i>A</i> , as specified by <i>uplo</i> . The dimension of <i>ap</i> must be at least $\max(1, n^*(n+1)/2)$ . <i>bp</i> (*) contains the packed upper or lower triangle of the Hermitian matrix <i>B</i> , as specified by <i>uplo</i> . The dimension of <i>bp</i> must be at least $\max(1, n^*(n+1)/2)$ . <i>work</i> (*) is a workspace array, <b>DIMENSION</b> at least $\max(1, 2n-1)$ .
<i>ldz</i>	<b>INTEGER.</b> The leading dimension of the output array <i>z</i> ; <i>ldz</i> $\geq 1$ . If <i>jobz</i> = 'V', <i>ldz</i> $\geq \max(1, n)$ .
<i>rwork</i>	<b>REAL</b> for <b>chpgv</b> <b>DOUBLE PRECISION</b> for <b>zhpgv</b> . Workspace array, <b>DIMENSION</b> at least $\max(1, 3n-2)$ .

## Output Parameters

<i>ap</i>	On exit, the contents of <i>ap</i> are overwritten.
<i>bp</i>	On exit, contains the triangular factor <i>U</i> or <i>L</i> from the Cholesky factorization $B = U^H U$ or $B = L L^H$ , in the same storage format as <i>B</i> .
<i>w</i>	<b>REAL</b> for <b>chpgv</b> <b>DOUBLE PRECISION</b> for <b>zhpgv</b> . Array, <b>DIMENSION</b> at least $\max(1, n)$ . If <i>info</i> = 0, contains the eigenvalues in ascending order.
<i>z</i>	<b>COMPLEX</b> for <b>chpgv</b> <b>DOUBLE COMPLEX</b> for <b>zhpgv</b> . Array <i>z</i> ( <i>ldz</i> , *). The second dimension of <i>z</i> must be at least $\max(1, n)$ . If <i>jobz</i> = 'V', then if <i>info</i> = 0, <i>z</i> contains the matrix <i>Z</i> of eigenvectors. The eigenvectors are normalized as

follows:

if *itype* = 1 or 2,  $Z^H B Z = I$ ;  
if *itype* = 3,  $Z^H B^{-1} Z = I$ ;

If *jobz* = 'N', then *z* is not referenced.

*info*

INTEGER.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*th argument had an illegal value.

If *info* > 0, **cpptrf/zpptrf** and **chpev/zhpev** returned an error code:

If *info* = *i* ≤ *n*, **chpev/zhpev** failed to converge, and *i* off-diagonal elements of an intermediate tridiagonal did not converge to zero;

If *info* = *n* + *i*, for  $1 \leq i \leq n$ , then the leading minor of order *i* of *B* is not positive-definite. The factorization of *B* could not be completed and no eigenvalues or eigenvectors were computed.

## ?spgvd

*Computes all eigenvalues and, optionally, eigenvectors of a real generalized symmetric definite eigenproblem with matrices in packed storage. If eigenvectors are desired, it uses a divide and conquer method.*

---

```
call sspgvd ( itype, jobz, uplo, n, ap, bp, w, z, ldz, work, lwork,
              iwork, liwork, info )
call dspgvd ( itype, jobz, uplo, n, ap, bp, w, z, ldz, work, lwork,
              iwork, liwork, info )
```

### Discussion

This routine computes all the eigenvalues, and optionally, the eigenvectors of a real generalized symmetric-definite eigenproblem, of the form

$$Ax = \lambda Bx, \quad ABx = \lambda x, \text{ or } BAx = \lambda x.$$

Here  $A$  and  $B$  are assumed to be symmetric, stored in packed format, and  $B$  is also positive definite. If eigenvectors are desired, it uses a divide and conquer algorithm.

### Input Parameters

<i>itype</i>	INTEGER. Must be 1 or 2 or 3. Specifies the problem type to be solved: if <i>itype</i> = 1, the problem type is $Ax = \lambda Bx$ ; if <i>itype</i> = 2, the problem type is $ABx = \lambda x$ ; if <i>itype</i> = 3, the problem type is $BAx = \lambda x$ .
<i>jobz</i>	CHARACTER*1. Must be ' <b>N</b> ' or ' <b>V</b> '. If <i>jobz</i> = ' <b>N</b> ', then compute eigenvalues only. If <i>jobz</i> = ' <b>V</b> ', then compute eigenvalues and eigenvectors.

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. If <i>uplo</i> = 'U', arrays <i>ap</i> and <i>bp</i> store the upper triangles of <i>A</i> and <i>B</i> ; If <i>uplo</i> = 'L', arrays <i>ap</i> and <i>bp</i> store the lower triangles of <i>A</i> and <i>B</i> .
<i>n</i>	INTEGER. The order of the matrices <i>A</i> and <i>B</i> ( <i>n</i> ≥ 0).
<i>ap</i> , <i>bp</i> , <i>work</i>	REAL for <i>sspgvd</i> DOUBLE PRECISION for <i>dspgvd</i> . Arrays: <i>ap</i> (*) contains the packed upper or lower triangle of the symmetric matrix <i>A</i> , as specified by <i>uplo</i> . The dimension of <i>ap</i> must be at least $\max(1, n*(n+1)/2)$ . <i>bp</i> (*) contains the packed upper or lower triangle of the symmetric matrix <i>B</i> , as specified by <i>uplo</i> . The dimension of <i>bp</i> must be at least $\max(1, n*(n+1)/2)$ . <i>work</i> ( <i>lwork</i> ) is a workspace array.
<i>ldz</i>	INTEGER. The leading dimension of the output array <i>z</i> ; <i>ldz</i> ≥ 1. If <i>jobz</i> = 'V', <i>ldz</i> ≥ $\max(1, n)$ .
<i>lwork</i>	INTEGER. The dimension of the array <i>work</i> . Constraints: If <i>n</i> ≤ 1, <i>lwork</i> ≥ 1; If <i>jobz</i> = 'N' and <i>n</i> > 1, <i>lwork</i> ≥ $2n$ ; If <i>jobz</i> = 'V' and <i>n</i> > 1, <i>lwork</i> ≥ $2n^2 + 6n + 1$ .
<i>iwork</i>	INTEGER. Workspace array, DIMENSION ( <i>liwork</i> ). .
<i>liwork</i>	INTEGER. The dimension of the array <i>iwork</i> . Constraints: If <i>n</i> ≤ 1, <i>liwork</i> ≥ 1; If <i>jobz</i> = 'N' and <i>n</i> > 1, <i>liwork</i> ≥ 1; If <i>jobz</i> = 'V' and <i>n</i> > 1, <i>liwork</i> ≥ $5n + 3$ .

## Output Parameters

<i>ap</i>	On exit, the contents of <i>ap</i> are overwritten.
-----------	---

<i>bp</i>	On exit, contains the triangular factor $U$ or $L$ from the Cholesky factorization $B = U^T U$ or $B = L L^T$ , in the same storage format as $B$ .
<i>w, z</i>	<p><b>REAL</b> for <b>sspgv</b>  <b>DOUBLE PRECISION</b> for <b>dspgv</b>.</p> <p>Arrays:</p> <p><i>w(*)</i>, <b>DIMENSION</b> at least <math>\max(1, n)</math>.  If <i>info</i> = 0, contains the eigenvalues in ascending order.</p> <p><i>z(ldz,*)</i>. The second dimension of <i>z</i> must be at least <math>\max(1, n)</math>.  If <i>jobz</i> = 'V', then if <i>info</i> = 0, <i>z</i> contains the matrix <math>Z</math> of eigenvectors. The eigenvectors are normalized as follows:  if <i>itype</i> = 1 or 2, <math>Z^T B Z = I</math>;  if <i>itype</i> = 3, <math>Z^T B^{-1} Z = I</math>;</p> <p>If <i>jobz</i> = 'N', then <i>z</i> is not referenced.</p>
<i>work(1)</i>	On exit, if <i>info</i> = 0, then <i>work(1)</i> returns the required minimal size of <i>lwork</i> .
<i>iwork(1)</i>	On exit, if <i>info</i> = 0, then <i>iwork(1)</i> returns the required minimal size of <i>liwork</i> .
<i>info</i>	<p><b>INTEGER</b>.</p> <p>If <i>info</i> = 0, the execution is successful.</p> <p>If <i>info</i> = <i>-i</i>, the <i>i</i>th argument had an illegal value.</p> <p>If <i>info</i> &gt; 0, <b>spptrf/dpptrf</b> and <b>sspevd/dspevd</b> returned an error code:</p> <ul style="list-style-type: none"> <li>If <i>info</i> = <i>i</i> ≤ <i>n</i>, <b>sspevd/dspevd</b> failed to converge, and <i>i</i> off-diagonal elements of an intermediate tridiagonal did not converge to zero;</li> <li>If <i>info</i> = <i>n + i</i>, for <math>1 \leq i \leq n</math>, then the leading minor of order <i>i</i> of <math>B</math> is not positive-definite. The factorization of <math>B</math> could not be completed and no eigenvalues or eigenvectors were computed.</li> </ul>

---

## ?hpgvd

*Computes all eigenvalues and, optionally, eigenvectors of a complex generalized Hermitian definite eigenproblem with matrices in packed storage. If eigenvectors are desired, it uses a divide and conquer method.*

---

```
call chpgvd ( itype, jobz, uplo, n, ap, bp, w, z, ldz, work, lwork,
              rwork, lrwork, iwork, liwork, info )
call zhpgvd ( itype, jobz, uplo, n, ap, bp, w, z, ldz, work, lwork,
              rwork, lrwork, iwork, liwork, info )
```

### Discussion

This routine computes all the eigenvalues, and optionally, the eigenvectors of a complex generalized Hermitian-definite eigenproblem, of the form

$$Ax = \lambda Bx, \quad ABx = \lambda x, \text{ or } BAx = \lambda x.$$

Here  $A$  and  $B$  are assumed to be Hermitian, stored in packed format, and  $B$  is also positive definite. If eigenvectors are desired, it uses a divide and conquer algorithm.

### Input Parameters

<i>itype</i>	<b>INTEGER.</b> Must be 1 or 2 or 3. Specifies the problem type to be solved: if <i>itype</i> = 1, the problem type is $Ax = \lambda Bx$ ; if <i>itype</i> = 2, the problem type is $ABx = \lambda x$ ; if <i>itype</i> = 3, the problem type is $BAx = \lambda x$ .
<i>jobz</i>	<b>CHARACTER*1.</b> Must be ' <b>N</b> ' or ' <b>V</b> '. If <i>jobz</i> = ' <b>N</b> ', then compute eigenvalues only. If <i>jobz</i> = ' <b>V</b> ', then compute eigenvalues and eigenvectors.

---

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. If <i>uplo</i> = 'U', arrays <i>ap</i> and <i>bp</i> store the upper triangles of <i>A</i> and <i>B</i> ; If <i>uplo</i> = 'L', arrays <i>ap</i> and <i>bp</i> store the lower triangles of <i>A</i> and <i>B</i> .
<i>n</i>	INTEGER. The order of the matrices <i>A</i> and <i>B</i> ( <i>n</i> $\geq$ 0).
<i>ap</i> , <i>bp</i> , <i>work</i>	COMPLEX for chpgvd DOUBLE COMPLEX for zhpgvd. Arrays: <i>ap</i> (*) contains the packed upper or lower triangle of the Hermitian matrix <i>A</i> , as specified by <i>uplo</i> . The dimension of <i>ap</i> must be at least $\max(1, n*(n+1)/2)$ . <i>bp</i> (*) contains the packed upper or lower triangle of the Hermitian matrix <i>B</i> , as specified by <i>uplo</i> . The dimension of <i>bp</i> must be at least $\max(1, n*(n+1)/2)$ . <i>work</i> ( <i>lwork</i> ) is a workspace array.
<i>ldz</i>	INTEGER. The leading dimension of the output array <i>z</i> ; <i>ldz</i> $\geq$ 1. If <i>jobz</i> = 'V', <i>ldz</i> $\geq \max(1, n)$ .
<i>lwork</i>	INTEGER. The dimension of the array <i>work</i> . Constraints: If <i>n</i> $\leq$ 1, <i>lwork</i> $\geq$ 1; If <i>jobz</i> = 'N' and <i>n</i> > 1, <i>lwork</i> $\geq n$ ; If <i>jobz</i> = 'V' and <i>n</i> > 1, <i>lwork</i> $\geq 2n$ .
<i>rwork</i>	REAL for chpgvd DOUBLE PRECISION for zhpgvd. Workspace array, DIMENSION ( <i>lrwork</i> ). Constraints: If <i>n</i> $\leq$ 1, <i>lrwork</i> $\geq$ 1; If <i>jobz</i> = 'N' and <i>n</i> > 1, <i>lrwork</i> $\geq n$ ; If <i>jobz</i> = 'V' and <i>n</i> > 1, <i>lrwork</i> $\geq 2n^2 + 5n + 1$ .
<i>iwork</i>	INTEGER. Workspace array, DIMENSION ( <i>liwork</i> ). .

---

*liwork*      **INTEGER.** The dimension of the array *iwork*.

Constraints:

If  $n \leq 1$ , *liwork*  $\geq 1$ ;

If *jobz* = 'N' and  $n > 1$ , *liwork*  $\geq 1$ ;

If *jobz* = 'V' and  $n > 1$ , *liwork*  $\geq 5n + 3$ .

## Output Parameters

*ap*      On exit, the contents of *ap* are overwritten.

*bp*      On exit, contains the triangular factor  $U$  or  $L$  from the Cholesky factorization  $B = U^H U$  or  $B = L L^H$ , in the same storage format as  $B$ .

*w*      **REAL** for *chpgvd*

**DOUBLE PRECISION** for *zhpgvd*.

Array, **DIMENSION** at least  $\max(1, n)$ .

If *info* = 0, contains the eigenvalues in ascending order.

*z*      **COMPLEX** for *chpgvd*

**DOUBLE COMPLEX** for *zhpgvd*.

Array  $z(ldz, *)$ . The second dimension of *z* must be at least  $\max(1, n)$ .

If *jobz* = 'V', then if *info* = 0, *z* contains the matrix  $Z$  of eigenvectors. The eigenvectors are normalized as follows:

if *itype* = 1 or 2,  $Z^H B Z = I$ ;

if *itype* = 3,  $Z^H B^{-1} Z = I$ ;

If *jobz* = 'N', then *z* is not referenced.

*work(1)*      On exit, if *info* = 0, then *work(1)* returns the required minimal size of *lwork*.

*rwork(1)*      On exit, if *info* = 0, then *rwork(1)* returns the required minimal size of *lrwork*.

*iwork(1)*      On exit, if *info* = 0, then *iwork(1)* returns the required minimal size of *liwork*.

*info*

INTEGER.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*th argument had an illegal value.

If *info* > 0, *cpptrf/zpptrf* and *chpevd/zhpevd* returned an error code:

If *info* = *i* ≤ *n*, *chpevd/zhpevd* failed to converge, and *i* off-diagonal elements of an intermediate tridiagonal did not converge to zero;  
If *info* = *n* + *i*, for  $1 \leq i \leq n$ , then the leading minor of order *i* of *B* is not positive-definite. The factorization of *B* could not be completed and no eigenvalues or eigenvectors were computed.

---

## ?spgvx

*Computes selected eigenvalues and,  
optionally, eigenvectors of a real generalized  
symmetric definite eigenproblem with  
matrices in packed storage.*

---

```
call sspgvx ( itype, jobz, range, uplo, n, ap, bp, vl, vu, il, iu,
abstol, m, w, z, ldz, work, iwork, ifail, info )
call dspgvx ( itype, jobz, range, uplo, n, ap, bp, vl, vu, il, iu,
abstol, m, w, z, ldz, work, iwork, ifail, info )
```

### Discussion

This routine computes selected eigenvalues, and optionally, the eigenvectors of a real generalized symmetric-definite eigenproblem, of the form

$$Ax = \lambda Bx, \quad ABx = \lambda x, \text{ or } BAx = \lambda x.$$

Here  $A$  and  $B$  are assumed to be symmetric, stored in packed format, and  $B$  is also positive definite.

Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

### Input Parameters

<i>itype</i>	<b>INTEGER.</b> Must be 1 or 2 or 3. Specifies the problem type to be solved: if <i>itype</i> = 1, the problem type is $Ax = \lambda Bx$ ; if <i>itype</i> = 2, the problem type is $ABx = \lambda x$ ; if <i>itype</i> = 3, the problem type is $BAx = \lambda x$ .
<i>jobz</i>	<b>CHARACTER*1.</b> Must be ' <b>N</b> ' or ' <b>V</b> '. If <i>jobz</i> = ' <b>N</b> ', then compute eigenvalues only. If <i>jobz</i> = ' <b>V</b> ', then compute eigenvalues and eigenvectors.
<i>range</i>	<b>CHARACTER*1.</b> Must be ' <b>A</b> ' or ' <b>V</b> ' or ' <b>I</b> '.

If `range = 'A'`, the routine computes all eigenvalues.  
 If `range = 'V'`, the routine computes eigenvalues  $\lambda_i$  in  
 the half-open interval:  $vl < \lambda_i \leq vu$ .  
 If `range = 'I'`, the routine computes eigenvalues with  
 indices `il` to `iu`.

`uplo`      CHARACTER\*1. Must be '`U`' or '`L`'.  
 If `uplo = 'U'`, arrays `ap` and `bp` store the upper  
 triangles of  $A$  and  $B$ ;  
 If `uplo = 'L'`, arrays `ap` and `bp` store the lower  
 triangles of  $A$  and  $B$ .

`n`            INTEGER. The order of the matrices  $A$  and  $B$  ( $n \geq 0$ ).

`ap, bp, work`    REAL for `sspgvx`  
                   DOUBLE PRECISION for `dspgvx`.

Arrays:

`ap(*)` contains the packed upper or lower triangle of  
 the symmetric matrix  $A$ , as specified by `uplo`. The  
 dimension of `ap` must be at least  $\max(1, n*(n+1)/2)$ .

`bp(*)` contains the packed upper or lower triangle of  
 the symmetric matrix  $B$ , as specified by `uplo`. The  
 dimension of `bp` must be at least  $\max(1, n*(n+1)/2)$ .

`work(*)` is a workspace array, DIMENSION at least  
 $\max(1, 8n)$ .

`vl, vu`        REAL for `sspgvx`  
                   DOUBLE PRECISION for `dspgvx`.  
 If `range = 'V'`, the lower and upper bounds of the  
 interval to be searched for eigenvalues.  
 Constraint:  $vl < vu$ .

If `range = 'A'` or '`I`', `vl` and `vu` are not referenced.

`il, iu`        INTEGER.  
 If `range = 'I'`, the indices in ascending order of the  
 smallest and largest eigenvalues to be returned.  
 Constraint:  $1 \leq il \leq iu \leq n$ , if  $n > 0$ ;  $il=1$  and  $iu=0$   
 if  $n = 0$ .  
 If `range = 'A'` or '`V`', `il` and `iu` are not referenced.

<i>abstol</i>	<small>REAL for <b>sspgvx</b> DOUBLE PRECISION for <b>dspgvx</b>.</small>
	The absolute error tolerance for the eigenvalues. See <i>Application Notes</i> for more information.
<i>ldz</i>	<small>INTEGER.</small> The leading dimension of the output array <i>z</i> . Constraints: $ldz \geq 1$ ; if <i>jobz</i> = 'V', $ldz \geq \max(1, n)$ .
<i>iwork</i>	<small>INTEGER.</small> Workspace array, <small>DIMENSION</small> at least $\max(1, 5n)$ .

## Output Parameters

<i>ap</i>	On exit, the contents of <i>ap</i> are overwritten.
<i>bp</i>	On exit, contains the triangular factor <i>U</i> or <i>L</i> from the Cholesky factorization $B = U^T U$ or $B = L L^T$ , in the same storage format as <i>B</i> .
<i>m</i>	<small>INTEGER.</small> The total number of eigenvalues found, $0 \leq m \leq n$ . If <i>range</i> = 'A', <i>m</i> = <i>n</i> , and if <i>range</i> = 'I', <i>m</i> = <i>iu</i> - <i>il</i> + 1.
<i>w, z</i>	<small>REAL for <b>sspgvx</b> DOUBLE PRECISION for <b>dspgvx</b>.</small> Arrays: <i>w</i> ( * ), <small>DIMENSION</small> at least $\max(1, n)$ . If <i>info</i> = 0, contains the eigenvalues in ascending order. <i>z</i> ( <i>ldz</i> , * ). The second dimension of <i>z</i> must be at least $\max(1, n)$ . If <i>jobz</i> = 'V', then if <i>info</i> = 0, the first <i>m</i> columns of <i>z</i> contain the orthonormal eigenvectors of the matrix <i>A</i> corresponding to the selected eigenvalues, with the <i>i</i> -th column of <i>z</i> holding the eigenvector associated with <i>w</i> ( <i>i</i> ). The eigenvectors are normalized as follows: if <i>itype</i> = 1 or 2, $Z^T B Z = I$ ; if <i>itype</i> = 3, $Z^T B^{-1} Z = I$ ; If <i>jobz</i> = 'N', then <i>z</i> is not referenced. If an eigenvector fails to converge, then that column of <i>z</i> contains the latest approximation to the eigenvector, and

the index of the eigenvector is returned in *ifail*.

Note: you must ensure that at least  $\max(1, m)$  columns are supplied in the array *z*; if *range* = 'V', the exact value of *m* is not known in advance and an upper bound must be used.

*ifail*

INTEGER.

Array, DIMENSION at least  $\max(1, n)$ .

If *jobz* = 'V', then if *info* = 0, the first *m* elements of *ifail* are zero; if *info* > 0, the *ifail* contains the indices of the eigenvectors that failed to converge.

If *jobz* = 'N', then *ifail* is not referenced.

*info*

INTEGER.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*th argument had an illegal value.

If *info* > 0, *spptrf/dpptrf* and *sspevx/dspevx* returned an error code:

If *info* = *i* ≤ *n*, *sspevx/dspevx* failed to converge, and *i* eigenvectors failed to converge. Their indices are stored in the array *ifail*;

If *info* = *n* + *i*, for  $1 \leq i \leq n$ , then the leading minor of order *i* of *B* is not positive-definite. The factorization of *B* could not be completed and no eigenvalues or eigenvectors were computed.

## Application Notes

An approximate eigenvalue is accepted as converged when it is determined to lie in an interval [a,b] of width less than or equal to

*abstol* +  $\epsilon * \max(|a|, |b|)$ , where  $\epsilon$  is the machine precision. If *abstol* is less than or equal to zero, then  $\epsilon * \|T\|_1$  will be used in its place, where *T* is the tridiagonal matrix obtained by reducing *A* to tridiagonal form.

Eigenvalues will be computed most accurately when *abstol* is set to twice the underflow threshold  $2 * ?lamch('S')$ , not zero. If this routine returns with *info* > 0, indicating that some eigenvectors did not converge, try setting *abstol* to  $2 * ?lamch('S')$ .

---

## ?hpgvx

*Computes selected eigenvalues and, optionally, eigenvectors of a generalized Hermitian definite eigenproblem with matrices in packed storage.*

---

```
call chpgvx ( itype, jobz, range, uplo, n, ap, bp, vl, vu, il, iu,
              abstol, m, w, z, ldz, work, rwork, iwork, ifail, info )
call zhpgvx ( itype, jobz, range, uplo, n, ap, bp, vl, vu, il, iu,
              abstol, m, w, z, ldz, work, rwork, iwork, ifail, info )
```

### Discussion

This routine computes selected eigenvalues, and optionally, the eigenvectors of a complex generalized Hermitian-definite eigenproblem, of the form

$$Ax = \lambda Bx, \quad ABx = \lambda x, \quad \text{or} \quad BAx = \lambda x.$$

Here  $A$  and  $B$  are assumed to be Hermitian, stored in packed format, and  $B$  is also positive definite.

Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

### Input Parameters

<i>itype</i>	<small>INTEGER.</small> Must be 1 or 2 or 3. Specifies the problem type to be solved: if <i>itype</i> = 1, the problem type is $Ax = \lambda Bx$ ; if <i>itype</i> = 2, the problem type is $ABx = \lambda x$ ; if <i>itype</i> = 3, the problem type is $BAx = \lambda x$ .
<i>jobz</i>	<small>CHARACTER*1.</small> Must be ' <b>N</b> ' or ' <b>V</b> '. If <i>jobz</i> = ' <b>N</b> ', then compute eigenvalues only. If <i>jobz</i> = ' <b>V</b> ', then compute eigenvalues and eigenvectors.
<i>range</i>	<small>CHARACTER*1.</small> Must be ' <b>A</b> ' or ' <b>V</b> ' or ' <b>I</b> '. If <i>range</i> = ' <b>A</b> ', the routine computes all eigenvalues. If <i>range</i> = ' <b>V</b> ', the routine computes eigenvalues $\lambda_i$ in the half-open interval: $vl < \lambda_i \leq vu$ .

If `range = 'I'`, the routine computes eigenvalues with indices `i1` to `iu`.

`uplo` CHARACTER\*1. Must be '`U`' or '`L`'.  
 If `uplo = 'U'`, arrays `ap` and `bp` store the upper triangles of  $A$  and  $B$ ;  
 If `uplo = 'L'`, arrays `ap` and `bp` store the lower triangles of  $A$  and  $B$ .

`n` INTEGER. The order of the matrices  $A$  and  $B$  ( $n \geq 0$ ).

`ap, bp, work` COMPLEX for `chpgvx`  
 DOUBLE COMPLEX for `zhpgvx`.

Arrays:

`ap(*)` contains the packed upper or lower triangle of the Hermitian matrix  $A$ , as specified by `uplo`. The dimension of `ap` must be at least  $\max(1, n*(n+1)/2)$ .

`bp(*)` contains the packed upper or lower triangle of the Hermitian matrix  $B$ , as specified by `uplo`. The dimension of `bp` must be at least  $\max(1, n*(n+1)/2)$ .

`work(*)` is a workspace array, DIMENSION at least  $\max(1, 2n)$ .

`vl, vu` REAL for `chpgvx`  
 DOUBLE PRECISION for `zhpgvx`.

If `range = 'V'`, the lower and upper bounds of the interval to be searched for eigenvalues.

Constraint:  $vl < vu$ .

If `range = 'A'` or '`I`', `vl` and `vu` are not referenced.

`i1, iu` INTEGER.  
 If `range = 'I'`, the indices in ascending order of the smallest and largest eigenvalues to be returned.  
 Constraint:  $1 \leq i1 \leq iu \leq n$ , if  $n > 0$ ;  $i1=1$  and  $iu=0$  if  $n = 0$ .

If `range = 'A'` or '`V`', `i1` and `iu` are not referenced.

`abstol` REAL for `chpgvx`  
 DOUBLE PRECISION for `zhpgvx`.  
 The absolute error tolerance for the eigenvalues.

See *Application Notes* for more information.

<i>ldz</i>	<b>INTEGER.</b> The leading dimension of the output array <i>z</i> ; $ldz \geq 1$ . If <i>jobz</i> = 'V', $ldz \geq \max(1, n)$ .
<i>rwork</i>	<b>REAL</b> for <i>chpgvx</i> <b>DOUBLE PRECISION</b> for <i>zhpgvx</i> . Workspace array, <b>DIMENSION</b> at least $\max(1, 7n)$ .
<i>iwork</i>	<b>INTEGER</b> . Workspace array, <b>DIMENSION</b> at least $\max(1, 5n)$ .

## Output Parameters

<i>ap</i>	On exit, the contents of <i>ap</i> are overwritten.
<i>bp</i>	On exit, contains the triangular factor <i>U</i> or <i>L</i> from the Cholesky factorization $B = U^H U$ or $B = L L^H$ , in the same storage format as <i>B</i> .
<i>m</i>	<b>INTEGER.</b> The total number of eigenvalues found, $0 \leq m \leq n$ . If <i>range</i> = 'A', $m = n$ , and if <i>range</i> = 'I', $m = iu - il + 1$ .
<i>w</i>	<b>REAL</b> for <i>chpgvx</i> <b>DOUBLE PRECISION</b> for <i>zhpgvx</i> . Array, <b>DIMENSION</b> at least $\max(1, n)$ . If <i>info</i> = 0, contains the eigenvalues in ascending order.
<i>z</i>	<b>COMPLEX</b> for <i>chpgvx</i> <b>DOUBLE COMPLEX</b> for <i>zhpgvx</i> . Array <i>z</i> ( <i>ldz</i> , *). The second dimension of <i>z</i> must be at least $\max(1, n)$ . If <i>jobz</i> = 'V', then if <i>info</i> = 0, the first <i>m</i> columns of <i>z</i> contain the orthonormal eigenvectors of the matrix <i>A</i> corresponding to the selected eigenvalues, with the <i>i</i> -th column of <i>z</i> holding the eigenvector associated with <i>w(i)</i> . The eigenvectors are normalized as follows: if <i>itype</i> = 1 or 2, $Z^H B Z = I$ ; if <i>itype</i> = 3, $Z^H B^{-1} Z = I$ ; If <i>jobz</i> = 'N', then <i>z</i> is not referenced.

If an eigenvector fails to converge, then that column of  $\mathbf{z}$  contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in  $\text{ifail}$ .

Note: you must ensure that at least  $\max(1, m)$  columns are supplied in the array  $\mathbf{z}$ ; if  $\text{range} = \text{'V'}$ , the exact value of  $m$  is not known in advance and an upper bound must be used.

$\text{ifail}$

**INTEGER.**

Array, **DIMENSION** at least  $\max(1, n)$ .

If  $\text{jobz} = \text{'V'}$ , then if  $\text{info} = 0$ , the first  $m$  elements of  $\text{ifail}$  are zero; if  $\text{info} > 0$ , the  $\text{ifail}$  contains the indices of the eigenvectors that failed to converge.

If  $\text{jobz} = \text{'N'}$ , then  $\text{ifail}$  is not referenced.

$\text{info}$

**INTEGER.**

If  $\text{info} = 0$ , the execution is successful.

If  $\text{info} = -i$ , the  $i$ th argument had an illegal value.

If  $\text{info} > 0$ , **cpptrf/zpptrf** and **chpevx/zhpevx** returned an error code:

If  $\text{info} = i \leq n$ , **chpevx/zhpevx** failed to converge, and  $i$  eigenvectors failed to converge. Their indices are stored in the array  $\text{ifail}$ ;

If  $\text{info} = n + i$ , for  $1 \leq i \leq n$ , then the leading minor of order  $i$  of  $B$  is not positive-definite. The factorization of  $B$  could not be completed and no eigenvalues or eigenvectors were computed.

## Application Notes

An approximate eigenvalue is accepted as converged when it is determined to lie in an interval  $[a,b]$  of width less than or equal to

$\text{abstol} + \epsilon * \max(|a|, |b|)$ , where  $\epsilon$  is the machine precision. If  $\text{abstol}$  is less than or equal to zero, then  $\epsilon * \|T\|_1$  will be used in its place, where  $T$  is the tridiagonal matrix obtained by reducing  $A$  to tridiagonal form.

Eigenvalues will be computed most accurately when  $\text{abstol}$  is set to twice the underflow threshold  $2 * \text{?lamch('S')}$ , not zero. If this routine returns with  $\text{info} > 0$ , indicating that some eigenvectors did not converge, try setting  $\text{abstol}$  to  $2 * \text{?lamch('S')}$ .

---

## ?sbgv

*Computes all eigenvalues and, optionally, eigenvectors of a real generalized symmetric definite eigenproblem with banded matrices.*

---

```
call ssbgv ( jobz, uplo, n, ka, kb, ab, ldab, bb, ldbb, w, z, ldz,
             work, info )
call dsbgv ( jobz, uplo, n, ka, kb, ab, ldab, bb, ldbb, w, z, ldz,
             work, info )
```

### Discussion

This routine computes all the eigenvalues, and optionally, the eigenvectors of a real generalized symmetric-definite banded eigenproblem, of the form  $Ax = \lambda Bx$ . Here  $A$  and  $B$  are assumed to be symmetric and banded, and  $B$  is also positive definite.

### Input Parameters

<i>jobz</i>	CHARACTER*1. Must be ' <b>N</b> ' or ' <b>V</b> '. If <i>jobz</i> = ' <b>N</b> ', then compute eigenvalues only. If <i>jobz</i> = ' <b>V</b> ', then compute eigenvalues and eigenvectors.
<i>uplo</i>	CHARACTER*1. Must be ' <b>U</b> ' or ' <b>L</b> '. If <i>uplo</i> = ' <b>U</b> ', arrays <i>ab</i> and <i>bb</i> store the upper triangles of $A$ and $B$ ; If <i>uplo</i> = ' <b>L</b> ', arrays <i>ab</i> and <i>bb</i> store the lower triangles of $A$ and $B$ .
<i>n</i>	INTEGER. The order of the matrices $A$ and $B$ ( <i>n</i> $\geq$ 0).
<i>ka</i>	INTEGER. The number of super- or sub-diagonals in $A$ ( <i>ka</i> $\geq$ 0).
<i>kb</i>	INTEGER. The number of super- or sub-diagonals in $B$ ( <i>kb</i> $\geq$ 0).

<i>ab, bb, work</i>	<small>REAL for ssbgv DOUBLE PRECISION for dsbgv</small>
Arrays:	
<i>ab</i> ( <i>ldab</i> , *)	is an array containing either upper or lower triangular part of the symmetric matrix $A$ (as specified by <i>uplo</i> ) in band storage format.
The second dimension of the array <i>ab</i>	must be at least $\max(1, n)$ .
<i>bb</i> ( <i>ldbb</i> , *)	is an array containing either upper or lower triangular part of the symmetric matrix $B$ (as specified by <i>uplo</i> ) in band storage format.
The second dimension of the array <i>bb</i>	must be at least $\max(1, n)$ .
<i>work</i> (*)	is a workspace array, <small>DIMENSION</small> at least $\max(1, 3n)$
<i>ldab</i>	<small>INTEGER.</small> The first dimension of the array <i>ab</i> ; must be at least <i>ka</i> +1.
<i>ldbb</i>	<small>INTEGER.</small> The first dimension of the array <i>bb</i> ; must be at least <i>kb</i> +1.
<i>ldz</i>	<small>INTEGER.</small> The leading dimension of the output array <i>z</i> ; <i>ldz</i> $\geq 1$ . If <i>jobz</i> = 'V', <i>ldz</i> $\geq \max(1, n)$ .

## Output Parameters

<i>ab</i>	On exit, the contents of <i>ab</i> are overwritten.
<i>bb</i>	On exit, contains the factor $S$ from the split Cholesky factorization $B = S^T S$ , as returned by <i>spbstf/dpbstf</i> .
<i>w, z</i>	<small>REAL for ssbgv DOUBLE PRECISION for dsbgv</small>
Arrays:	
<i>w</i> (*)	<small>DIMENSION</small> at least $\max(1, n)$ .
If <i>info</i> = 0,	contains the eigenvalues in ascending order.
<i>z</i> ( <i>ldz</i> , *)	The second dimension of <i>z</i> must be at least $\max(1, n)$ .
If <i>jobz</i> = 'V', then if <i>info</i> = 0,	<i>z</i> contains the matrix $Z$ of eigenvectors, with the <i>i</i> -th column of <i>z</i> holding the

eigenvector associated with  $w(i)$ . The eigenvectors are normalized so that  $Z^T B Z = I$ .

If  $jobz = 'N'$ , then  $z$  is not referenced.

*info*

INTEGER.

If  $info = 0$ , the execution is successful.

If  $info = -i$ , the  $i$ th argument had an illegal value.

If  $info > 0$ , and

if  $i \leq n$ , the algorithm failed to converge, and  $i$  off-diagonal elements of an intermediate tridiagonal did not converge to zero;

if  $info = n + i$ , for  $1 \leq i \leq n$ , then **spbstf/dpbstf** returned  $info = i$  and  $B$  is not positive-definite. The factorization of  $B$  could not be completed and no eigenvalues or eigenvectors were computed.

## ?hbgv

*Computes all eigenvalues and, optionally, eigenvectors of a complex generalized Hermitian definite eigenproblem with banded matrices.*

```
call chbgv ( jobz, uplo, n, ka, kb, ab, ldab, bb, ldbb, w, z, ldz,
             work, rwork, info )
call zhbgv ( jobz, uplo, n, ka, kb, ab, ldab, bb, ldbb, w, z, ldz,
             work, rwork, info )
```

### Discussion

This routine computes all the eigenvalues, and optionally, the eigenvectors of a complex generalized Hermitian-definite banded eigenproblem, of the form  $Ax = \lambda Bx$ . Here  $A$  and  $B$  are assumed to be Hermitian and banded, and  $B$  is also positive definite.

### Input Parameters

<i>jobz</i>	CHARACTER*1. Must be ' <b>N</b> ' or ' <b>V</b> '. If <i>jobz</i> = ' <b>N</b> ', then compute eigenvalues only. If <i>jobz</i> = ' <b>V</b> ', then compute eigenvalues and eigenvectors.
<i>uplo</i>	CHARACTER*1. Must be ' <b>U</b> ' or ' <b>L</b> '. If <i>uplo</i> = ' <b>U</b> ', arrays <i>ab</i> and <i>bb</i> store the upper triangles of $A$ and $B$ ; If <i>uplo</i> = ' <b>L</b> ', arrays <i>ab</i> and <i>bb</i> store the lower triangles of $A$ and $B$ .
<i>n</i>	INTEGER. The order of the matrices $A$ and $B$ ( <i>n</i> $\geq 0$ ).
<i>ka</i>	INTEGER. The number of super- or sub-diagonals in $A$ ( <i>ka</i> $\geq 0$ ).
<i>kb</i>	INTEGER. The number of super- or sub-diagonals in $B$ ( <i>kb</i> $\geq 0$ ).

<i>ab, bb, work</i>	<small>COMPLEX for chbgv DOUBLE COMPLEX for zhbgv</small>
Arrays:	
<i>ab (ldab, *)</i>	is an array containing either upper or lower triangular part of the Hermitian matrix $A$ (as specified by <i>uplo</i> ) in band storage format.
The second dimension of the array <i>ab</i> must be at least $\max(1, n)$ .	
<i>bb (ldbb, *)</i>	is an array containing either upper or lower triangular part of the Hermitian matrix $B$ (as specified by <i>uplo</i> ) in band storage format.
The second dimension of the array <i>bb</i> must be at least $\max(1, n)$ .	
<i>work (*)</i>	is a workspace array, <small>DIMENSION</small> at least $\max(1, n)$ .
<i>ldab</i>	<small>INTEGER.</small> The first dimension of the array <i>ab</i> ; must be at least <i>ka</i> +1.
<i>ldbb</i>	<small>INTEGER.</small> The first dimension of the array <i>bb</i> ; must be at least <i>kb</i> +1.
<i>ldz</i>	<small>INTEGER.</small> The leading dimension of the output array <i>z</i> ; <i>ldz</i> $\geq 1$ . If <i>jobz</i> = 'V', <i>ldz</i> $\geq \max(1, n)$ .
<i>rwork</i>	<small>REAL for chbgv DOUBLE PRECISION for zhbgv.</small> Workspace array, <small>DIMENSION</small> at least $\max(1, 3n)$ .

## Output Parameters

<i>ab</i>	On exit, the contents of <i>ab</i> are overwritten.
<i>bb</i>	On exit, contains the factor $S$ from the split Cholesky factorization $B = S^H S$ , as returned by <i>cpbstf/zpbstf</i> .
<i>w</i>	<small>REAL for chbgv DOUBLE PRECISION for zhbgv.</small> Array, <small>DIMENSION</small> at least $\max(1, n)$ . If <i>info</i> = 0, contains the eigenvalues in ascending order.

*z*

COMPLEX for `chbgv`

DOUBLE COMPLEX for `zhbgv`

Array `z(1:dz, *)`. The second dimension of `z` must be at least `max(1, n)`.

If `jobz = 'V'`, then if `info = 0`, `z` contains the matrix  $Z$  of eigenvectors, with the  $i$ -th column of `z` holding the eigenvector associated with  $w(i)$ . The eigenvectors are normalized so that  $Z^H B Z = I$ .

If `jobz = 'N'`, then `z` is not referenced.

*info*

INTEGER.

If `info = 0`, the execution is successful.

If `info = -i`, the  $i$ th argument had an illegal value.

If `info > 0`, and

if  $i \leq n$ , the algorithm failed to converge, and  $i$  off-diagonal elements of an intermediate tridiagonal did not converge to zero;

if `info = n + i`, for  $1 \leq i \leq n$ , then `cpbstf/zpbstf` returned `info = i` and  $B$  is not positive-definite. The factorization of  $B$  could not be completed and no eigenvalues or eigenvectors were computed.

## ?sbgvd

*Computes all eigenvalues and, optionally, eigenvectors of a real generalized symmetric definite eigenproblem with banded matrices. If eigenvectors are desired, it uses a divide and conquer method.*

---

```
call ssbgvd ( jobz, uplo, n, ka, kb, ab, ldab, bb, ldbb, w, z, ldz,
               work, lwork, iwork, liwork, info )
call dsbgvd ( jobz, uplo, n, ka, kb, ab, ldab, bb, ldbb, w, z, ldz,
               work, lwork, iwork, liwork, info )
```

### Discussion

This routine computes all the eigenvalues, and optionally, the eigenvectors of a real generalized symmetric-definite banded eigenproblem, of the form  $Ax = \lambda Bx$ . Here  $A$  and  $B$  are assumed to be symmetric and banded, and  $B$  is also positive definite. If eigenvectors are desired, it uses a divide and conquer algorithm.

### Input Parameters

<i>jobz</i>	CHARACTER*1. Must be ' <b>N</b> ' or ' <b>V</b> '. If <i>jobz</i> = ' <b>N</b> ', then compute eigenvalues only. If <i>jobz</i> = ' <b>V</b> ', then compute eigenvalues and eigenvectors.
<i>uplo</i>	CHARACTER*1. Must be ' <b>U</b> ' or ' <b>L</b> '. If <i>uplo</i> = ' <b>U</b> ', arrays <i>ab</i> and <i>bb</i> store the upper triangles of $A$ and $B$ ; If <i>uplo</i> = ' <b>L</b> ', arrays <i>ab</i> and <i>bb</i> store the lower triangles of $A$ and $B$ .
<i>n</i>	INTEGER. The order of the matrices $A$ and $B$ ( <i>n</i> $\geq$ 0).
<i>ka</i>	INTEGER. The number of super- or sub-diagonals in $A$ ( <i>ka</i> $\geq$ 0).

<i>kb</i>	<code>INTEGER</code> . The number of super- or sub-diagonals in <i>B</i> ( <i>kb</i> ≥ 0).
<i>ab, bb, work</i>	<code>REAL</code> for <code>ssbgvd</code> <code>DOUBLE PRECISION</code> for <code>dsbgvd</code>
	Arrays: <i>ab</i> ( <i>ldab</i> , *) is an array containing either upper or lower triangular part of the symmetric matrix <i>A</i> (as specified by <i>uplo</i> ) in band storage format. The second dimension of the array <i>ab</i> must be at least $\max(1, n)$ . <i>bb</i> ( <i>ldbb</i> , *) is an array containing either upper or lower triangular part of the symmetric matrix <i>B</i> (as specified by <i>uplo</i> ) in band storage format. The second dimension of the array <i>bb</i> must be at least $\max(1, n)$ . <i>work</i> ( <i>lwork</i> ) is a workspace array.
<i>ldab</i>	<code>INTEGER</code> . The first dimension of the array <i>ab</i> ; must be at least <i>ka</i> +1.
<i>ldbb</i>	<code>INTEGER</code> . The first dimension of the array <i>bb</i> ; must be at least <i>kb</i> +1.
<i>ldz</i>	<code>INTEGER</code> . The leading dimension of the output array <i>z</i> ; <i>ldz</i> ≥ 1. If <i>jobz</i> = 'V', <i>ldz</i> ≥ $\max(1, n)$ .
<i>lwork</i>	<code>INTEGER</code> . The dimension of the array <i>work</i> . Constraints: If <i>n</i> ≤ 1, <i>lwork</i> ≥ 1; If <i>jobz</i> = 'N' and <i>n</i> >1, <i>lwork</i> ≥ $3n$ ; If <i>jobz</i> = 'V' and <i>n</i> >1, <i>lwork</i> ≥ $2n^2+5n+1$ .
<i>iwork</i>	<code>INTEGER</code> . Workspace array, <code>DIMENSION</code> ( <i>liwork</i> ). .
<i>liwork</i>	<code>INTEGER</code> . The dimension of the array <i>iwork</i> . Constraints: If <i>n</i> ≤ 1, <i>liwork</i> ≥ 1; If <i>jobz</i> = 'N' and <i>n</i> >1, <i>liwork</i> ≥ 1; If <i>jobz</i> = 'V' and <i>n</i> >1, <i>liwork</i> ≥ $5n+3$ .

## Output Parameters

<i>ab</i>	On exit, the contents of <i>ab</i> are overwritten.
<i>bb</i>	On exit, contains the factor <i>S</i> from the split Cholesky factorization $B = S^T S$ , as returned by <b>spbstf/dpbstf</b> .
<i>w, z</i>	<p><b>REAL</b> for <b>ssbgvd</b>  <b>DOUBLE PRECISION</b> for <b>dsbgvd</b></p> <p>Arrays:</p> <p><i>w(*)</i>, <b>DIMENSION</b> at least <math>\max(1, n)</math>.  If <i>info</i> = 0, contains the eigenvalues in ascending order.</p> <p><i>z(ldz, *)</i>. The second dimension of <i>z</i> must be at least <math>\max(1, n)</math>.  If <i>jobz</i> = 'V', then if <i>info</i> = 0, <i>z</i> contains the matrix <i>Z</i> of eigenvectors, with the <i>i</i>-th column of <i>z</i> holding the eigenvector associated with <i>w(i)</i>. The eigenvectors are normalized so that <math>Z^T B Z = I</math>.  If <i>jobz</i> = 'N', then <i>z</i> is not referenced.</p>
<i>work(1)</i>	On exit, if <i>info</i> = 0, then <i>work(1)</i> returns the required minimal size of <i>lwork</i> .
<i>iwork(1)</i>	On exit, if <i>info</i> = 0, then <i>iwork(1)</i> returns the required minimal size of <i>liwork</i> .
<i>info</i>	<p><b>INTEGER</b>.  If <i>info</i> = 0, the execution is successful.  If <i>info</i> = -<i>i</i>, the <i>i</i>th argument had an illegal value.  If <i>info</i> &gt; 0, and</p> <ul style="list-style-type: none"> <li>if <i>i</i> ≤ <i>n</i>, the algorithm failed to converge, and <i>i</i> off-diagonal elements of an intermediate tridiagonal did not converge to zero;</li> <li>if <i>info</i> = <i>n+i</i>, for <math>1 \leq i \leq n</math>, then <b>spbstf/dpbstf</b> returned <i>info</i> = <i>i</i> and <i>B</i> is not positive-definite. The factorization of <i>B</i> could not be completed and no eigenvalues or eigenvectors were computed.</li> </ul>

## ?hbvd

*Computes all eigenvalues and, optionally, eigenvectors of a complex generalized Hermitian definite eigenproblem with banded matrices. If eigenvectors are desired, it uses a divide and conquer method.*

```
call chbgvd ( jobz, uplo, n, ka, kb, ab, ldab, bb, ldbb, w, z, ldz,
               work, lwork, rwork, lrwork, iwork, liwork, info )
call zhbgvd ( jobz, uplo, n, ka, kb, ab, ldab, bb, ldbb, w, z, ldz,
               work, lwork, rwork, lrwork, iwork, liwork, info )
```

### Discussion

This routine computes all the eigenvalues, and optionally, the eigenvectors of a complex generalized Hermitian-definite banded eigenproblem, of the form  $Ax = \lambda Bx$ . Here  $A$  and  $B$  are assumed to be Hermitian and banded, and  $B$  is also positive definite. If eigenvectors are desired, it uses a divide and conquer algorithm.

### Input Parameters

<i>jobz</i>	CHARACTER*1. Must be ' <b>N</b> ' or ' <b>V</b> '. If <i>jobz</i> = ' <b>N</b> ', then compute eigenvalues only. If <i>jobz</i> = ' <b>V</b> ', then compute eigenvalues and eigenvectors.
<i>uplo</i>	CHARACTER*1. Must be ' <b>U</b> ' or ' <b>L</b> '. If <i>uplo</i> = ' <b>U</b> ', arrays <i>ab</i> and <i>bb</i> store the upper triangles of $A$ and $B$ ; If <i>uplo</i> = ' <b>L</b> ', arrays <i>ab</i> and <i>bb</i> store the lower triangles of $A$ and $B$ .
<i>n</i>	INTEGER. The order of the matrices $A$ and $B$ ( <i>n</i> $\geq 0$ ).
<i>ka</i>	INTEGER. The number of super- or sub-diagonals in $A$ ( <i>ka</i> $\geq 0$ ).

<i>kb</i>	<b>INTEGER.</b> The number of super- or sub-diagonals in <i>B</i> ( <i>kb</i> ≥ 0).
<i>ab, bb, work</i>	<b>COMPLEX</b> for <b>chbgvd</b> <b>DOUBLE COMPLEX</b> for <b>zhbgvd</b> Arrays: <i>ab</i> ( <i>ldab</i> , *) is an array containing either upper or lower triangular part of the Hermitian matrix <i>A</i> (as specified by <i>uplo</i> ) in band storage format. The second dimension of the array <i>ab</i> must be at least $\max(1, n)$ . <i>bb</i> ( <i>ldbb</i> , *) is an array containing either upper or lower triangular part of the Hermitian matrix <i>B</i> (as specified by <i>uplo</i> ) in band storage format. The second dimension of the array <i>bb</i> must be at least $\max(1, n)$ . <i>work</i> ( <i>lwork</i> ) is a workspace array.
<i>ldab</i>	<b>INTEGER.</b> The first dimension of the array <i>ab</i> ; must be at least <i>ka</i> +1.
<i>ldbb</i>	<b>INTEGER.</b> The first dimension of the array <i>bb</i> ; must be at least <i>kb</i> +1.
<i>ldz</i>	<b>INTEGER.</b> The leading dimension of the output array <i>z</i> ; <i>ldz</i> ≥ 1. If <i>jobz</i> = 'V', <i>ldz</i> ≥ $\max(1, n)$ .
<i>lwork</i>	<b>INTEGER.</b> The dimension of the array <i>work</i> . Constraints: If <i>n</i> ≤ 1, <i>lwork</i> ≥ 1; If <i>jobz</i> = 'N' and <i>n</i> >1, <i>lwork</i> ≥ <i>n</i> ; If <i>jobz</i> = 'V' and <i>n</i> >1, <i>lwork</i> ≥ $2n^2$ .
<i>rwork</i>	<b>REAL</b> for <b>chbgvd</b> <b>DOUBLE PRECISION</b> for <b>zhbgvd</b> . Workspace array, <b>DIMENSION</b> ( <i>lrwork</i> ).
<i>lrwork</i>	<b>INTEGER.</b> The dimension of the array <i>rwork</i> .

Constraints:

If  $n \leq 1$ ,  $lrwork \geq 1$ ;  
 If  $jobz = 'N'$  and  $n > 1$ ,  $lrwork \geq n$ ;  
 If  $jobz = 'V'$  and  $n > 1$ ,  $lrwork \geq 2n^2 + 5n + 1$ .

*iwork*

INTEGER.

Workspace array, DIMENSION (*liwork*).

*liwork*

INTEGER. The dimension of the array *iwork*.

Constraints:

If  $n \leq 1$ ,  $liwork \geq 1$ ;  
 If  $jobz = 'N'$  and  $n > 1$ ,  $liwork \geq 1$ ;  
 If  $jobz = 'V'$  and  $n > 1$ ,  $liwork \geq 5n + 3$ .

## Output Parameters

*ab*

On exit, the contents of *ab* are overwritten.

*bb*

On exit, contains the factor  $S$  from the split Cholesky factorization  $B = S^H S$ , as returned by *cpbstf/zpbstf*.

*w*

REAL for *chbgvd*

DOUBLE PRECISION for *zhbgvd*.

Array, DIMENSION at least  $\max(1, n)$ .

If *info* = 0, contains the eigenvalues in ascending order.

*z*

COMPLEX for *chbgvd*

DOUBLE COMPLEX for *zhbgvd*

Array *z*(*ldz*, \*). The second dimension of *z* must be at least  $\max(1, n)$ .

If  $jobz = 'V'$ , then if *info* = 0, *z* contains the matrix  $Z$  of eigenvectors, with the *i*-th column of *z* holding the eigenvector associated with *w(i)*. The eigenvectors are normalized so that  $Z^H B Z = I$ .

If  $jobz = 'N'$ , then *z* is not referenced.

*work(1)*

On exit, if *info* = 0, then *work(1)* returns the required minimal size of *lwork*.

*rwork(1)*

On exit, if *info* = 0, then *rwork(1)* returns the required minimal size of *lrwork*.

*iwork(1)*

On exit, if *info* = 0, then *iwork(1)* returns the required minimal size of *lwork*.

*info*

INTEGER.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*th argument had an illegal value.

If *info* > 0, and

if *i* ≤ *n*, the algorithm failed to converge, and *i* off-diagonal elements of an intermediate tridiagonal did not converge to zero;

if *info* = *n* + *i*, for  $1 \leq i \leq n$ , then *cpbstf/zpbstf* returned *info* = *i* and *B* is not positive-definite. The factorization of *B* could not be completed and no eigenvalues or eigenvectors were computed.

## ?sbgvx

*Computes selected eigenvalues and, optionally, eigenvectors of a real generalized symmetric definite eigenproblem with banded matrices.*

```
call ssbgvx ( jobz, range, uplo, n, ka, kb, ab, ldab, bb, ldbb, q,
               ldq, vl, vu, il, iu, abstol, m, w, z, ldz, work, iwork,
               ifail, info )
call dsbgvx ( jobz, range, uplo, n, ka, kb, ab, ldab, bb, ldbb, q,
               ldq, vl, vu, il, iu, abstol, m, w, z, ldz, work, iwork,
               ifail, info )
```

### Discussion

This routine computes selected eigenvalues, and optionally, the eigenvectors of a real generalized symmetric-definite banded eigenproblem, of the form  $Ax = \lambda Bx$ . Here  $A$  and  $B$  are assumed to be symmetric and banded, and  $B$  is also positive definite.

Eigenvalues and eigenvectors can be selected by specifying either all eigenvalues, a range of values or a range of indices for the desired eigenvalues.

### Input Parameters

<i>jobz</i>	CHARACTER*1. Must be ' <b>N</b> ' or ' <b>V</b> '. If <i>jobz</i> = ' <b>N</b> ', then compute eigenvalues only. If <i>jobz</i> = ' <b>V</b> ', then compute eigenvalues and eigenvectors.
<i>range</i>	CHARACTER*1. Must be ' <b>A</b> ' or ' <b>V</b> ' or ' <b>I</b> '. If <i>range</i> = ' <b>A</b> ', the routine computes all eigenvalues. If <i>range</i> = ' <b>V</b> ', the routine computes eigenvalues $\lambda_i$ in the half-open interval: $vl < \lambda_i \leq vu$ . If <i>range</i> = ' <b>I</b> ', the routine computes eigenvalues with indices <i>il</i> to <i>iu</i> .
<i>uplo</i>	CHARACTER*1. Must be ' <b>U</b> ' or ' <b>L</b> '.

	If <i>uplo</i> = 'U', arrays <i>ab</i> and <i>bb</i> store the upper triangles of <i>A</i> and <i>B</i> ; If <i>uplo</i> = 'L', arrays <i>ab</i> and <i>bb</i> store the lower triangles of <i>A</i> and <i>B</i> .
<i>n</i>	<b>INTEGER.</b> The order of the matrices <i>A</i> and <i>B</i> ( <i>n</i> ≥ 0).
<i>ka</i>	<b>INTEGER.</b> The number of super- or sub-diagonals in <i>A</i> ( <i>ka</i> ≥ 0).
<i>kb</i>	<b>INTEGER.</b> The number of super- or sub-diagonals in <i>B</i> ( <i>kb</i> ≥ 0).
<i>ab, bb, work</i>	<b>REAL</b> for <i>ssbgvx</i> <b>DOUBLE PRECISION</b> for <i>dsbgvx</i> Arrays: <i>ab</i> ( <i>ldab</i> , *) is an array containing either upper or lower triangular part of the symmetric matrix <i>A</i> (as specified by <i>uplo</i> ) in band storage format. The second dimension of the array <i>ab</i> must be at least $\max(1, n)$ . <i>bb</i> ( <i>ldbb</i> , *) is an array containing either upper or lower triangular part of the symmetric matrix <i>B</i> (as specified by <i>uplo</i> ) in band storage format. The second dimension of the array <i>bb</i> must be at least $\max(1, n)$ . <i>work</i> (*) is a workspace array, <b>DIMENSION</b> at least $\max(1, 7n)$ .
<i>ldab</i>	<b>INTEGER.</b> The first dimension of the array <i>ab</i> ; must be at least <i>ka</i> +1.
<i>ldbb</i>	<b>INTEGER.</b> The first dimension of the array <i>bb</i> ; must be at least <i>kb</i> +1.
<i>vl, vu</i>	<b>REAL</b> for <i>ssbgvx</i> <b>DOUBLE PRECISION</b> for <i>dsbgvx</i> . If <i>range</i> = 'V', the lower and upper bounds of the interval to be searched for eigenvalues. Constraint: <i>vl</i> < <i>vu</i> . If <i>range</i> = 'A' or 'I', <i>vl</i> and <i>vu</i> are not referenced.

---

<i>il, iu</i>	<b>INTEGER.</b> If <i>range</i> = 'I', the indices in ascending order of the smallest and largest eigenvalues to be returned. Constraint: $1 \leq il \leq iu \leq n$ , if <i>n</i> > 0; <i>il</i> =1 and <i>iu</i> =0 if <i>n</i> = 0. If <i>range</i> = 'A' or 'V', <i>il</i> and <i>iu</i> are not referenced.
<i>abstol</i>	<b>REAL</b> for <i>ssbgvx</i> <b>DOUBLE PRECISION</b> for <i>dsbgvx</i> . The absolute error tolerance for the eigenvalues. See <i>Application Notes</i> for more information.
<i>ldz</i>	<b>INTEGER.</b> The leading dimension of the output array <i>z</i> ; <i>ldz</i> $\geq 1$ . If <i>jobz</i> = 'V', <i>ldz</i> $\geq \max(1, n)$ .
<i>ldq</i>	<b>INTEGER.</b> The leading dimension of the output array <i>q</i> ; <i>ldq</i> $\geq 1$ . If <i>jobz</i> = 'V', <i>ldq</i> $\geq \max(1, n)$ .
<i>iwork</i>	<b>INTEGER.</b> Workspace array, <b>DIMENSION</b> at least $\max(1, 5n)$ .

## Output Parameters

<i>ab</i>	On exit, the contents of <i>ab</i> are overwritten.
<i>bb</i>	On exit, contains the factor <i>S</i> from the split Cholesky factorization $B = S^T S$ , as returned by <i>s pbstf/dpbstf</i> .
<i>m</i>	<b>INTEGER.</b> The total number of eigenvalues found, $0 \leq m \leq n$ . If <i>range</i> = 'A', <i>m</i> = <i>n</i> , and if <i>range</i> = 'I', <i>m</i> = <i>iu</i> - <i>il</i> + 1.
<i>w, z, q</i>	<b>REAL</b> for <i>ssbgvx</i> <b>DOUBLE PRECISION</b> for <i>dsbgvx</i> Arrays: <i>w</i> ( * ), <b>DIMENSION</b> at least $\max(1, n)$ . If <i>info</i> = 0, contains the eigenvalues in ascending order. <i>z</i> ( <i>ldz</i> , * ). The second dimension of <i>z</i> must be at least $\max(1, n)$ . If <i>jobz</i> = 'V', then if <i>info</i> = 0, <i>z</i> contains the matrix <i>Z</i> of eigenvectors, with the <i>i</i> -th column of <i>z</i> holding the eigenvector associated with <i>w</i> ( <i>i</i> ). The eigenvectors are

normalized so that  $Z^T B Z = I$ .

If  $\text{jobz} = 'N'$ , then  $\text{z}$  is not referenced.

$\text{q}(ldq, *)$ . The second dimension of  $\text{q}$  must be at least  $\max(1, n)$ .

If  $\text{jobz} = 'V'$ , then  $\text{q}$  contains the  $n$ -by- $n$  matrix used in the reduction of  $Ax = \lambda Bx$  to standard form, that is,  $Cx = \lambda x$  and consequently  $C$  to tridiagonal form.

If  $\text{jobz} = 'N'$ , then  $\text{q}$  is not referenced.

*ifail*

`INTEGER`.

Array, `DIMENSION` at least  $\max(1, n)$ .

If  $\text{jobz} = 'V'$ , then if  $\text{info} = 0$ , the first  $m$  elements of *ifail* are zero; if  $\text{info} > 0$ , the *ifail* contains the indices of the eigenvectors that failed to converge.

If  $\text{jobz} = 'N'$ , then *ifail* is not referenced.

*info*

`INTEGER`.

If  $\text{info} = 0$ , the execution is successful.

If  $\text{info} = -i$ , the  $i$ th argument had an illegal value.

If  $\text{info} > 0$ , and

if  $i \leq n$ , the algorithm failed to converge, and  $i$  off-diagonal elements of an intermediate tridiagonal did not converge to zero;

if  $\text{info} = n + i$ , for  $1 \leq i \leq n$ , then `spbstf/dpbstf` returned  $\text{info} = i$  and  $B$  is not positive-definite. The factorization of  $B$  could not be completed and no eigenvalues or eigenvectors were computed.

## Application Notes

An approximate eigenvalue is accepted as converged when it is determined to lie in an interval  $[a,b]$  of width less than or equal to

$\text{abstol} + \epsilon * \max(|a|, |b|)$ , where  $\epsilon$  is the machine precision. If `abstol` is less than or equal to zero, then  $\epsilon * \|T\|_1$  will be used in its place, where  $T$  is the tridiagonal matrix obtained by reducing  $A$  to tridiagonal form.

Eigenvalues will be computed most accurately when `abstol` is set to twice the underflow threshold  $2 * ?lamch('S')$ , not zero. If this routine returns with  $\text{info} > 0$ , indicating that some eigenvectors did not converge, try setting `abstol` to  $2 * ?lamch('S')$ .

## ?hbgvx

*Computes selected eigenvalues and, optionally, eigenvectors of a complex generalized Hermitian definite eigenproblem with banded matrices.*

```
call chbgvx ( jobz, range, uplo, n, ka, kb, ab, ldab, bb, ldbb, q,
               ldq, vl, vu, il, iu, abstol, m, w, z, ldz, work, rwork,
               iwork, ifail, info )
call zhbgvx ( jobz, range, uplo, n, ka, kb, ab, ldab, bb, ldbb, q,
               ldq, vl, vu, il, iu, abstol, m, w, z, ldz, work, rwork,
               iwork, ifail, info )
```

### Discussion

This routine computes selected eigenvalues, and optionally, the eigenvectors of a complex generalized Hermitian-definite banded eigenproblem, of the form  $Ax = \lambda Bx$ . Here  $A$  and  $B$  are assumed to be Hermitian and banded, and  $B$  is also positive definite.

Eigenvalues and eigenvectors can be selected by specifying either all eigenvalues, a range of values or a range of indices for the desired eigenvalues.

### Input Parameters

<i>jobz</i>	CHARACTER*1. Must be ' <b>N</b> ' or ' <b>V</b> '. If <i>jobz</i> = ' <b>N</b> ', then compute eigenvalues only. If <i>jobz</i> = ' <b>V</b> ', then compute eigenvalues and eigenvectors.
<i>range</i>	CHARACTER*1. Must be ' <b>A</b> ' or ' <b>V</b> ' or ' <b>I</b> '. If <i>range</i> = ' <b>A</b> ', the routine computes all eigenvalues. If <i>range</i> = ' <b>V</b> ', the routine computes eigenvalues $\lambda_i$ in the half-open interval: $vl < \lambda_i \leq vu$ . If <i>range</i> = ' <b>I</b> ', the routine computes eigenvalues with indices <i>il</i> to <i>iu</i> .
<i>uplo</i>	CHARACTER*1. Must be ' <b>U</b> ' or ' <b>L</b> '.

If  $\text{uplo} = \text{'U'}$ , arrays  $\text{ab}$  and  $\text{bb}$  store the upper triangles of  $A$  and  $B$ ;

If  $\text{uplo} = \text{'L'}$ , arrays  $\text{ab}$  and  $\text{bb}$  store the lower triangles of  $A$  and  $B$ .

$n$  INTEGER. The order of the matrices  $A$  and  $B$  ( $n \geq 0$ ).

$ka$  INTEGER. The number of super- or sub-diagonals in  $A$  ( $ka \geq 0$ ).

$kb$  INTEGER. The number of super- or sub-diagonals in  $B$  ( $kb \geq 0$ ).

$ab, bb, work$  COMPLEX for `chbgvx`

DOUBLE COMPLEX for `zhbgvx`

Arrays:

$ab$  ( $l\text{dab}, *$ ) is an array containing either upper or lower triangular part of the Hermitian matrix  $A$  (as specified by  $\text{uplo}$ ) in band storage format.

The second dimension of the array  $ab$  must be at least  $\max(1, n)$ .

$bb$  ( $l\text{dbb}, *$ ) is an array containing either upper or lower triangular part of the Hermitian matrix  $B$  (as specified by  $\text{uplo}$ ) in band storage format.

The second dimension of the array  $bb$  must be at least  $\max(1, n)$ .

$work(*)$  is a workspace array, DIMENSION at least  $\max(1, n)$ .

$l\text{dab}$  INTEGER. The first dimension of the array  $ab$ ; must be at least  $ka+1$ .

$l\text{dbb}$  INTEGER. The first dimension of the array  $bb$ ; must be at least  $kb+1$ .

$vl, vu$  REAL for `chbgvx`

DOUBLE PRECISION for `zhbgvx`.

If  $\text{range} = \text{'V'}$ , the lower and upper bounds of the interval to be searched for eigenvalues.

Constraint:  $vl < vu$ .

If  $\text{range} = \text{'A}'$  or  $\text{'I'}$ ,  $vl$  and  $vu$  are not referenced.

<i>il, iu</i>	<b>INTEGER.</b> If <i>range</i> = 'I', the indices in ascending order of the smallest and largest eigenvalues to be returned. Constraint: $1 \leq il \leq iu \leq n$ , if <i>n</i> > 0; <i>il</i> =1 and <i>iu</i> =0 if <i>n</i> = 0. If <i>range</i> = 'A' or 'V', <i>il</i> and <i>iu</i> are not referenced.
<i>abstol</i>	<b>REAL</b> for <i>chbgvx</i> <b>DOUBLE PRECISION</b> for <i>zhbgvx</i> . The absolute error tolerance for the eigenvalues. See <i>Application Notes</i> for more information.
<i>ldz</i>	<b>INTEGER.</b> The leading dimension of the output array <i>z</i> ; <i>ldz</i> $\geq 1$ . If <i>jobz</i> = 'V', <i>ldz</i> $\geq \max(1, n)$ .
<i>ldq</i>	<b>INTEGER.</b> The leading dimension of the output array <i>q</i> ; <i>ldq</i> $\geq 1$ . If <i>jobz</i> = 'V', <i>ldq</i> $\geq \max(1, n)$ .
<i>rwork</i>	<b>REAL</b> for <i>chbgvx</i> <b>DOUBLE PRECISION</b> for <i>zhbgvx</i> . Workspace array, <b>DIMENSION</b> at least $\max(1, 7n)$ .
<i>iwork</i>	<b>INTEGER.</b> Workspace array, <b>DIMENSION</b> at least $\max(1, 5n)$ .

## Output Parameters

<i>ab</i>	On exit, the contents of <i>ab</i> are overwritten.
<i>bb</i>	On exit, contains the factor <i>S</i> from the split Cholesky factorization $B = S^H S$ , as returned by <i>cpbstf/zpbstf</i> .
<i>m</i>	<b>INTEGER.</b> The total number of eigenvalues found, $0 \leq m \leq n$ . If <i>range</i> = 'A', <i>m</i> = <i>n</i> , and if <i>range</i> = 'I', <i>m</i> = <i>iu</i> - <i>il</i> +1.
<i>w</i>	<b>REAL</b> for <i>chbgvx</i> <b>DOUBLE PRECISION</b> for <i>zhbgvx</i> . Array <i>w</i> (*), <b>DIMENSION</b> at least $\max(1, n)$ . If <i>info</i> = 0, contains the eigenvalues in ascending order.
<i>z, q</i>	<b>COMPLEX</b> for <i>chbgvx</i> <b>DOUBLE COMPLEX</b> for <i>zhbgvx</i> Arrays:

$\mathbf{z}(\mathbf{ldz}, *)$ . The second dimension of  $\mathbf{z}$  must be at least  $\max(1, \mathbf{n})$ .

If  $\mathbf{jobz} = 'V'$ , then if  $\mathbf{info} = 0$ ,  $\mathbf{z}$  contains the matrix  $Z$  of eigenvectors, with the  $i$ -th column of  $\mathbf{z}$  holding the eigenvector associated with  $w(i)$ . The eigenvectors are normalized so that  $Z^H B Z = I$ .

If  $\mathbf{jobz} = 'N'$ , then  $\mathbf{z}$  is not referenced.

$\mathbf{q}(\mathbf{ldq}, *)$ . The second dimension of  $\mathbf{q}$  must be at least  $\max(1, \mathbf{n})$ .

If  $\mathbf{jobz} = 'V'$ , then  $\mathbf{q}$  contains the  $n$ -by- $n$  matrix used in the reduction of  $Ax = \lambda Bx$  to standard form, that is,  $Cx = \lambda x$  and consequently  $C$  to tridiagonal form.

If  $\mathbf{jobz} = 'N'$ , then  $\mathbf{q}$  is not referenced.

#### *ifail*

**INTEGER.**

Array, **DIMENSION** at least  $\max(1, \mathbf{n})$ .

If  $\mathbf{jobz} = 'V'$ , then if  $\mathbf{info} = 0$ , the first  $m$  elements of *ifail* are zero; if  $\mathbf{info} > 0$ , the *ifail* contains the indices of the eigenvectors that failed to converge.

If  $\mathbf{jobz} = 'N'$ , then *ifail* is not referenced.

#### *info*

**INTEGER.**

If  $\mathbf{info} = 0$ , the execution is successful.

If  $\mathbf{info} = -i$ , the  $i$ th argument had an illegal value.

If  $\mathbf{info} > 0$ , and

if  $i \leq n$ , the algorithm failed to converge, and  $i$  off-diagonal elements of an intermediate tridiagonal did not converge to zero;

if  $\mathbf{info} = n + i$ , for  $1 \leq i \leq n$ , then *cpbstf/zpbstf* returned  $\mathbf{info} = i$  and  $B$  is not positive-definite. The factorization of  $B$  could not be completed and no eigenvalues or eigenvectors were computed.

## Application Notes

An approximate eigenvalue is accepted as converged when it is determined to lie in an interval  $[a,b]$  of width less than or equal to

$\mathbf{abstol} + \epsilon * \max(|a|, |b|)$ , where  $\epsilon$  is the machine precision. If  $\mathbf{abstol}$  is less than or equal to zero, then  $\epsilon * \|T\|_1$  will be used in its place, where  $T$

is the tridiagonal matrix obtained by reducing  $A$  to tridiagonal form.  
Eigenvalues will be computed most accurately when  $\text{abstol}$  is set to twice  
the underflow threshold  $2 * \text{?lamch}('S')$ , not zero. If this routine returns with  
 $\text{info} > 0$ , indicating that some eigenvectors did not converge, try setting  
 $\text{abstol}$  to  $2 * \text{?lamch}('S')$ .

## Generalized Nonsymmetric Eigenproblems

This section describes LAPACK driver routines used for solving generalized nonsymmetric eigenproblems. See also [computational routines](#) that can be called to solve these problems.

[Table 5-14](#) lists routines described in more detail below.

**Table 5-14    Driver Routines for Solving Generalized Nonsymmetric Eigenproblems**

Routine Name	Operation performed
<a href="#">?gges</a>	Computes the generalized eigenvalues, Shur form, and the left and/or right Shur vectors for a pair of nonsymmetric matrices.
<a href="#">?ggesx</a>	Computes the generalized eigenvalues, Shur form, and, optionally, the left and/or right matrices of Shur vectors .
<a href="#">?ggev</a>	Computes the generalized eigenvalues, and the left and/or right generalized eigenvectors for a pair of nonsymmetric matrices.
<a href="#">?ggevx</a>	Computes the generalized eigenvalues, and, optionally, the left and/or right generalized eigenvectors.

---

### ?gges

*Computes the generalized eigenvalues, Shur form, and the left and/or right Shur vectors for a pair of nonsymmetric matrices.*

---

```
call sgges ( jobvsl, jobvsr, sort, selctg, n, a, lda, b, ldb, sdim,
            alphar, alphai, beta, vsl, ldvsl, vsr, ldvsr, work,
            lwork, bwork, info )
call dgges ( jobvsl, jobvsr, sort, selctg, n, a, lda, b, ldb, sdim,
            alphar, alphai, beta, vsl, ldvsl, vsr, ldvsr, work,
            lwork, bwork, info )
call cgges ( jobvsl, jobvsr, sort, selctg, n, a, lda, b, ldb, sdim,
            alpha, beta, vsl, ldvsl, vsr, ldvsr, work, lwork, rwork,
            bwork, info )
```

```
call zggsl( jobvsl, jobvsr, sort, selctg, n, a, lda, b, ldb, sdim,
            alpha, beta, vsl, ldvsl, vsr, ldvsr, work, lwork, rwork,
            bwork, info )
```

### Discussion

This routine computes for a pair of  $n$ -by- $n$  real/complex nonsymmetric matrices  $(A, B)$ , the generalized eigenvalues, the generalized real/complex Schur form  $(S, T)$ , optionally, the left and/or right matrices of Schur vectors ( $vsl$  and  $vsr$ ). This gives the generalized Schur factorization

$$(A, B) = (vsl * S * vsr^H, vsl * T * vsr^H)$$

Optionally, it also orders the eigenvalues so that a selected cluster of eigenvalues appears in the leading diagonal blocks of the upper quasi-triangular matrix  $S$  and the upper triangular matrix  $T$ . The leading columns of  $vsl$  and  $vsr$  then form an orthonormal/unitary basis for the corresponding left and right eigenspaces (deflating subspaces).

(If only the generalized eigenvalues are needed, use the driver [?ggev](#) instead, which is faster.)

A generalized eigenvalue for a pair of matrices  $(A, B)$  is a scalar  $w$  or a ratio  $\alpha / \beta = w$ , such that  $A - wB$  is singular. It is usually represented as the pair  $(\alpha, \beta)$ , as there is a reasonable interpretation for  $\beta=0$  or for both being zero.

A pair of matrices  $(S, T)$  is in generalized real Schur form if  $T$  is upper triangular with non-negative diagonal and  $S$  is block upper triangular with 1-by-1 and 2-by-2 blocks. 1-by-1 blocks correspond to real generalized eigenvalues, while 2-by-2 blocks of  $S$  will be “standardized” by making the corresponding elements of  $T$  have the form:

$$\begin{pmatrix} a & 0 \\ 0 & b \end{pmatrix}$$

and the pair of corresponding 2-by-2 blocks in  $S$  and  $T$  will have a complex conjugate pair of generalized eigenvalues.

A pair of matrices  $(S, T)$  is in generalized complex Schur form if  $S$  and  $T$  are upper triangular and, in addition, the diagonal of  $T$  are non-negative real numbers.

## Input Parameters

<i>jobvs1</i>	CHARACTER*1. Must be ' <b>N</b> ' or ' <b>V</b> '. If <i>jobvs1</i> = ' <b>N</b> ', then the left Shur vectors are not computed. If <i>jobvs1</i> = ' <b>V</b> ', then the left Shur vectors are computed.
<i>jobvsr</i>	CHARACTER*1. Must be ' <b>N</b> ' or ' <b>V</b> '. If <i>jobvsr</i> = ' <b>N</b> ', then the right Shur vectors are not computed. If <i>jobvsr</i> = ' <b>V</b> ', then the right Shur vectors are computed.
<i>sort</i>	CHARACTER*1. Must be ' <b>N</b> ' or ' <b>S</b> '. Specifies whether or not to order the eigenvalues on the diagonal of the generalized Schur form. If <i>sort</i> = ' <b>N</b> ', then eigenvalues are not ordered. If <i>sort</i> = ' <b>S</b> ', eigenvalues are ordered (see <i>selctg</i> ).
<i>selctg</i>	LOGICAL FUNCTION of three REAL arguments for real flavors. LOGICAL FUNCTION of two COMPLEX arguments for complex flavors.  <i>selctg</i> must be declared EXTERNAL in the calling subroutine. If <i>sort</i> = ' <b>S</b> ', <i>selctg</i> is used to select eigenvalues to sort to the top left of the Shur form. If <i>sort</i> = ' <b>N</b> ', <i>selctg</i> is not referenced.  <i>For real flavors:</i> An eigenvalue ( <i>alphar</i> (j) + <i>alphaai</i> (j))/ <i>beta</i> (j) is selected if <i>selctg</i> ( <i>alphar</i> (j), <i>alphaai</i> (j), <i>beta</i> (j)) is true; that is, if either one of a complex conjugate pair of eigenvalues is selected, then both complex eigenvalues are selected. Note that in the ill-conditioned case, a selected complex eigenvalue may no longer satisfy <i>selctg</i> ( <i>alphar</i> (j), <i>alphaai</i> (j), <i>beta</i> (j)) = .TRUE.. after ordering. In this case <i>info</i> is set to <i>n</i> +2 .

For complex flavors:

An eigenvalue `alpha(j) / beta(j)` is selected if  
`selctg(alpha(j), beta(j))` is true.

Note that a selected complex eigenvalue may no longer satisfy `selctg(alpha(j), beta(j)) = .TRUE.` after ordering, since ordering may change the value of complex eigenvalues (especially if the eigenvalue is ill-conditioned); in this case `info` is set to `n+2` (see `info` below).

`n` INTEGER. The order of the matrices `A`, `B`, `vsl`, and `vsr` ( $n \geq 0$ ).

`a, b, work` REAL for `sgges`  
 DOUBLE PRECISION for `dgges`  
 COMPLEX for `cgges`  
 DOUBLE COMPLEX for `zgges`.

Arrays:

`a( lda, * )` is an array containing the  $n$ -by- $n$  matrix `A` (first of the pair of matrices).

The second dimension of `a` must be at least  $\max(1, n)$ .

`b( ldb, * )` is an array containing the  $n$ -by- $n$  matrix `B` (second of the pair of matrices).

The second dimension of `b` must be at least  $\max(1, n)$ .

`work(lwork)` is a workspace array.

`lda` INTEGER. The first dimension of the array `a`.  
 Must be at least  $\max(1, n)$ .

`ldb` INTEGER. The first dimension of the array `b`.  
 Must be at least  $\max(1, n)$ .

`ldvsl, ldvsr` INTEGER. The first dimensions of the output matrices `vsl` and `vsr`, respectively. Constraints:  
 $ldvsl \geq 1$ . If `jobvsl = 'V'`,  $ldvsl \geq \max(1, n)$ .  
 $ldvsr \geq 1$ . If `jobvsr = 'V'`,  $ldvsr \geq \max(1, n)$ .

`lwork` INTEGER. The dimension of the array `work`.

	$lwork \geq \max(1, 8n+16)$ for real flavors; $lwork \geq \max(1, 2n)$ for complex flavors. For good performance, $lwork$ must generally be larger.
$rwork$	<b>REAL</b> for <code>cgges</code> <b>DOUBLE PRECISION</b> for <code>zgges</code> Workspace array, <b>DIMENSION</b> at least $\max(1, 8n)$ . This array is used in complex flavors only.
$bwork$	<b>LOGICAL</b> . Workspace array, <b>DIMENSION</b> at least $\max(1, n)$ . Not referenced if $sort = 'N'$ .

## Output Parameters

$a$	On exit, this array has been overwritten by its generalized Shur form $S$ .
$b$	On exit, this array has been overwritten by its generalized Shur form $T$ .
$sdim$	<b>INTEGER</b> . If $sort = 'N'$ , $sdim = 0$ . If $sort = 'S'$ , $sdim$ is equal to the number of eigenvalues (after sorting) for which <code>selctg</code> is true. Note that for real flavors complex conjugate pairs for which <code>selctg</code> is true for either eigenvalue count as 2.
$alphar, alphai$	<b>REAL</b> for <code>sgges</code> ; <b>DOUBLE PRECISION</b> for <code>dgges</code> . Arrays, <b>DIMENSION</b> at least $\max(1, n)$ each. Contain values that form generalized eigenvalues in real flavors. See <code>beta</code> .
$alpha$	<b>COMPLEX</b> for <code>cgges</code> ; <b>DOUBLE COMPLEX</b> for <code>zgges</code> . Array, <b>DIMENSION</b> at least $\max(1, n)$ . Contain values that form generalized eigenvalues in complex flavors. See <code>beta</code> .
$beta$	<b>REAL</b> for <code>sgges</code> <b>DOUBLE PRECISION</b> for <code>dgges</code> <b>COMPLEX</b> for <code>cgges</code>

**DOUBLE COMPLEX** for **zgges**.

Array, **DIMENSION** at least  $\max(1, n)$ .

*For real flavors:*

On exit,  $(\text{alphar}(j) + \text{alphaai}(j)*i)/\text{beta}(j)$ ,  $j=1,\dots,n$ , will be the generalized eigenvalues.

$\text{alphar}(j) + \text{alphaai}(j)*i$  and  $\text{beta}(j)$ ,  $j=1,\dots,n$  are the diagonals of the complex Schur form  $(S, T)$  that would result if the 2-by-2 diagonal blocks of the real generalized Schur form of  $(A, B)$  were further reduced to triangular form using complex unitary transformations. If  $\text{alphaai}(j)$  is zero, then the  $j$ -th eigenvalue is real; if positive, then the  $j$ -th and  $(j+1)$ -st eigenvalues are a complex conjugate pair, with  $\text{alphaai}(j+1)$  negative.

*For complex flavors:*

On exit,  $\text{alpha}(j)/\text{beta}(j)$ ,  $j=1,\dots,n$ , will be the generalized eigenvalues.  $\text{alpha}(j)$ ,  $j=1,\dots,n$ , and  $\text{beta}(j)$ ,  $j=1,\dots,n$ , are the diagonals of the complex Schur form  $(S, T)$  output by **cgges/zgges**. The  $\text{beta}(j)$  will be non-negative real.

See also *Application Notes* below.

**vsl, vsr**

**REAL** for **sgges**

**DOUBLE PRECISION** for **dgges**

**COMPLEX** for **cgges**

**DOUBLE COMPLEX** for **zgges**.

Arrays:

**vsl**(*ldvsl, \**), the second dimension of **vsl** must be at least  $\max(1, n)$ .

If **jobvsl** = 'V', this array will contain the left Shur vectors.

If **jobvsl** = 'N', **vsl** is not referenced.

**vsr**(*ldvsr, \**), the second dimension of **vsr** must be at least  $\max(1, n)$ .

If **jobvsr** = 'V', this array will contain the right Shur vectors.

If **jobvsr** = 'N', **vsr** is not referenced.

<i>work(1)</i>	On exit, if <i>info</i> = 0, then <i>work(1)</i> returns the required minimal size of <i>lwork</i> .
<i>info</i>	<i>INTEGER.</i> If <i>info</i> = 0, the execution is successful. If <i>info</i> = <i>-i</i> , the <i>i</i> th parameter had an illegal value. If <i>info</i> = <i>i</i> , and <i>i</i> ≤ <i>n</i> : the <i>QZ</i> iteration failed. ( <i>A,B</i> ) is not in Schur form, but <i>alphar(j)</i> , <i>alphai(j)</i> (for real flavors), or <i>alpha(j)</i> (for complex flavors), and <i>beta(j)</i> , <i>j</i> = <i>info</i> +1,..., <i>n</i> should be correct. <i>i</i> > <i>n</i> : errors that usually indicate LAPACK problems: <i>i</i> = <i>n</i> +1: other than <i>QZ</i> iteration failed in ? <i>hgeqz</i> ; <i>i</i> = <i>n</i> +2: after reordering, roundoff changed values of some complex eigenvalues so that leading eigenvalues in the generalized Schur form no longer satisfy <i>selectg</i> = .TRUE.. This could also be caused due to scaling; <i>i</i> = <i>n</i> +3: reordering failed in ? <i>tgsen</i> .

## Application Notes

If you are in doubt how much workspace to supply for the array *work*, use a generous value of *lwork* for the first run. On exit, examine *work(1)* and use this value for subsequent runs.

The quotients *alphar(j)/beta(j)* and *alphai(j)/beta(j)* may easily over- or underflow, and *beta(j)* may even be zero. Thus, you should avoid simply computing the ratio. However, *alphar* and *alphai* will be always less than and usually comparable with *norm(A)* in magnitude, and *beta* always less than and usually comparable with *norm(B)*.

## ?ggesx

*Computes the generalized eigenvalues,  
Shur form, and, optionally, the left  
and/or right matrices of Shur vectors .*

```
call sggesx (jobvsl, jobvsr, sort, selctg, sense, n, a, lda, b, ldb,
            sdim, alphar, alphai, beta, vsl, ldvsl, vsr, ldvsr,
            rconde, rcondv, work, lwork, iwork, liwork, bwork, info )
call dggesx (jobvsl, jobvsr, sort, selctg, sense, n, a, lda, b, ldb,
            sdim, alphar, alphai, beta, vsl, ldvsl, vsr, ldvsr,
            rconde, rcondv, work, lwork, iwork, liwork, bwork, info )
call cggesx (jobvsl, jobvsr, sort, selctg, sense, n, a, lda, b, ldb,
            sdim, alpha, beta, vsl, ldvsl, vsr, ldvsr, rconde, rcondv,
            work, lwork, rwork, iwork, liwork, bwork, info )
call zggesx (jobvsl, jobvsr, sort, selctg, sense, n, a, lda, b, ldb,
            sdim, alpha, beta, vsl, ldvsl, vsr, ldvsr, rconde, rcondv,
            work, lwork, rwork, iwork, liwork, bwork, info )
```

### Discussion

This routine computes for a pair of  $n$ -by- $n$  real/complex nonsymmetric matrices  $(A, B)$ , the generalized eigenvalues, the generalized real/complex Schur form  $(S, T)$ , optionally, the left and/or right matrices of Schur vectors ( $vsl$  and  $vsr$ ). This gives the generalized Schur factorization

$$(A, B) = (vsl * S * vsr^H, vsl * T * vsr^H)$$

Optionally, it also orders the eigenvalues so that a selected cluster of eigenvalues appears in the leading diagonal blocks of the upper quasi-triangular matrix  $S$  and the upper triangular matrix  $T$ ; computes a reciprocal condition number for the average of the selected eigenvalues ( $rconde$ ); and computes a reciprocal condition number for the right and left deflating subspaces corresponding to the selected eigenvalues ( $rcondv$ ). The leading columns of  $vsl$  and  $vsr$  then form an orthonormal/unitary basis for the corresponding left and right eigenspaces (deflating subspaces).

A generalized eigenvalue for a pair of matrices ( $A, B$ ) is a scalar  $w$  or a ratio  $\alpha / \beta = w$ , such that  $A - w^*B$  is singular. It is usually represented as the pair ( $\alpha, \beta$ ), as there is a reasonable interpretation for  $\beta=0$  or for both being zero.

A pair of matrices ( $S, T$ ) is in generalized real Schur form if  $T$  is upper triangular with non-negative diagonal and  $S$  is block upper triangular with 1-by-1 and 2-by-2 blocks. 1-by-1 blocks correspond to real generalized eigenvalues, while 2-by-2 blocks of  $S$  will be “standardized” by making the corresponding elements of  $T$  have the form:

$$\begin{pmatrix} a & 0 \\ 0 & b \end{pmatrix}$$

and the pair of corresponding 2-by-2 blocks in  $S$  and  $T$  will have a complex conjugate pair of generalized eigenvalues.

A pair of matrices ( $S, T$ ) is in generalized complex Schur form if  $S$  and  $T$  are upper triangular and, in addition, the diagonal of  $T$  are non-negative real numbers.

## Input Parameters

<i>jobvs1</i>	CHARACTER*1. Must be ' <b>N</b> ' or ' <b>V</b> '. If <i>jobvs1</i> = ' <b>N</b> ', then the left Shur vectors are not computed. If <i>jobvs1</i> = ' <b>V</b> ', then the left Shur vectors are computed.
<i>jobvsr</i>	CHARACTER*1. Must be ' <b>N</b> ' or ' <b>V</b> '. If <i>jobvsr</i> = ' <b>N</b> ', then the right Shur vectors are not computed. If <i>jobvsr</i> = ' <b>V</b> ', then the right Shur vectors are computed.
<i>sort</i>	CHARACTER*1. Must be ' <b>N</b> ' or ' <b>S</b> '. Specifies whether or not to order the eigenvalues on the diagonal of the generalized Schur form. If <i>sort</i> = ' <b>N</b> ', then eigenvalues are not ordered. If <i>sort</i> = ' <b>S</b> ', eigenvalues are ordered (see <i>selectg</i> ).

---

<i>selctg</i>	<p><b>LOGICAL FUNCTION</b> of three <b>REAL</b> arguments for real flavors.</p> <p><b>LOGICAL FUNCTION</b> of two <b>COMPLEX</b> arguments for complex flavors.</p> <p><i>selctg</i> must be declared <b>EXTERNAL</b> in the calling subroutine.</p> <p>If <i>sort</i> = '<b>S</b>', <i>selctg</i> is used to select eigenvalues to sort to the top left of the Shur form.</p> <p>If <i>sort</i> = '<b>N</b>', <i>selctg</i> is not referenced.</p> <p><i>For real flavors:</i></p> <p>An eigenvalue (<i>alphar</i>(j) + <i>alphai</i>(j))/<i>beta</i>(j) is selected if <i>selctg</i>(<i>alphar</i>(j), <i>alphai</i>(j), <i>beta</i>(j)) is true; that is, if either one of a complex conjugate pair of eigenvalues is selected, then both complex eigenvalues are selected.</p> <p>Note that in the ill-conditioned case, a selected complex eigenvalue may no longer satisfy <i>selctg</i>(<i>alphar</i>(j), <i>alphai</i>(j), <i>beta</i>(j)) = .TRUE. after ordering. In this case <i>info</i> is set to <i>n</i>+2.</p> <p><i>For complex flavors:</i></p> <p>An eigenvalue <i>alpha</i>(j) / <i>beta</i>(j) is selected if <i>selctg</i>(<i>alpha</i>(j), <i>beta</i>(j)) is true.</p> <p>Note that a selected complex eigenvalue may no longer satisfy <i>selctg</i>(<i>alpha</i>(j), <i>beta</i>(j)) = .TRUE. after ordering, since ordering may change the value of complex eigenvalues (especially if the eigenvalue is ill-conditioned); in this case <i>info</i> is set to <i>n</i>+2 (see <i>info</i> below).</p>
<i>sense</i>	<p><b>CHARACTER*1</b>. Must be '<b>N</b>', '<b>E</b>', '<b>V</b>', or '<b>B</b>'.</p> <p>Determines which reciprocal condition number are computed.</p> <p>If <i>sense</i> = '<b>N</b>', none are computed;</p> <p>If <i>sense</i> = '<b>E</b>', computed for average of selected eigenvalues only;</p> <p>If <i>sense</i> = '<b>V</b>', computed for selected deflating subspaces only;</p>

If `sense` = 'B', computed for both.

If `sense` is 'E', 'V', or 'B', then `sort` must equal 'S'.

`n` **INTEGER.** The order of the matrices  $A$ ,  $B$ ,  $vsl$ , and  $vsr$  ( $n \geq 0$ ).

`a, b, work` **REAL** for `sggesx`  
**DOUBLE PRECISION** for `dggex`  
**COMPLEX** for `cggex`  
**DOUBLE COMPLEX** for `zggex`.

Arrays:

`a( lda, * )` is an array containing the  $n$ -by- $n$  matrix  $A$  (first of the pair of matrices).

The second dimension of `a` must be at least  $\max(1, n)$ .

`b( ldb, * )` is an array containing the  $n$ -by- $n$  matrix  $B$  (second of the pair of matrices).

The second dimension of `b` must be at least  $\max(1, n)$ .

`work(lwork)` is a workspace array.

`lda` **INTEGER.** The first dimension of the array `a`.  
 Must be at least  $\max(1, n)$ .

`ldb` **INTEGER.** The first dimension of the array `b`.  
 Must be at least  $\max(1, n)$ .

`ldvsl, ldvsr` **INTEGER.** The first dimensions of the output matrices `vsl` and `vsr`, respectively. Constraints:  
 $ldvsl \geq 1$ . If `jobvsl` = 'V',  $ldvsl \geq \max(1, n)$ .  
 $ldvsr \geq 1$ . If `jobvsr` = 'V',  $ldvsr \geq \max(1, n)$ .

`lwork` **INTEGER.** The dimension of the array `work`.

For real flavors:

$lwork \geq \max(1, 8(n+1)+16)$ ;  
 if `sense` = 'E', 'V', or 'B', then  
 $lwork \geq \max( 8(n+1)+16, 2 * sdim * (n - sdim) )$ .

For complex flavors:

$lwork \geq \max(1, 2n)$ ;  
 if `sense` = 'E', 'V', or 'B', then  
 $lwork \geq \max( 2n, 2 * sdim * (n - sdim) )$ .

For good performance, *lwork* must generally be larger.

*rwork*

REAL for *cggexx*

DOUBLE PRECISION for *zggexx*

Workspace array, DIMENSION at least max(1, 8*n*).

This array is used in complex flavors only.

*iwork*

INTEGER.

Workspace array, DIMENSION (*liwork*). Not

referenced if *sense* = 'N'.

*liwork*

INTEGER. The dimension of the array *iwork*.

*liwork*  $\geq n+6$  for real flavors;

*liwork*  $\geq n+2$  for complex flavors.

*bwork*

LOGICAL.

Workspace array, DIMENSION at least max(1, *n*).

Not referenced if *sort* = 'N'.

## Output Parameters

*a*

On exit, this array has been overwritten by its generalized Shur form *S*.

*b*

On exit, this array has been overwritten by its generalized Shur form *T*.

*sdim*

INTEGER.

If *sort* = 'N', *sdim* = 0.

If *sort* = 'S', *sdim* is equal to the number of eigenvalues (after sorting) for which *selectg* is true.

Note that for real flavors complex conjugate pairs for which *selectg* is true for either eigenvalue count as 2.

*alphar*, *alphaai* REAL for *sggesx*;

DOUBLE PRECISION for *dggexx*.

Arrays, DIMENSION at least max(1, *n*) each. Contain values that form generalized eigenvalues in real flavors.

See *beta*.

<i>alpha</i>	COMPLEX for <code>cgesx</code> ; DOUBLE COMPLEX for <code>zgesx</code> . Array, DIMENSION at least max(1, <i>n</i> ). Contain values that form generalized eigenvalues in complex flavors. See <i>beta</i> .
<i>beta</i>	REAL for <code>sgesx</code> DOUBLE PRECISION for <code>dgesx</code> COMPLEX for <code>cgesx</code> DOUBLE COMPLEX for <code>zgesx</code> . Array, DIMENSION at least max(1, <i>n</i> ). <i>For real flavors:</i> On exit, ( <i>alphar</i> (j) + <i>alphai</i> (j)*i)/ <i>beta</i> (j), j=1,..., <i>n</i> , will be the generalized eigenvalues. <i>alphar</i> (j) + <i>alphai</i> (j)*i and <i>beta</i> (j), j=1,..., <i>n</i> are the diagonals of the complex Schur form ( <i>S,T</i> ) that would result if the 2-by-2 diagonal blocks of the real generalized Schur form of ( <i>A,B</i> ) were further reduced to triangular form using complex unitary transformations. If <i>alphai</i> (j) is zero, then the <i>j</i> -th eigenvalue is real; if positive, then the <i>j</i> -th and ( <i>j</i> +1)-st eigenvalues are a complex conjugate pair, with <i>alphai</i> ( <i>j</i> +1) negative. <i>For complex flavors:</i> On exit, <i>alpha</i> (j)/ <i>beta</i> (j), j=1,..., <i>n</i> , will be the generalized eigenvalues. <i>alpha</i> (j), j=1,..., <i>n</i> , and <i>beta</i> (j), j=1,..., <i>n</i> , are the diagonals of the complex Schur form ( <i>S,T</i> ) output by <code>cgesx/zgesx</code> . The <i>beta</i> (j) will be non-negative real. See also <i>Application Notes</i> below.
<i>vsl, vsr</i>	REAL for <code>sgesx</code> DOUBLE PRECISION for <code>dgesx</code> COMPLEX for <code>cgesx</code> DOUBLE COMPLEX for <code>zgesx</code> . Arrays: <i>vsl</i> ( <i>ldvsl,*</i> ), the second dimension of <i>vsl</i> must be at least max(1, <i>n</i> ).

If  $\text{jobvsl} = \text{'V'}$ , this array will contain the left Shur vectors.

If  $\text{jobvsl} = \text{'N'}$ ,  $\text{vsl}$  is not referenced.

$\text{vsr}(\text{ldvsr}, *)$ , the second dimension of  $\text{vsr}$  must be at least  $\max(1, n)$ .

If  $\text{jobvsr} = \text{'V'}$ , this array will contain the right Shur vectors.

If  $\text{jobvsr} = \text{'N'}$ ,  $\text{vsr}$  is not referenced.

$\text{rconde}, \text{rcondv}$  **REAL** for single precision flavors  
**DOUBLE PRECISION** for double precision flavors.  
 Arrays, **DIMENSION** (2) each

If  $\text{sense} = \text{'E'}$  or  $\text{'B'}$ ,  $\text{rconde}(1)$  and  $\text{rconde}(2)$  contain the reciprocal condition numbers for the average of the selected eigenvalues.

Not referenced if  $\text{sense} = \text{'N'}$  or  $\text{'V'}$ .

If  $\text{sense} = \text{'V'}$  or  $\text{'B'}$ ,  $\text{rcondv}(1)$  and  $\text{rcondv}(2)$  contain the reciprocal condition numbers for the selected deflating subspaces.

Not referenced if  $\text{sense} = \text{'N'}$  or  $\text{'E'}$ .

$\text{work}(1)$  On exit, if  $\text{info} = 0$ , then  $\text{work}(1)$  returns the required minimal size of  $\text{lwork}$ .

$\text{info}$

**INTEGER**.

If  $\text{info} = 0$ , the execution is successful.

If  $\text{info} = -i$ , the  $i$ th parameter had an illegal value.

If  $\text{info} = i$ , and

$i \leq n$  :

the  $QZ$  iteration failed.  $(A, B)$  is not in Shur form, but  $\text{alphar}(j)$ ,  $\text{alphai}(j)$  (for real flavors), or  $\text{alpha}(j)$  (for complex flavors), and  $\text{beta}(j)$ ,  $j = \text{info} + 1, \dots, n$  should be correct.

$i > n$  : errors that usually indicate LAPACK problems:

$i = n+1$ : other than  $QZ$  iteration failed in **?hgeqz**;

*i* = *n*+2: after reordering, roundoff changed values of some complex eigenvalues so that leading eigenvalues in the generalized Schur form no longer satisfy *selctg* = .TRUE.. This could also be caused due to scaling;

*i* = *n*+3: reordering failed in ?tgse.

### Application Notes

If you are in doubt how much workspace to supply for the array *work*, use a generous value of *lwork* for the first run. On exit, examine *work(1)* and use this value for subsequent runs.

The quotients *alphar(j)/beta(j)* and *alphai(j)/beta(j)* may easily over- or underflow, and *beta(j)* may even be zero. Thus, you should avoid simply computing the ratio. However, *alphar* and *alphai* will be always less than and usually comparable with *norm(A)* in magnitude, and *beta* always less than and usually comparable with *norm(B)*.

## ?ggev

*Computes the generalized eigenvalues,  
and the left and/or right generalized  
eigenvectors for a pair of nonsymmetric  
matrices.*

```
call sggev ( jobvl, jobvr, n, a, lda, b, ldb, alphar, alphai, beta,
             vl, ldvl, vr, ldvr, work, lwork, info )
call dggev ( jobvl, jobvr, n, a, lda, b, ldb, alphar, alphai, beta,
             vl, ldvl, vr, ldvr, work, lwork, info )
call cggev ( jobvl, jobvr, n, a, lda, b, ldb, alpha, beta,
             vl, ldvl, vr, ldvr, work, lwork, rwork, info )
call zggev ( jobvl, jobvr, n, a, lda, b, ldb, alpha, beta,
             vl, ldvl, vr, ldvr, work, lwork, rwork, info )
```

### Discussion

This routine computes for a pair of *n*-by-*n* real/complex nonsymmetric matrices (*A,B*), the generalized eigenvalues, and optionally, the left and/or right generalized eigenvectors.

A generalized eigenvalue for a pair of matrices (*A,B*) is a scalar  $\lambda$  or a ratio  $\alpha / \beta = \lambda$ , such that  $A - \lambda * B$  is singular. It is usually represented as the pair ( $\alpha, \beta$ ), as there is a reasonable interpretation for  $\beta=0$  and even for both being zero.

The right generalized eigenvector  $v(j)$  corresponding to the generalized eigenvalue  $\lambda(j)$  of (*A,B*) satisfies

$$A * v(j) = \lambda(j) * B * v(j).$$

The left generalized eigenvector  $u(j)$  corresponding to the generalized eigenvalue  $\lambda(j)$  of (*A,B*) satisfies

$$u(j)^H * A = \lambda(j) * u(j)^H * B$$

where  $u(j)^H$  denotes the conjugate transpose of  $u(j)$ .

**Input Parameters**

<i>jobvl</i>	<b>CHARACTER*1.</b> Must be ' <b>N</b> ' or ' <b>V</b> '. If <i>jobvl</i> = ' <b>N</b> ', the left generalized eigenvectors are not computed; If <i>jobvl</i> = ' <b>V</b> ', the left generalized eigenvectors are computed.
<i>jobvr</i>	<b>CHARACTER*1.</b> Must be ' <b>N</b> ' or ' <b>V</b> '. If <i>jobvr</i> = ' <b>N</b> ', the right generalized eigenvectors are not computed; If <i>jobvr</i> = ' <b>V</b> ', the right generalized eigenvectors are computed.
<i>n</i>	<b>INTEGER.</b> The order of the matrices <i>A</i> , <i>B</i> , <i>vl</i> , and <i>vr</i> ( <i>n</i> ≥ 0).
<i>a</i> , <i>b</i> , <i>work</i>	<b>REAL</b> for <b>sggev</b> <b>DOUBLE PRECISION</b> for <b>dggev</b> <b>COMPLEX</b> for <b>cggev</b> <b>DOUBLE COMPLEX</b> for <b>zggev</b> . Arrays: <i>a( lda, * )</i> is an array containing the <i>n</i> -by- <i>n</i> matrix <i>A</i> (first of the pair of matrices). The second dimension of <i>a</i> must be at least $\max(1, n)$ . <i>b( ldb, * )</i> is an array containing the <i>n</i> -by- <i>n</i> matrix <i>B</i> (second of the pair of matrices). The second dimension of <i>b</i> must be at least $\max(1, n)$ . <i>work( lwork )</i> is a workspace array.
<i>lda</i>	<b>INTEGER.</b> The first dimension of the array <i>a</i> . Must be at least $\max(1, n)$ .
<i>ldb</i>	<b>INTEGER.</b> The first dimension of the array <i>b</i> . Must be at least $\max(1, n)$ .
<i>ldvl</i> , <i>ldvr</i>	<b>INTEGER.</b> The first dimensions of the output matrices <i>vl</i> and <i>vr</i> , respectively. Constraints: <i>ldvl</i> ≥ 1. If <i>jobvl</i> = ' <b>V</b> ', <i>ldvl</i> ≥ $\max(1, n)$ . <i>ldvr</i> ≥ 1. If <i>jobvr</i> = ' <b>V</b> ', <i>ldvr</i> ≥ $\max(1, n)$ .
<i>lwork</i>	<b>INTEGER.</b> The dimension of the array <i>work</i> .

*lwork*  $\geq \max(1, 8n+16)$  for real flavors;  
*lwork*  $\geq \max(1, 2n)$  for complex flavors.  
 For good performance, *lwork* must generally be larger.

*rwork*      REAL for *cggev*  
                   DOUBLE PRECISION for *zggev*  
 Workspace array, DIMENSION at least  $\max(1, 8n)$ .  
 This array is used in complex flavors only.

## Output Parameters

*a, b*      On exit, these arrays have been overwritten.

*alphar, alphai*      REAL for *sggev*;  
                   DOUBLE PRECISION for *dggev*.  
 Arrays, DIMENSION at least  $\max(1, n)$  each. Contain values that form generalized eigenvalues in real flavors.  
 See *beta*.

*alpha*      COMPLEX for *cggev*;  
                   DOUBLE COMPLEX for *zggev*.  
 Array, DIMENSION at least  $\max(1, n)$ . Contain values that form generalized eigenvalues in complex flavors.  
 See *beta*.

*beta*      REAL for *sggev*  
                   DOUBLE PRECISION for *dggev*  
                   COMPLEX for *cggev*  
                   DOUBLE COMPLEX for *zggev*.  
 Array, DIMENSION at least  $\max(1, n)$ .  
*For real flavors:*  
 On exit,  $(\text{alphar}(j) + \text{alphai}(j)*i)/\text{beta}(j)$ ,  $j=1, \dots, n$ , will be the generalized eigenvalues.  
 If *alphai(j)* is zero, then the j-th eigenvalue is real; if positive, then the j-th and (j+1)-st eigenvalues are a complex conjugate pair, with *alphai(j+1)* negative.  
*For complex flavors:*  
 On exit,  $\text{alpha}(j)/\text{beta}(j)$ ,  $j=1, \dots, n$ , will be the generalized eigenvalues.

See also *Application Notes* below.

$vl, vr$ 

REAL for sggev

DOUBLE PRECISION for dggev

COMPLEX for cggev

DOUBLE COMPLEX for zggev.

Arrays:

 $vl(ldvl, *)$ ; the second dimension of  $vl$  must be at least  $\max(1, n)$ .

If  $jobvl = 'V'$ , the left generalized eigenvectors  $u(j)$  are stored one after another in the columns of  $vl$ , in the same order as their eigenvalues. Each eigenvector will be scaled so the largest component have  $\text{abs}(\text{Re}) + \text{abs}(\text{Im}) = 1$ . If  $jobvl = 'N'$ ,  $vl$  is not referenced.

*For real flavors:*

If the  $j$ -th eigenvalue is real, then  $u(j) = vl(:,j)$ , the  $j$ -th column of  $vl$ . If the  $j$ -th and  $(j+1)$ -st eigenvalues form a complex conjugate pair, then  $u(j) = vl(:,j) + i * vl(:,j+1)$  and  $u(j+1) = vl(:,j) - i * vl(:,j+1)$ , where  $i = \sqrt{-1}$ .

*For complex flavors:* $u(j) = vl(:,j)$ , the  $j$ -th column of  $vl$ . $vr(ldvr, *)$ ; the second dimension of  $vr$  must be at least  $\max(1, n)$ .

If  $jobvr = 'V'$ , the right generalized eigenvectors  $v(j)$  are stored one after another in the columns of  $vr$ , in the same order as their eigenvalues. Each eigenvector will be scaled so the largest component have  $\text{abs}(\text{Re}) + \text{abs}(\text{Im}) = 1$ . If  $jobvr = 'N'$ ,  $vr$  is not referenced.

*For real flavors:*

If the  $j$ -th eigenvalue is real, then  $v(j) = vr(:,j)$ , the  $j$ -th column of  $vr$ . If the  $j$ -th and  $(j+1)$ -st eigenvalues form a complex conjugate pair, then  $v(j) = vr(:,j) + i * vr(:,j+1)$  and  $v(j+1) = vr(:,j) - i * vr(:,j+1)$ .

*For complex flavors:* $v(j) = vr(:,j)$ , the  $j$ -th column of  $vr$ . $work(1)$ 

On exit, if  $info = 0$ , then  $work(1)$  returns the required minimal size of  $lwork$ .

*info*

INTEGER.

If *info* = 0, the execution is successful.If *info* = *-i*, the *i*th parameter had an illegal value.If *info* = *i*, and*i* ≤ *n* :

the *QZ* iteration failed. No eigenvectors have been calculated, but *alphar(j)*, *alphai(j)* (for real flavors), or *alpha(j)* (for complex flavors), and *beta(j)*,  
*j*=*info*+1,...,*n* should be correct.

*i* > *n* : errors that usually indicate LAPACK problems:*i* = *n*+1: other than *QZ* iteration failed in ?*hgeqz*;*i* = *n*+2: error return from ?*tgevc*.

### Application Notes

If you are in doubt how much workspace to supply for the array *work*, use a generous value of *lwork* for the first run. On exit, examine *work(1)* and use this value for subsequent runs.

The quotients *alphar(j)/beta(j)* and *alphai(j)/beta(j)* may easily over- or underflow, and *beta(j)* may even be zero. Thus, you should avoid simply computing the ratio. However, *alphar* and *alphai* (for real flavors) or *alpha* (for complex flavors) will be always less than and usually comparable with *norm(A)* in magnitude, and *beta* always less than and usually comparable with *norm(B)*.

---

## ?ggevx

*Computes the generalized eigenvalues,  
and, optionally, the left and/or right  
generalized eigenvectors.*

---

```
call sggevx ( balanc, jobvl, jobvr, sense, n, a, lda, b, ldb,
              alphar, alphai, beta, vl, ldvl, vr, ldvr, ilo, ihi,
              lscale, rscale, abnrm, bbnrm, rconde, rcondv, work,
              lwork, iwork, bwork, info)
call dggevx ( balanc, jobvl, jobvr, sense, n, a, lda, b, ldb,
              alphar, alphai, beta, vl, ldvl, vr, ldvr, ilo, ihi,
              lscale, rscale, abnrm, bbnrm, rconde, rcondv, work,
              lwork, iwork, bwork, info)
call cggevx ( balanc, jobvl, jobvr, sense, n, a, lda, b, ldb,
              alpha, beta, vl, ldvl, vr, ldvr, ilo, ihi,
              lscale, rscale, abnrm, bbnrm, rconde, rcondv, work,
              lwork, rwork, iwork, bwork, info)
call zggevx ( balanc, jobvl, jobvr, sense, n, a, lda, b, ldb,
              alpha, beta, vl, ldvl, vr, ldvr, ilo, ihi,
              lscale, rscale, abnrm, bbnrm, rconde, rcondv, work,
              lwork, rwork, iwork, bwork, info)
```

### Discussion

This routine computes for a pair of *n*-by-*n* real/complex nonsymmetric matrices (*A,B*), the generalized eigenvalues, and optionally, the left and/or right generalized eigenvectors.

Optionally also, it computes a balancing transformation to improve the conditioning of the eigenvalues and eigenvectors (*ilo*, *ihi*, *lscale*, *rscale*, *abnrm*, and *bbnrm*), reciprocal condition numbers for the eigenvalues (*rconde*), and reciprocal condition numbers for the right eigenvectors (*rcondv*).

A generalized eigenvalue for a pair of matrices (*A,B*) is a scalar  $\lambda$  or a ratio *alpha / beta* =  $\lambda$ , such that  $A - \lambda^*B$  is singular. It is usually represented as the pair (*alpha*, *beta*), as there is a reasonable interpretation for *beta*=0 and

even for both being zero.

The right generalized eigenvector  $v(j)$  corresponding to the generalized eigenvalue  $\lambda(j)$  of  $(A,B)$  satisfies

$$A^*v(j) = \lambda(j)^*B^*v(j).$$

The left generalized eigenvector  $u(j)$  corresponding to the generalized eigenvalue  $\lambda(j)$  of  $(A,B)$  satisfies

$$u(j)^H A = \lambda(j)^* u(j)^H B$$

where  $u(j)^H$  denotes the conjugate transpose of  $u(j)$ .

### Input Parameters

<i>balanc</i>	CHARACTER*1. Must be ' <b>N</b> ', ' <b>P</b> ', ' <b>S</b> ', or ' <b>B</b> '. Specifies the balance option to be performed.  If <i>balanc</i> = ' <b>N</b> ', do not diagonally scale or permute; If <i>balanc</i> = ' <b>P</b> ', permute only; If <i>balanc</i> = ' <b>S</b> ', scale only; If <i>balanc</i> = ' <b>B</b> ', both permute and scale.  Computed reciprocal condition numbers will be for the matrices after balancing and/or permuting. Permuting does not change condition numbers (in exact arithmetic), but balancing does.
<i>jobvl</i>	CHARACTER*1. Must be ' <b>N</b> ' or ' <b>V</b> '. If <i>jobvl</i> = ' <b>N</b> ', the left generalized eigenvectors are not computed; If <i>jobvl</i> = ' <b>V</b> ', the left generalized eigenvectors are computed.
<i>jobvr</i>	CHARACTER*1. Must be ' <b>N</b> ' or ' <b>V</b> '. If <i>jobvr</i> = ' <b>N</b> ', the right generalized eigenvectors are not computed; If <i>jobvr</i> = ' <b>V</b> ', the right generalized eigenvectors are computed.
<i>sense</i>	CHARACTER*1. Must be ' <b>N</b> ', ' <b>E</b> ', ' <b>V</b> ', or ' <b>B</b> '. Determines which reciprocal condition number are computed.

If `sense = 'N'`, none are computed;  
 If `sense = 'E'`, computed for eigenvalues only;  
 If `sense = 'V'`, computed for eigenvectors only;  
 If `sense = 'B'`, computed for eigenvalues and eigenvectors.

`n` **INTEGER**. The order of the matrices  $A$ ,  $B$ ,  $v_l$ , and  $vr$  ( $n \geq 0$ ).

`a, b, work` **REAL** for `sggevx`  
**DOUBLE PRECISION** for `dggevx`  
**COMPLEX** for `cggevx`  
**DOUBLE COMPLEX** for `zggevx`.

Arrays:

`a( lda, * )` is an array containing the  $n$ -by- $n$  matrix  $A$  (first of the pair of matrices).

The second dimension of `a` must be at least  $\max(1, n)$ .

`b( ldb, * )` is an array containing the  $n$ -by- $n$  matrix  $B$  (second of the pair of matrices).

The second dimension of `b` must be at least  $\max(1, n)$ .

`work( lwork )` is a workspace array.

`lda` **INTEGER**. The first dimension of the array `a`.  
 Must be at least  $\max(1, n)$ .

`ldb` **INTEGER**. The first dimension of the array `b`.  
 Must be at least  $\max(1, n)$ .

`ldvl, ldvr` **INTEGER**. The first dimensions of the output matrices  $v_l$  and  $vr$ , respectively. Constraints:  
 $ldvl \geq 1$ . If `jobvl = 'V'`,  $ldvl \geq \max(1, n)$ .  
 $ldvr \geq 1$ . If `jobvr = 'V'`,  $ldvr \geq \max(1, n)$ .

`lwork` **INTEGER**. The dimension of the array `work`.  
*For real flavors:*  
 $lwork \geq \max(1, 6n)$ ;  
 if `sense = 'E'`,  $lwork \geq 12n$ ;  
 if `sense = 'V'`, or `'B'`,  $lwork \geq 2n^2 + 12n + 16$ .  
*For complex flavors:*

---

$lwork \geq \max(1, 2n);$   
 if  $sense = 'N'$ , or ' $E$ ',  $lwork \geq 2n$ ;  
 if  $sense = 'V'$ , or ' $B$ ',  $lwork \geq 2n^2 + 2n$ .  
  
 $rwork$       REAL for `cggevx`  
                 DOUBLE PRECISION for `zggevx`  
 Workspace array, DIMENSION at least  $\max(1, 6n)$ .  
 This array is used in complex flavors only.  
  
 $iwork$       INTEGER.  
 Workspace array, DIMENSION at least  $(n+6)$  for real  
 flavors and at least  $(n+2)$  for complex flavors.  
 Not referenced if  $sense = 'E'$ .  
  
 $bwork$       LOGICAL.  
 Workspace array, DIMENSION at least  $\max(1, n)$ .  
 Not referenced if  $sense = 'N'$ .

## Output Parameters

$a, b$       On exit, these arrays have been overwritten.  
 If  $jobvl = 'V'$  or  $jobvr = 'V'$  or both, then  $a$  contains  
 the first part of the real Schur form of the "balanced"  
 versions of the input  $A$  and  $B$ , and  $b$  contains its second  
 part.  
  
 $alphar, alphai$       REAL for `sggevx`;  
                 DOUBLE PRECISION for `dggevx`.  
 Arrays, DIMENSION at least  $\max(1, n)$  each. Contain  
 values that form generalized eigenvalues in real flavors.  
 See  $\beta$ .  
  
 $alpha$       COMPLEX for `cggevx`;  
                 DOUBLE COMPLEX for `zggevx`.  
 Array, DIMENSION at least  $\max(1, n)$ . Contain values  
 that form generalized eigenvalues in complex flavors.  
 See  $\beta$ .  
  
 $\beta$       REAL for `sggevx`  
                 DOUBLE PRECISION for `dggevx`  
                 COMPLEX for `cggevx`  
                 DOUBLE COMPLEX for `zggevx`.

Array, **DIMENSION** at least  $\max(1, n)$ .

*For real flavors:*

On exit,  $(\text{alphar}(j) + \text{alpha}(j)*i)/\text{beta}(j)$ ,  $j=1, \dots, n$ , will be the generalized eigenvalues.

If  $\text{alpha}(j)$  is zero, then the  $j$ -th eigenvalue is real; if positive, then the  $j$ -th and  $(j+1)$ -st eigenvalues are a complex conjugate pair, with  $\text{alpha}(j+1)$  negative.

*For complex flavors:*

On exit,  $\text{alpha}(j)/\text{beta}(j)$ ,  $j=1, \dots, n$ , will be the generalized eigenvalues.

See also *Application Notes* below.

**vl, vr**

**REAL** for **sggevx**

**DOUBLE PRECISION** for **dggevx**

**COMPLEX** for **cggevx**

**DOUBLE COMPLEX** for **zggevx**.

Arrays:

**vl** (*ldvl, \**); the second dimension of **vl** must be at least  $\max(1, n)$ .

If  $\text{jobvl} = 'V'$ , the left generalized eigenvectors  $u(j)$  are stored one after another in the columns of **vl**, in the same order as their eigenvalues. Each eigenvector will be scaled so the largest component have  $\text{abs}(\text{Re}) + \text{abs}(\text{Im}) = 1$ . If  $\text{jobvl} = 'N'$ , **vl** is not referenced.

*For real flavors:*

If the  $j$ -th eigenvalue is real, then  $u(j) = \text{vl}(:, j)$ , the  $j$ -th column of **vl**. If the  $j$ -th and  $(j+1)$ -st eigenvalues form a complex conjugate pair, then  $u(j) = \text{vl}(:, j) + i^* \text{vl}(:, j+1)$  and  $u(j+1) = \text{vl}(:, j) - i^* \text{vl}(:, j+1)$ , where  $i = \sqrt{-1}$ .

*For complex flavors:*

$u(j) = \text{vl}(:, j)$ , the  $j$ -th column of **vl**.

**vr** (*ldvr, \**); the second dimension of **vr** must be at least  $\max(1, n)$ .

If  $\text{jobvr} = 'V'$ , the right generalized eigenvectors  $v(j)$  are stored one after another in the columns of **vr**, in the same order as their eigenvalues. Each eigenvector will

be scaled so the largest component have  $\text{abs}(\text{Re}) + \text{abs}(\text{Im}) = 1$ . If  $\text{jobvr} = 'N'$ ,  $\text{vr}$  is not referenced.

*For real flavors:*

If the j-th eigenvalue is real, then  $v(j) = \text{vr}(:,j)$ , the j-th column of  $\text{vr}$ . If the j-th and (j+1)-st eigenvalues form a complex conjugate pair, then  $v(j) = \text{vr}(:,j) + i^* \text{vr}(:,j+1)$  and  $v(j+1) = \text{vr}(:,j) - i^* \text{vr}(:,j+1)$ .

*For complex flavors:*

$v(j) = \text{vr}(:,j)$ , the j-th column of  $\text{vr}$ .

*ilo, ihi*

**INTEGER**.

*ilo* and *ihi* are integer values such that on exit

$A(i,j) = 0$  and  $B(i,j) = 0$  if  $i > j$  and  $j = 1, \dots, \text{ilo}-1$  or  
 $i = \text{ihi}+1, \dots, n$ .

If  $\text{balanc} = 'N'$  or ' $S$ ', *ilo* = 1 and *ihi* = *n*.

*lscale, rscale*

**REAL** for single-precision flavors

**DOUBLE PRECISION** for double-precision flavors.

Arrays, **DIMENSION** at least  $\max(1, n)$  each.

*lscale* contains details of the permutations and scaling factors applied to the left side of *A* and *B*.

If  $PL(j)$  is the index of the row interchanged with row *j*, and  $DL(j)$  is the scaling factor applied to row *j*, then

$$\text{lscale}(j) = PL(j), \quad \text{for } j = 1, \dots, \text{ilo}-1$$

$$= DL(j), \quad \text{for } j = \text{ilo}, \dots, \text{ihi}$$

$$= PL(j) \quad \text{for } j = \text{ihi}+1, \dots, n.$$

The order in which the interchanges are made is *n* to *ihi*+1, then 1 to *ilo*-1.

*rscale* contains details of the permutations and scaling factors applied to the right side of *A* and *B*.

If  $PR(j)$  is the index of the column interchanged with column *j*, and  $DR(j)$  is the scaling factor applied to column *j*, then

$$\text{rscale}(j) = PR(j), \quad \text{for } j = 1, \dots, \text{ilo}-1$$

$$= DR(j), \quad \text{for } j = \text{ilo}, \dots, \text{ihi}$$

$$= PR(j) \quad \text{for } j = \text{ihi}+1, \dots, n.$$

The order in which the interchanges are made is  $n$  to  $ihi+1$ , then 1 to  $ilo-1$ .

*abnrm, bbnrm*

**REAL** for single-precision flavors

**DOUBLE PRECISION** for double-precision flavors.

The one-norms of the balanced matrices  $A$  and  $B$ , respectively.

*rconde, rcondv*

**REAL** for single precision flavors

**DOUBLE PRECISION** for double precision flavors.

Arrays, **DIMENSION** at least  $\max(1, n)$  each.

If  $sense = 'E'$ , or ' $B$ ', *rconde* contains the reciprocal condition numbers of the selected eigenvalues, stored in consecutive elements of the array. For a complex conjugate pair of eigenvalues two consecutive elements of *rconde* are set to the same value. Thus *rconde(j)*, *rcondv(j)*, and the  $j$ -th columns of *vl* and *vr* all correspond to the same eigenpair (but not in general the  $j$ -th eigenpair, unless all eigenpairs are selected).

If  $sense = 'V'$ , *rconde* is not referenced.

If  $sense = 'V'$ , or ' $B$ ', *rcondv* contains the estimated reciprocal condition numbers of the selected eigenvectors, stored in consecutive elements of the array. For a complex eigenvector two consecutive elements of *rcondv* are set to the same value. If the eigenvalues cannot be reordered to compute *rcondv(j)*, *rcondv(j)* is set to 0; this can only occur when the true value would be very small anyway.

If  $sense = 'E'$ , *rcondv* is not referenced.

*work(1)*

On exit, if  $info = 0$ , then *work(1)* returns the required minimal size of *lwork*.

*info*

**INTEGER**.

If  $info = 0$ , the execution is successful.

If  $info = -i$ , the  $i$ th parameter had an illegal value.

If  $info = i$ , and

$i \leq n$  :

---

the  $QZ$  iteration failed. No eigenvectors have been calculated, but `alphar(j)`, `alphai(j)` (for real flavors), or `alpha(j)` (for complex flavors), and `beta(j)`,  
 $j=\text{info}+1,\dots,n$  should be correct.

$i > n$  : errors that usually indicate LAPACK problems:

$i = n+1$ : other than  $QZ$  iteration failed in `?hgeqz`;

$i = n+2$ : error return from `?tgevc`.

### Application Notes

If you are in doubt how much workspace to supply for the array `work`, use a generous value of `lwork` for the first run. On exit, examine `work(1)` and use this value for subsequent runs.

The quotients `alphar(j)/beta(j)` and `alphai(j)/beta(j)` may easily over- or underflow, and `beta(j)` may even be zero. Thus, you should avoid simply computing the ratio. However, `alphar` and `alphai` (for real flavors) or `alpha` (for complex flavors) will be always less than and usually comparable with `norm(A)` in magnitude, and `beta` always less than and usually comparable with `norm(B)`.

## References

- |           |   |
|-----------|---|
| [LUG]     | E. Anderson, Z. Bai et al. <i>LAPACK User's Guide</i> . Third edition, SIAM, Philadelphia, 1999.                    |
| [Golub96] | G. Golub, C. Van Loan. <i>Matrix Computations</i> . Johns Hopkins University Press, Baltimore, third edition, 1996. |

# *Vector Mathematical Functions*

6

---

This chapter describes Vector Mathematical Functions Library (VML), which is designed to compute elementary functions on vector arguments. VML is an integral part of the Math Kernel Library and the VML terminology is used here for simplicity in discussing this group of functions.

VML includes a set of highly optimized implementations of certain computationally expensive core mathematical functions (power, trigonometric, exponential, hyperbolic etc.) that operate on vectors.

Application programs that might significantly improve performance with VML include nonlinear programming software, integrals computation, and many others.

VML functions are divided into the following groups according to the operations they perform:

- [VML Mathematical Functions](#) compute values of elementary functions (such as sine, cosine, exponential, logarithm and so on) on vectors with unit increment indexing.
- [VML Pack/Unpack Functions](#) convert to and from vectors with positive increment indexing, vector indexing and mask indexing (see [Appendix A](#) for details on vector indexing methods).
- [VML Service Functions](#) allow the user to set /get the accuracy mode, and set/get the error code.

VML mathematical functions take an input vector as argument, compute values of the respective elementary function element-wise, and return the results in an output vector.

## Data Types and Accuracy Modes

Mathematical and pack/unpack vector functions in VML have been implemented for vector arguments of single and double precision real data. Both Fortran- and C-interfaces to all functions, including VML service functions, are provided in the library. The differences in naming and calling the functions for Fortran- and C-interfaces are detailed in the [Function Naming Conventions](#) section below.

Each vector function from VML (for each data format) can work in two modes: High Accuracy (HA) and Low Accuracy (LA). For many functions, using the LA version will improve performance at the cost of accuracy. For some cases, the advantage of relaxing the accuracy improves performance very little so the same function is employed for both versions. Error behavior depends not only on whether the HA or LA version is chosen, but also depends on the processor on which the software runs. In addition, special value behavior may differ between the HA and LA versions of the functions. Any information on accuracy behavior can be found in the Release Notes for MKL.

Switching between the two modes (HA and LA) is accomplished by using `vmlSetMode(mode)` (see [Table 6-10](#)). The function `vmlGetMode()` will return the currently used mode. The High Accuracy mode is used by default.

## Function Naming Conventions

Full names of all VML functions include only lowercase letters for Fortran-interface, whereas for C-interface names the lowercase letters are mixed with uppercase.

VML mathematical and pack/unpack function full names have the following structure:

`v <p> <name> <mod>`

The initial letter `v` is a prefix indicating that a function belongs to VML. The `<p>` field is a precision prefix that indicates the data type:

- `s`    `REAL` for Fortran-interface, or `float` for C-interface
- `d`    `DOUBLE PRECISION` for Fortran-interface, or `double` for C-interface.

The `<name>` field indicates the function short name, with some of its letters in uppercase for C-interface (see [Tables 6-2, 6-8](#)).

The `<mod>` field (written in uppercase for C-interface) is present in pack/unpack functions only; it indicates the indexing method used:

- `i` indexing with positive increment
- `v` indexing with index vector
- `m` indexing with mask vector.

VML service function full names have the following structure:

`vml <name>`

where `vml` is a prefix indicating that a function belongs to VML, and `<name>` is the function short name, which includes some uppercase letters for C-interface (see [Table 6-9](#)).

To call VML functions from an application program, use conventional function calls. For example, the VML exponential function for single precision data can be called as

`call vsexp ( n, a, y )` for Fortran-interface, or  
`vsExp ( n, a, y );` for C-interface.

## Functions Interface

The interface to VML functions includes function full names and the arguments list.

The Fortran- and C-interface descriptions for different groups of VML functions are given below. Note that some functions (`Div`, `Pow`, and `Atan2`) have two input vectors `a` and `b` as their arguments, while `SinCos` function has two output vectors `y` and `z`.

### VML Mathematical Functions:

Fortran:

```
call v<p><name>( n, a, y )
call v<p><name>( n, a, b, y )
call v<p><name>( n, a, y, z )
```

C:

```
v<p><name>( n, a, y );
v<p><name>( n, a, b, y );
v<p><name>( n, a, y, z );
```

### Pack Functions:

Fortran:

```
call v<p>packi( n, a, inca, y )
call v<p>packv( n, a, ia, y )
call v<p>packm( n, a, ma, y )
```

C:

```
v<p>PackI( n, a, inca, y );
v<p>PackV( n, a, ia, y );
v<p>PackM( n, a, ma, y );
```

### Unpack Functions:

Fortran:

```
call v<p>unpacki( n, a, y, incy )
call v<p>unpackv( n, a, y, iy )
call v<p>unpackm( n, a, y, my )
```

C:

```
v<p>UnpackI( n, a, y, incy );
v<p>UnpackV( n, a, y, iy );
v<p>UnpackM( n, a, y, my );
```

### Service Functions:

Fortran:

```
oldmode = vmlsetmode( mode )
mode    = vmlgetmode( )
olderr  = vmlseterrstatus( err )
err     = vmlgeterrstatus( )
olderr  = vmlclearerrstatus( )
oldcallback = vmlseterrorcallback( callback )
callback = vmlgeterrorcallback( )
oldcallback = vmlclearerrorcallback( )
```

C:

```

oldmode = vmlSetMode( mode );
mode    = vmlGetMode( void );
olderr  = vmlSetErrStatus( err );
err     = vmlGetErrStatus(void);
olderr  = vmlClearErrStatus(void);
oldcallback = vmlsetErrorCallBack(callback );
callback = vmlGetErrorCallBack( void );
oldcallback = vmlClearErrorCallBack(void );

```

#### **Input Parameters:**

<i>n</i>	number of elements to be calculated
<i>a</i>	first input vector
<i>b</i>	second input vector
<i>inca</i>	vector increment for the input vector <i>a</i>
<i>ia</i>	index vector for the input vector <i>a</i>
<i>ma</i>	mask vector for the input vector <i>a</i>
<i>incy</i>	vector increment for the output vector <i>y</i>
<i>iy</i>	index vector for the output vector <i>y</i>
<i>my</i>	mask vector for the output vector <i>y</i>
<i>err</i>	error code
<i>mode</i>	VML mode
<i>callback</i>	pointer to the callback function

#### **Output Parameters:**

<i>y</i>	first output vector
<i>z</i>	second output vector
<i>err</i>	error code
<i>mode</i>	VML mode
<i>olderr</i>	former error code
<i>oldmode</i>	former VML mode
<i>oldcallback</i>	pointer to the former callback function

The data types of the parameters used in each function are specified in the respective function description section. All VML mathematical functions can perform in-place operations, which means that the same vector can be used as both input and output parameter. This holds true for functions with two input vectors as well, in which case one of them may be overwritten with the output vector. For functions with two output vectors, one of them may coincide with the input vector.

## Vector Indexing Methods

Current VML mathematical functions work only with unit increment. Arrays with other increments, or more complicated indexing, can be accommodated by gathering the elements into a contiguous vector and then scattering them after the computation is complete.

Three following indexing methods are used to gather/scatter the vector elements in VML:

- positive increment
- index vector
- mask vector.

The indexing method used in a particular function is indicated by the indexing modifier (see the description of the `<mod>` field in [Function Naming Conventions](#)). For more information on indexing methods see [Vector Arguments in VML](#) in Appendix A.

## Error Diagnostics

The VML library has its own error handler. The only difference for C- and Fortran- interfaces is that the MKL error reporting routine `XERBLA` can be called after the Fortran- interface VML function encounters an error, and this routine gets information on `VML_STATUS_BADSIZE` and `VML_STATUS_BADMEM` input errors (see [Table 6-12](#)).

The VML error handler has the following properties:

- 1) The Error Status (`vmlErrStatus`) global variable is set after each VML function call. The possible values of this variable are shown in the [Table 6-12](#).
- 2) Depending on the VML mode, the error handler function invokes:
  - `errno` variable setting. The possible values are shown in the [Table 6-1](#)
  - writing error text information to the `stderr` stream
  - raising the appropriate exception on error, if necessary
  - calling the additional error handler callback function.

**Table 6-1 Set Values of the `errno` Variable**

Value of <code>errno</code>	Description
<code>0</code>	No errors are detected.
<code>EINVAL</code>	The array dimension is not positive.
<code>EACCES</code>	NULL pointer is passed.
<code>EDOM</code>	At least one of array values is out of a range of definition.
<code>ERANGE</code>	At least one of array values caused a singularity, overflow or underflow.

## VML Mathematical Functions

This section describes VML functions which compute values of elementary mathematical functions on real vector arguments with unit increment.

Each function group is introduced by its short name, a brief description of its purpose, and the calling sequence for each type of data both for Fortran- and C-interfaces, as well as a description of the input/output arguments.

For all VML mathematical functions, the input range of parameters is equal to the mathematical range of definition in the set of defined values for the respective data type. Several VML functions, specifically `Div`, `Exp`, `Sinh`, `Cosh`, and `Pow`, can result in an overflow. For these functions, the respective input threshold values that mark off the precision overflow are specified in the function description section. Note that in these specifications, `FLOAT_MAX` denotes the maximum number representable in single precision data type, while `DBL_MAX` denotes the maximum number representable in double precision data type.

[Table 6-2](#) lists available mathematical functions and data types associated with them.

**Table 6-2      VML Mathematical Functions**

Function Short Name	Data Types	Description
<b>Power and Root Functions</b>		
<a href="#"><u>Inv</u></a>	s, d	Inversion of the vector elements
<a href="#"><u>Div</u></a>	s, d	Divide elements of one vector by elements of second vector
<a href="#"><u>Sqrt</u></a>	s, d	Square root of vector elements
<a href="#"><u>InvSqrt</u></a>	s, d	Inverse square root of vector elements
<a href="#"><u>Cbrt</u></a>	s, d	Cube root of vector elements
<a href="#"><u>InvCbrt</u></a>	s, d	Inverse cube root of vector elements
<a href="#"><u>Pow</u></a>	s, d	Each vector element raised to the specified power

\* continued

**Table 6-2 VML Mathematical Functions (continued)**

Function Name	Short Types	Description
<b>Exponential and Logarithmic Functions</b>		
<u>Exp</u>	s, d	Exponential of vector elements
<u>Ln</u>	s, d	Natural logarithm of vector elements
<u>Log10</u>	s, d	Denary logarithm of vector elements
<b>Trigonometric Functions</b>		
<u>Cos</u>	s, d	Cosine of vector elements
<u>Sin</u>	s, d	Sine of vector elements
<u>SinCos</u>	s, d	Sine and cosine of vector elements
<u>Tan</u>	s, d	Tangent of vector elements
<u>Acos</u>	s, d	Inverse cosine of vector elements
<u>Asin</u>	s, d	Inverse sine of vector elements
<u>Atan</u>	s, d	Inverse tangent of vector elements
<u>Atan2</u>	s, d	Four-quadrant inverse tangent of elements of two vectors
<b>Hyperbolic Functions</b>		
<u>Cosh</u>	s, d	Hyperbolic cosine of vector elements
<u>Sinh</u>	s, d	Hyperbolic sine of vector elements
<u>Tanh</u>	s, d	Hyperbolic tangent of vector elements
<u>Acosh</u>	s, d	Inverse hyperbolic cosine (nonnegative) of vector elements
<u>Asinh</u>	s, d	Inverse hyperbolic sine of vector elements
<u>Atanh</u>	s, d	Inverse hyperbolic tangent of vector elements

---

## Inv

*Performs element by element inversion  
of the vector.*

---

### Fortran:

```
call vsinv( n, a, y )
call vdinv( n, a, y )
```

### C:

```
vsInv( n, a, Y );
vdInv( n, a, Y );
```

## Input Parameters

Fortran:

*n*            INTEGER, INTENT(IN). Specifies the number of elements to be calculated.  
*a*            REAL, INTENT(IN) for **vsinv**  
                DOUBLE PRECISION, INTENT(IN) for **vdinv**  
                Array, specifies the input vector *a*.

C:

*n*            int. Specifies the number of elements to be calculated.  
*a*            const float\* for **vsInv**  
                const double\* for **vdInv**  
                Pointer to an array that contains the input vector *a*.

## Output Parameters

Fortran:

*y*            REAL for **vsinv**  
                DOUBLE PRECISION for **vdinv**  
                Array, specifies the output vector *y*.

C:

**y**      `float*` for `vsInv`  
`double*` for `vdInv`

Pointer to an array that contains the output vector **y**.

## Div

*Performs element by element division of vector **a** by vector **b***

---

**Fortran:**

```
call vsdiv( n, a, b, y )
call vddiv( n, a, b, y )
```

**C:**

```
vsDiv( n, a, b, y );
vdDiv( n, a, b, y );
```

### Input Parameters

Fortran:

**n**      `INTEGER, INTENT(IN)`. Specifies the number of elements to be calculated.

**a, b**      `REAL, INTENT(IN)` for `vsdiv`  
`DOUBLE PRECISION, INTENT(IN)` for `vddiv`  
 Arrays, specify the input vectors **a** and **b**.

C:

**n**      `int`. Specifies the number of elements to be calculated.

**a, b**      `const float*` for `vsDiv`  
`const double*` for `vdDiv`

Pointers to arrays that contain the input vectors **a** and **b**.

**Table 6-3 Precision Overflow Thresholds for `Div` Function**

Data Type	Threshold Limitations on Input Parameters
single precision	$\text{abs}(\textcolor{red}{a}[i]) < \text{abs}(\textcolor{red}{b}[i]) * \text{FLT\_MAX}$
double precision	$\text{abs}(\textcolor{red}{a}[i]) < \text{abs}(\textcolor{red}{b}[i]) * \text{DBL\_MAX}$

**Output Parameters**

Fortran:

`Y`      `REAL`    for `vsdiv`  
               `DOUBLE PRECISION` for `vddiv`  
               Array, specifies the output vector `Y`.

C:

`Y`      `float*`    for `vsDiv`  
               `double*` for `vdDiv`  
               Pointer to an array that contains the output vector `Y`.

**Sqrt**

*Computes a square root  
of vector elements.*

**Fortran:**

```
call vssqrt( n, a, y )
call vdsqrt( n, a, y )
```

**C:**

```
vsSqrt( n, a, y );
vdSqrt( n, a, y );
```

## Input Parameters

Fortran:

*n*            INTEGER, INTENT(IN). Specifies the number of elements to be calculated.  
*a*            REAL, INTENT(IN) for vssqrt  
                DOUBLE PRECISION, INTENT(IN) for vdsqrt  
                Array, specifies the input vector *a*.

C:

*n*            int. Specifies the number of elements to be calculated.  
*a*            const float\* for vsqrt  
                const double\* for vdsqrt  
                Pointer to an array that contains the input vector *a*.

## Output Parameters

Fortran:

*y*            REAL for vssqrt  
                DOUBLE PRECISION for vdsqrt  
                Array, specifies the output vector *y*.

C:

*y*            float\* for vsqrt  
                double\* for vdsqrt  
                Pointer to an array that contains the output vector *y*.

---

## InvSqrt

*Computes an inverse square root  
of vector elements.*

---

### Fortran:

```
call vsinvsqrt( n, a, y )
call vdinvsqrt( n, a, Y )
```

**C:**

```
vsInvSqrt( n, a, y );
vdInvSqrt( n, a, y );
```

**Input Parameters**

Fortran:

*n*            INTEGER, INTENT(IN). Specifies the number of elements to be calculated.

*a*            REAL, INTENT(IN) for `vsinvsqrt`  
                DOUBLE PRECISION, INTENT(IN) for `vdinvsqrt`  
                Array, specifies the input vector *a*.

**C:**

*n*            int. Specifies the number of elements to be calculated.

*a*            const float\* for `vsInvSqrt`  
                const double\* for `vdInvSqrt`  
                Pointer to an array that contains the input vector *a*.

**Output Parameters**

Fortran:

*y*            REAL for `vsinvsqrt`  
                DOUBLE PRECISION for `vdinvsqrt`  
                Array, specifies the output vector *y*.

**C:**

*y*            float\* for `vsInvSqrt`  
                double\* for `vdInvSqrt`  
                Pointer to an array that contains the output vector *y*.

---

## Cbrt

*Computes a cube root  
of vector elements.*

---

### Fortran:

```
call vscbrt( n, a, y )
call vdcbrt( n, a, y )
```

### C:

```
vsCbrt( n, a, y );
vdCbrt( n, a, y );
```

### Input Parameters

Fortran:

*n*            INTEGER, INTENT(IN). Specifies the number of elements to be calculated.

*a*            REAL, INTENT(IN) for vscbrt  
                DOUBLE PRECISION, INTENT(IN) for vdcbrt  
                Array, specifies the input vector *a*.

C:

*n*            int. Specifies the number of elements to be calculated.

*a*            const float\* for vsCbrt  
                const double\* for vdCbrt  
                Pointer to an array that contains the input vector *a*.

### Output Parameters

Fortran:

*y*            REAL for vscbrt  
                DOUBLE PRECISION for vdcbrt  
                Array, specifies the output vector *y*.

C:

*y*            float\* for vsCbrt  
                double\* for vdCbrt  
                Pointer to an array that contains the output vector *y*.

## InvCbrt

*Computes an inverse cube root  
of vector elements.*

---

### Fortran:

```
call vsinvcbrt( n, a, y )
call vdinvcbrt( n, a, y )
```

### C:

```
vsInvCbrt( n, a, y );
vdInvCbrt( n, a, y );
```

## Input Parameters

Fortran:

**n**            INTEGER, INTENT(IN). Specifies the number of elements to be calculated.

**a**            REAL, INTENT(IN) for **vsinvcbrt**  
                DOUBLE PRECISION, INTENT(IN) for **vdinvcbrt**  
                Array, specifies the input vector **a**.

C:

**n**            int. Specifies the number of elements to be calculated.

**a**            const float\* for **vsInvCbrt**  
                const double\* for **vdInvCbrt**  
                Pointer to an array that contains the input vector **a**.

## Output Parameters

Fortran:

**y**            REAL for **vsinvcbrt**  
                DOUBLE PRECISION for **vdinvcbrt**  
                Array, specifies the output vector **y**.

C:

**y**            float\* for **vsInvCbrt**  
                double\* for **vdInvCbrt**  
                Pointer to an array that contains the output vector **y**.

## Pow

Computes  $a$  to the power  $b$   
for elements of two vectors.

### Fortran:

```
call vspow( n, a, b, y )
call vdpow( n, a, b, y )
```

### C:

```
vsPow( n, a, b, y );
vdPow( n, a, b, y );
```

### Input Parameters

Fortran:

*n*            INTEGER, INTENT(IN). Specifies the number of elements to be calculated.  
*a, b*        REAL, INTENT(IN) for vspow  
                DOUBLE PRECISION, INTENT(IN) for vdPow  
                Arrays, specify the input vectors *a* and *b*.

C:

*n*            int. Specifies the number of elements to be calculated.  
*a, b*        const float\* for vsPow  
                const double\* for vdPow  
                Pointers to arrays that contain the input vectors *a* and *b*.

**Table 6-4      Precision Overflow Thresholds for Pow Function**

Data Type	Threshold Limitations on Input Parameters
single precision	$\text{abs}(a[i]) < (\text{FLT\_MAX})^{1/b[i]}$
double precision	$\text{abs}(a[i]) < (\text{DBL\_MAX})^{1/b[i]}$

## Output Parameters

Fortran:

`y`            `REAL`    for `vspow`  
                 `DOUBLE PRECISION` for `vdpow`  
                 Array, specifies the output vector `y`.

C:

`y`            `float*`    for `vsPow`  
                 `double*` for `vdPow`  
                 Pointer to an array that contains the output vector `y`.

## Discussion

The function `Pow` has certain limitations on the input range of `a` and `b` parameters. Specifically, if `a[i]` is positive, then `b[i]` may be arbitrary. For negative or zero `a[i]`, the value of `b[i]` must be integer (either positive or negative).

---

## Exp

*Computes an exponential  
of vector elements.*

---

### Fortran:

```
call vsexp( n, a, y )
call vdexp( n, a, y )
```

### C:

```
vsExp( n, a, y );
vdExp( n, a, y );
```

## Input Parameters

Fortran:

`n`            `INTEGER, INTENT(IN)`. Specifies the number of elements to be calculated.

*a*            `REAL, INTENT(IN)` for `vsexp`  
               `DOUBLE PRECISION, INTENT(IN)` for `vdexp`  
               Array, specifies the input vector *a*.

C:

*n*            `int`. Specifies the number of elements to be calculated.

*a*            `const float*` for `vsExp`  
               `const double*` for `vdExp`  
               Pointer to an array that contains the input vector *a*.

**Table 6-5      Precision Overflow Thresholds for `Exp` Function**

---

Data Type	Threshold Limitations on Input Parameters
single precision	$a[i] < \ln(\text{FLT\_MAX})$
double precision	$a[i] < \ln(\text{DBL\_MAX})$

---

### Output Parameters

Fortran:

*y*            `REAL` for `vsexp`  
               `DOUBLE PRECISION` for `vdexp`  
               Array, specifies the output vector *y*.

C:

*y*            `float*` for `vsExp`  
               `double*` for `vdExp`  
               Pointer to an array that contains the output vector *y*.

---

**Ln**

*Computes natural logarithm  
of vector elements.*

---

**Fortran:**

```
call vsln( n, a, y )
call vdln( n, a, y )
```

**C:**

```
vsLn( n, a, y );
vdLn( n, a, y );
```

**Input Parameters**

Fortran:

*n*            **INTEGER, INTENT(IN)**. Specifies the number of elements to be calculated.

*a*            **REAL, INTENT(IN)** for **vsln**  
**DOUBLE PRECISION, INTENT(IN)** for **vdln**  
 Array, specifies the input vector *a*.

C:

*n*            **int**. Specifies the number of elements to be calculated.  
*a*            **const float\*** for **vsLn**  
**const double\*** for **vdLn**  
 Pointer to an array that contains the input vector *a*.

**Output Parameters**

Fortran:

*y*            **REAL** for **vsln**  
**DOUBLE PRECISION** for **vdln**  
 Array, specifies the output vector *y*.

C:

*y*            **float\*** for **vsLn**  
**double\*** for **vdLn**  
 Pointer to an array that contains the output vector *y*.

## Log10

*Computes denary logarithm of vector elements.*

---

### Fortran:

```
call vslog10( n, a, y )
call vdlog10( n, a, y )
```

### C:

```
vsLog10( n, a, y );
vdLog10( n, a, y );
```

### Input Parameters

Fortran:

*n*            INTEGER, INTENT( IN ). Specifies the number of elements to be calculated.

*a*            REAL, INTENT( IN )    for vslog10  
DOUBLE PRECISION, INTENT( IN )    for vdlog10  
Array, specifies the input vector *a*.

C:

*n*            int. Specifies the number of elements to be calculated.

*a*            const float\*    for vsLog10  
const double\*    for vdLog10  
Pointer to an array that contains the input vector *a*.

### Output Parameters

Fortran:

*y*            REAL    for vslog10  
DOUBLE PRECISION for vdlog10  
Array, specifies the output vector *y*.

C:

*y*            float\*    for vsLog10  
double\*    for vdLog10  
Pointer to an array that contains the output vector *y*.

---

## Cos

*Computes cosine of vector elements.*

---

### Fortran:

```
call vscos( n, a, y )
call vdcos( n, a, y )
```

### C:

```
vsCos( n, a, Y );
vdCos( n, a, Y );
```

### Input Parameters

Fortran:

*n*            INTEGER, INTENT(IN). Specifies the number of elements to be calculated.  
*a*            REAL, INTENT(IN) for **vscos**  
                DOUBLE PRECISION, INTENT(IN) for **vdcos**  
                Array, specifies the input vector *a*.

C:

*n*            int. Specifies the number of elements to be calculated.  
*a*            const float\* for **vsCos**  
                const double\* for **vdCos**  
                Pointer to an array that contains the input vector *a*.

### Output Parameters

Fortran:

*y*            REAL for **vscos**  
                DOUBLE PRECISION for **vdcos**  
                Array, specifies the output vector *y*.

C:

*y*            float\* for **vsCos**  
                double\* for **vdCos**  
                Pointer to an array that contains the output vector *y*.

---

## Sin

*Computes sine of vector elements.*

---

### Fortran:

```
call vssin( n, a, y )
call vdsin( n, a, y )
```

### C:

```
vsSin( n, a, y );
vdSin( n, a, y );
```

### Input Parameters

Fortran:

*n*            INTEGER, INTENT(IN). Specifies the number of elements to be calculated.  
*a*            REAL, INTENT(IN) for vssin  
                DOUBLE PRECISION, INTENT(IN) for vdsin  
                Array, specifies the input vector *a*.

C:

*n*            int. Specifies the number of elements to be calculated.  
*a*            const float\* for vsSin  
                const double\* for vdSin  
                Pointer to an array that contains the input vector *a*.

### Output Parameters

Fortran:

*y*            REAL for vssin  
                DOUBLE PRECISION for vdsin  
                Array, specifies the output vector *y*.

C:

*y*            float\* for vsSin  
                double\* for vdSin  
                Pointer to an array that contains the output vector *y*.

## SinCos

*Computes sine and cosine  
of vector elements.*

---

### Fortran:

```
call vssincos( n, a, y, z )
call vdsincos( n, a, y, z )
```

### C:

```
vsSinCos( n, a, y, z );
vdSinCos( n, a, y, z );
```

### Input Parameters

Fortran:

*n*            INTEGER, INTENT(IN). Specifies the number of elements to be calculated.

*a*            REAL, INTENT(IN) for **vssincos**  
                DOUBLE PRECISION, INTENT(IN) for **vdsincos**  
                Array, specifies the input vector **a**.

C:

*n*            int. Specifies the number of elements to be calculated.

*a*            const float\* for **vsSinCos**  
                const double\* for **vdSinCos**

Pointer to an array that contains the input vector **a**.

### Output Parameters

Fortran:

*y, z*        REAL for **vssincos**

DOUBLE PRECISION for **vdsincos**

Arrays, specify the output vectors **y** (for sine values) and **z** (for cosine values).

C:

*y, z*        float\* for **vsSinCos**

double\* for **vdSinCos**

Pointers to arrays that contain the output vectors **y** (for sine values) and **z** (for cosine values).

## Tan

*Computes tangent of vector elements.*

---

### Fortran:

```
call vstan( n, a, y )
call vdtn( n, a, y )
```

### C:

```
vsTan( n, a, y );
vdTan( n, a, y );
```

### Input Parameters

Fortran:

*n*            INTEGER, INTENT(IN). Specifies the number of elements to be calculated.  
*a*            REAL, INTENT(IN) for vstan  
                DOUBLE PRECISION, INTENT(IN) for vdtn  
                Array, specifies the input vector *a*.

C:

*n*            int. Specifies the number of elements to be calculated.  
*a*            const float\* for vsTan  
                const double\* for vdTan  
                Pointer to an array that contains the input vector *a*.

### Output Parameters

Fortran:

*y*            REAL for vstan  
                DOUBLE PRECISION for vdtn  
                Array, specifies the output vector *y*.

C:

*y*            float\* for vsTan  
                double\* for vdTan  
                Pointer to an array that contains the output vector *y*.

---

## Acos

*Computes inverse cosine  
of vector elements.*

---

### Fortran:

```
call vsacos( n, a, y )
call vdacos( n, a, y )
```

### C:

```
vsAcos( n, a, y );
vdAcos( n, a, y );
```

## Input Parameters

Fortran:

*n*            INTEGER, INTENT(IN). Specifies the number of elements to be calculated.

*a*            REAL, INTENT(IN) for vsacos  
                DOUBLE PRECISION, INTENT(IN) for vdacos  
                Array, specifies the input vector *a*.

C:

*n*            int. Specifies the number of elements to be calculated.

*a*            const float\* for vsAcos  
                const double\* for vdAcos  
                Pointer to an array that contains the input vector *a*.

## Output Parameters

Fortran:

*y*            REAL for vsacos  
                DOUBLE PRECISION for vdacos  
                Array, specifies the output vector *y*.

C:

*y*            float\* for vsAcos  
                double\* for vdAcos  
                Pointer to an array that contains the output vector *y*.

## Asin

*Computes inverse sine  
of vector elements.*

### Fortran:

```
call vsasin( n, a, y )
call vdasin( n, a, y )
```

### C:

```
vsAsin( n, a, y );
vdAsin( n, a, y );
```

### Input Parameters

Fortran:

*n*            INTEGER, INTENT(IN). Specifies the number of elements to be calculated.

*a*            REAL, INTENT(IN) for vsasin  
                DOUBLE PRECISION, INTENT(IN) for vdasin  
                Array, specifies the input vector *a*.

C:

*n*            int. Specifies the number of elements to be calculated.

*a*            const float\* for vsAsin  
                const double\* for vdAsin  
                Pointer to an array that contains the input vector *a*.

### Output Parameters

Fortran:

*y*            REAL for vsasin  
                DOUBLE PRECISION for vdasin  
                Array, specifies the output vector *y*.

C:

*y*            float\* for vsAsin  
                double\* for vdAsin  
                Pointer to an array that contains the output vector *y*.

---

## Atan

*Computes inverse tangent  
of vector elements.*

---

### Fortran:

```
call vsatan( n, a, y )
call vdatan( n, a, y )
```

### C:

```
vsAtan( n, a, y );
vdAtan( n, a, y );
```

## Input Parameters

Fortran:

*n*            INTEGER, INTENT(IN). Specifies the number of elements to be calculated.

*a*            REAL, INTENT(IN) for vsatan  
                DOUBLE PRECISION, INTENT(IN) for vdatan  
                Array, specifies the input vector *a*.

C:

*n*            int. Specifies the number of elements to be calculated.

*a*            const float\* for vsAtan  
                const double\* for vdAtan  
                Pointer to an array that contains the input vector *a*.

## Output Parameters

Fortran:

*y*            REAL for vsatan  
                DOUBLE PRECISION for vdatan  
                Array, specifies the output vector *y*.

C:

*y*            float\* for vsAtan  
                double\* for vdAtan  
                Pointer to an array that contains the output vector *y*.

## Atan2

*Computes four-quadrant inverse tangent of elements of two vectors.*

### Fortran:

```
call vsatan2( n, a, b, y )
call vdatan2( n, a, b, y )
```

### C:

```
vsAtan2( n, a, b, y );
vdAtan2( n, a, b, y );
```

### Input Parameters

Fortran:

**n**            INTEGER, INTENT( IN). Specifies the number of elements to be calculated.

**a, b**        REAL, INTENT( IN)    for vsatan2  
DOUBLE PRECISION, INTENT( IN)    for vdatan2  
Arrays, specify the input vectors **a** and **b**.

C:

**n**            int. Specifies the number of elements to be calculated.

**a, b**        const float\*    for vsAtan2  
const double\*   for vdAtan2  
Pointers to arrays that contain the input vectors **a** and **b**.

### Output Parameters

Fortran:

**y**            REAL    for vsatan2  
DOUBLE PRECISION for vdatan2  
Array, specifies the output vector **y**.

C:

`y`            `float*` for `vsAtan2`  
              `double*` for `vdAtan2`

Pointer to an array that contains the output vector `y`.

The elements of the output vector `y` are computed as the four-quadrant arctangent of `a[i] / b[i]`.

---

## Cosh

*Computes hyperbolic cosine  
of vector elements.*

---

**Fortran:**

`call vscosh( n, a, y )`  
`call vdcsosh( n, a, y )`

**C:**

`vsCosh( n, a, y );`  
`vdCosh( n, a, y );`

### Input Parameters

Fortran:

`n`            `INTEGER, INTENT(IN)`. Specifies the number of elements to be calculated.

`a`            `REAL, INTENT(IN)` for `vscosh`  
`DOUBLE PRECISION, INTENT(IN)` for `vdcsosh`  
Array, specifies the input vector `a`.

C:

`n`            `int`. Specifies the number of elements to be calculated.

`a`            `const float*` for `vsCosh`  
              `const double*` for `vdCosh`

Pointer to an array that contains the input vector `a`.

**Table 6-6 Precision Overflow Thresholds for `cosh` Function**

Data Type	Threshold Limitations on Input Parameters
single precision	$-\ln(\text{FLT\_MAX}) - \ln 2 < a[i] < \ln(\text{FLT\_MAX}) + \ln 2$
double precision	$-\ln(\text{DBL\_MAX}) - \ln 2 < a[i] < \ln(\text{DBL\_MAX}) + \ln 2$

### Output Parameters

Fortran:

`y`      `REAL`    for `vscosh`  
               `DOUBLE PRECISION` for `vdcosh`  
               Array, specifies the output vector `y`.

C:

`y`      `float*`    for `vsCosh`  
               `double*` for `vdCosh`  
               Pointer to an array that contains the output vector `y`.

---

## Sinh

*Computes hyperbolic sine  
of vector elements.*

---

### Fortran:

```
call vssinh( n, a, y )
call vdsinh( n, a, y )
```

### C:

```
vsSinh( n, a, y );
vdSinh( n, a, y );
```

## Input Parameters

Fortran:

- n*            `INTEGER, INTENT(IN)`. Specifies the number of elements to be calculated.
- a*            `REAL, INTENT(IN)` for `vssinh`  
`DOUBLE PRECISION, INTENT(IN)` for `vdsinh`  
 Array, specifies the input vector *a*.

C:

- n*            `int`. Specifies the number of elements to be calculated.
- a*            `const float*` for `vsSinh`  
`const double*` for `vdSinh`  
 Pointer to an array that contains the input vector *a*.

---

**Table 6-7      Precision Overflow Thresholds for `sinh` Function**

---

Data Type	Threshold Limitations on Input Parameters
single precision	$-\ln(\text{FLT\_MAX}) - \ln 2 < a[i] < \ln(\text{FLT\_MAX}) + \ln 2$
double precision	$-\ln(\text{DBL\_MAX}) - \ln 2 < a[i] < \ln(\text{DBL\_MAX}) + \ln 2$

---

## Output Parameters

Fortran:

- y*            `REAL` for `vssinh`  
`DOUBLE PRECISION` for `vdsinh`  
 Array, specifies the output vector *y*.

C:

- y*            `float*` for `vsSinh`  
`double*` for `vdSinh`  
 Pointer to an array that contains the output vector *y*.

## Tanh

*Computes hyperbolic tangent  
of vector elements.*

### Fortran:

```
call vstanh( n, a, y )
call vdtnanh( n, a, y )
```

### C:

```
vsTanh( n, a, y );
vdTanh( n, a, y );
```

### Input Parameters

Fortran:

*n*            INTEGER, INTENT( IN ). Specifies the number of elements to be calculated.

*a*            REAL, INTENT( IN )    for vstanh  
                DOUBLE PRECISION, INTENT( IN )    for vdtnanh  
                Array, specifies the input vector *a*.

C:

*n*            int. Specifies the number of elements to be calculated.

*a*            const float\*    for vsTanh  
                const double\*    for vdTanh  
                Pointer to an array that contains the input vector *a*.

### Output Parameters

Fortran:

*y*            REAL    for vstanh  
                DOUBLE PRECISION for vdtnanh  
                Array, specifies the output vector *y*.

C:

*y*            float\*    for vsTanh  
                double\*    for vdTanh  
                Pointer to an array that contains the output vector *y*.

## Acosh

*Computes inverse hyperbolic cosine  
(nonnegative) of vector elements.*

---

### Fortran:

```
call vsacosh( n, a, y )  
call vdacosh( n, a, y )
```

### C:

```
vsAcosh( n, a, y );  
vdAcosh( n, a, y );
```

### Input Parameters

Fortran:

*n*            INTEGER, INTENT(IN). Specifies the number of elements to be calculated.

*a*            REAL, INTENT(IN) for vsacosh  
                DOUBLE PRECISION, INTENT(IN) for vdacosh  
                Array, specifies the input vector *a*.

C:

*n*            int. Specifies the number of elements to be calculated.

*a*            const float\* for vsAcosh  
                const double\* for vdAcosh  
                Pointer to an array that contains the input vector *a*.

### Output Parameters

Fortran:

*y*            REAL for vsacosh  
                DOUBLE PRECISION for vdacosh  
                Array, specifies the output vector *y*.

C:

*y*            float\* for vsAcosh  
                double\* for vdAcosh  
                Pointer to an array that contains the output vector *y*.

## Asinh

*Computes inverse hyperbolic sine  
of vector elements.*

### Fortran:

```
call vsasinh( n, a, y )
call vdasin( n, a, y )
```

### C:

```
vsAinh( n, a, y );
vdAinh( n, a, y );
```

### Input Parameters

Fortran:

*n*            INTEGER, INTENT( IN ). Specifies the number of elements to be calculated.

*a*            REAL, INTENT( IN )    for **vsasinh**  
                DOUBLE PRECISION, INTENT( IN )    for **vdasin**  
                Array, specifies the input vector **a**.

C:

*n*            int. Specifies the number of elements to be calculated.

*a*            const float\*    for **vsAinh**  
                const double\*    for **vdAinh**  
                Pointer to an array that contains the input vector **a**.

### Output Parameters

Fortran:

*y*            REAL    for **vsasinh**  
                DOUBLE PRECISION for **vdasin**  
                Array, specifies the output vector **y**.

C:

*y*            float\*    for **vsAinh**  
                double\*    for **vdAinh**  
                Pointer to an array that contains the output vector **y**.

## Atanh

*Computes inverse hyperbolic tangent  
of vector elements.*

---

### Fortran:

```
call vsatanh( n, a, y )
call vdatanh( n, a, y )
```

### C:

```
vsAtanh( n, a, y );
vdAtanh( n, a, y );
```

### Input Parameters

Fortran:

*n*            INTEGER, INTENT(IN). Specifies the number of elements to be calculated.

*a*            REAL, INTENT(IN) for vsatanh  
                DOUBLE PRECISION, INTENT(IN) for vdatanh  
                Array, specifies the input vector *a*.

C:

*n*            int. Specifies the number of elements to be calculated.

*a*            const float\* for vsAtanh  
                const double\* for vdAtanh  
                Pointer to an array that contains the input vector *a*.

### Output Parameters

Fortran:

*y*            REAL for vsatanh  
                DOUBLE PRECISION for vdatanh  
                Array, specifies the output vector *y*.

C:

*y*            float\* for vsAtanh  
                double\* for vdAtanh  
                Pointer to an array that contains the output vector *y*.

## VML Pack/Unpack Functions

This section describes VML functions which convert vectors with unit increment to and from vectors with positive increment indexing, vector indexing and mask indexing (see [Appendix A](#) for details on vector indexing methods).

[Table 6-8](#) lists available VML Pack/Unpack functions, together with data types and indexing methods associated with them.

**Table 6-8     VML Pack/Unpack Functions**

Function Short Name	Data Types	Indexing Methods	Description
<a href="#">Pack</a>	s, d	I, V, M	Gathers elements of arrays, indexed by different methods.
<a href="#">Unpack</a>	s, d	I, V, M	Scatters vector elements to arrays with different indexing.

---

## Pack

*Copies elements of an array with specified indexing to a vector with unit increment.*

---

### Fortran:

```
call vspacki( n, a, inca, y )
call vspackv( n, a, ia, y )
call vspackm( n, a, ma, y )
call vdpacki( n, a, inca, y )
call vdpackv( n, a, ia, y )
call vdpackm( n, a, ma, y )
```

**C:**

```
vsPackI( n, a, inca, y );
vsPackV( n, a, ia, y );
vsPackM( n, a, ma, y );
vdPackI( n, a, inca, y );
vdPackV( n, a, ia, y );
vdPackM( n, a, ma, y );
```

**Input Parameters**

Fortran:

<i>n</i>	INTEGER, INTENT(IN). Specifies the number of elements to be calculated.
<i>a</i>	REAL, INTENT(IN) for <b>vspacki</b> , <b>vspackv</b> , <b>vspackm</b> DOUBLE PRECISION, INTENT(IN) for <b>vdpacki</b> , <b>vdpackv</b> , <b>vdpackm</b> Array, DIMENSION at least $(1 + (n-1)*inca)$ for <b>vspacki</b> , at least $\max(n, \max(ia[j]))$ , $j=0, \dots, n-1$ , for <b>vspackv</b> , at least <i>n</i> for <b>vspackm</b> , Specifies the input vector <i>a</i> .
<i>inca</i>	INTEGER, INTENT(IN) for <b>vspacki</b> , <b>vdpacki</b> . Specifies the increment for the elements of <i>a</i> .
<i>ia</i>	INTEGER, INTENT(IN) for <b>vspackv</b> , <b>vdpackv</b> . Array, DIMENSION at least <i>n</i> Specifies the index vector for the elements of <i>a</i> .
<i>ma</i>	INTEGER, INTENT(IN) for <b>vspackm</b> , <b>vdpackm</b> . Array, DIMENSION at least <i>n</i> Specifies the mask vector for the elements of <i>a</i> .

## C:

<i>n</i>	int. Specifies the number of elements to be calculated
<i>a</i>	const float* for <b>vsPackI</b> , <b>vsPackV</b> , <b>vsPackM</b> const double* for <b>vdPackI</b> , <b>vdPackV</b> , <b>vdPackM</b> Specifies the pointer to an array that contains the input vector <i>a</i> . Size of the array must be:

at least  $(1 + (n-1)*inca)$  for **vsPackI**,  
 at least  $\max(n, \max(i_a[j]))$ ,  $j=0, \dots, n-1$ , for **vsPackV**,  
 at least  $n$  for **vsPackM**.

- inca**      **int** for **vsPackI**, **vdPackI**.  
 Specifies the increment for the elements of **a**.
- ia**      **const int\*** for **vsPackV**, **vdPackV**. Specifies the pointer to  
 an array of size at least **n** that contains the index vector  
 for the elements of **a**.
- ma**      **const int\*** for **vsPackM**, **vdPackM**. Specifies the pointer to  
 an array of size at least **n** that contains the mask vector  
 for the elements of **a**.

### Output Parameters

Fortran:

- y**      **REAL** for **vsPackI**, **vsPackV**, **vsPackM**  
**DOUBLE PRECISION** for **vdPackI**, **vdPackV**, **vdPackM**  
 Array, **DIMENSION** at least **n**, specifies the output vector **y**.

C:

- y**      **float\*** for **vsPackI**, **vsPackV**, **vsPackM**  
**double\*** for **vdPackI**, **vdPackV**, **vdPackM**  
 Specifies the pointer to an array of size at least **n** that contains  
 the output vector **y**.

---

## Unpack

*Copies elements of a vector with unit  
 increment to an array with specified  
 indexing.*

---

**Fortran:**

```
call vsunpacki( n, a, y, incy )
call vsunpackv( n, a, y, iy )
call vsunpackm( n, a, y, my )
call vdunpacki( n, a, y, incy )
```

```
call vdunpackv( n, a, y, iy )
call vdunpackm( n, a, y, my )
```

**C:**

```
vsUnpackI( n, a, y, incy );
vsUnpackV( n, a, y, iy );
vsUnpackM( n, a, y, my );
vdUnpackI( n, a, y, incy );
vdUnpackV( n, a, y, iy );
vdUnpackM( n, a, y, my );
```

**Input Parameters****Fortran:**

*n*            INTEGER, INTENT(IN). Specifies the number of elements to be calculated.

*a*            REAL, INTENT(IN) for **vsunpacki**, **vsunpackv**, **vsunpackm**  
DOUBLE PRECISION, INTENT(IN) for **vdunpacki**,  
**vdunpackv**, **vdunpackm**.

Array, DIMENSION at least *n*, specifies the input vector *a*.

*incy*        INTEGER, INTENT(IN) for **vsunpacki**, **vdunpacki**.  
Specifies the increment for the elements of *y*.

*iy*            INTEGER, INTENT(IN) for **vsunpackv**, **vdunpackv**.  
Array, DIMENSION at least *n*, specifies the index vector  
for the elements of *y*.

*my*            INTEGER, INTENT(IN) for **vsunpackm**, **vdunpackm**.  
Array, DIMENSION at least *n*, specifies the mask vector  
for the elements of *y*.

**C:**

*n*            int. Specifies the number of elements to be calculated .

*a*            const float\* for **vsUnpackI**, **vsUnpackV**, **vsUnpackM**  
const double\* for **vdUnpackI**, **vdUnpackV**, **vdUnpackM**  
Specifies the pointer to an array of size at least *n* that contains  
the input vector *a*.

*incy*        int for **vsUnpackI**, **vdUnpackI**.

Specifies the increment for the elements of *y*.

- iy*      `const int*` for `vsUnpackV`, `vdUnpackV`. Specifies the pointer to an array of size at least *n* that contains the index vector for the elements of *a*.
- my*      `const int*` for `vsUnpackM`, `vdUnpackM`. Specifies the pointer to an array of size at least *n* that contains the mask vector for the elements of *a*.

## Output Parameters

Fortran:

- y*      `REAL` for `vsunpacki`, `vsunpackv`, `vsunpackm`  
`DOUBLE PRECISION` for `vdunpacki`, `vdunpackv`,  
`vdunpackm`.  
 Array, `DIMENSION`  
     at least `(1 + (n-1)*incy)` for `vsunpacki`,  
     at least `max( n, max(iy[j]) ), j=0, ..., n-1`, for `vsunpackv`,  
     at least *n* for `vsunpackm`  
 Specifies the output vector *y*.

C:

- y*      `float*` for `vsUnpackI`, `vsUnpackV`, `vsUnpackM`  
`double*` for `vdUnpackI`, `vdUnpackV`, `vdUnpackM`  
 Specifies the pointer to an array that contains the output vector *y*.  
 Size of the array must be:  
     at least `(1 + (n-1)*incy)` for `vsUnPackI`,  
     at least `max( n, max(ia[j]) ), j=0, ..., n-1`, for `vsUnPackV`,  
     at least *n* for `vsUnPackM`.

## VML Service Functions

This section describes VML functions which allow the user to set /get the accuracy mode, and set/get the error code. All these functions are available both in Fortran- and C- interfaces.

[Table 6-9](#) lists available VML Service functions and their short description.

**Table 6-9      VML Service Functions**

---

Function Short Name	Description
<a href="#"><u>SetMode</u></a>	Sets the VML mode
<a href="#"><u>GetMode</u></a>	Gets the VML mode
<a href="#"><u>SetErrStatus</u></a>	Sets the VML error status
<a href="#"><u>GetErrStatus</u></a>	Gets the VML error status
<a href="#"><u>ClearErrStatus</u></a>	Clears the VML error status
<a href="#"><u>setErrorCallBack</u></a>	Sets the additional error handler callback function
<a href="#"><u>getErrorCallBack</u></a>	Gets the additional error handler callback function
<a href="#"><u>ClearErrorCallBack</u></a>	Deletes the additional error handler callback function

---

### SetMode

*Sets the VML mode to `mode` parameter and stores the previous VML mode to `oldmode`.*

---

#### Fortran:

```
oldmode = vmlsetmode( mode )
```

#### C:

```
oldmode = vmlSetMode( mode );
```

## Input Parameters

Fortran:

`mode`      `INTEGER, INTENT( IN )`. Specifies the VML mode to be set.

C:

`mode`      `int`. Specifies the VML mode to be set.

## Output Parameters

Fortran:

`oldmode`      `INTEGER, INTENT( IN )`. Specifies the former VML mode.

C:

`oldmode`      `int`. Specifies the former VML mode.

## Discussion

The `mode` parameter is designed to control accuracy, FPU and error handling options. [Table 6-10](#) lists values of the `mode` parameter, defined in the `mode.h` header file for C- interface and in the `VML.f90` file for Fortran- interface. All other possible values of the `mode` parameter may be obtained from these values by using bitwise OR ( | ) addition operation to combine one value for accuracy, one for FPU, and one for error control options. The default value of the `mode` parameter is `VML_HA | VML_ERRMODE_DEFAULT`. Thus, the current FPU control word (FPU precision and the rounding method) is used by default.

If any VML mathematical function requires different FPU precision, or rounding method, it changes these options automatically and then restores the former values. The `mode` parameter enables you to minimize switching the internal FPU mode inside each VML mathematical function that works with similar precision and accuracy settings. To accomplish this, set the `mode` parameter to `VML_FLOAT_CONSISTENT` for single precision functions, or to `VML_DOUBLE_CONSISTENT` for double precision functions. These values of the `mode` parameter are the optimal choice for the respective function groups, as they are required for most of the VML mathematical functions. After the execution is over, set the `mode` to `VML_RESTORE` if you need to restore the previous FPU mode.

**Table 6-10    Values of the *mode* Parameter**

<b>Value of <i>mode</i></b>	<b>Description</b>
<b>Accuracy Control</b>	
<code>VML_HA</code>	High accuracy versions of VML functions will be used
<code>VML_LA</code>	Low accuracy versions of VML functions will be used
<b>Additional FPU Mode Control</b>	
<code>VML_FLOAT_CONSISTENT</code>	The optimal FPU mode (control word) for single precision functions is set, and the previous FPU mode is saved
<code>VML_DOUBLE_CONSISTENT</code>	The optimal FPU mode (control word) for double precision functions is set, and the previous FPU mode is saved
<code>VML_RESTORE</code>	The previously saved FPU mode is restored
<b>Error Mode Control</b>	
<code>VML_ERRMODE_IGNORE</code>	No action is set for computation errors
<code>VML_ERRMODE_ERRNO</code>	On error, the <code>errno</code> variable is set
<code>VML_ERRMODE_STDERR</code>	On error, the error text information is written to <code>stderr</code>
<code>VML_ERRMODE_EXCEPT</code>	On error, an exception is raised
<code>VML_ERRMODE_CALLBACK</code>	On error, an additional error handler function is called
<code>VML_ERRMODE_DEFAULT</code>	On error, the <code>errno</code> variable is set, an exception is raised, and an additional error handler function is called

## Examples

Several examples of calling the function `vmlSetMode()` with different values of the `mode` parameter are given below:

Fortran:

```
oldmode = vmlsetmode( VML_LA )
call vmlsetmode( IOR(VML_LA, IOR(VML_FLOAT_CONSISTENT,
    VML_ERRMODE_IGNORE )))
call vmlsetmode( VML_RESTORE )
```

C:

```
vmlSetMode( VML_LA );
vmlSetMode( VML_LA | VML_FLOAT_CONSISTENT | VML_ERRMODE_IGNORE );
vmlSetMode( VML_RESTORE );
```

## GetMode

*Gets the VML mode.*

---

**Fortran:**

```
mod = vmlgetmode()
```

**C:**

```
mod = vmlGetMode( void );
```

### Output Parameters

Fortran:

`mod`        INTEGER. Specifies the full `mode` parameter.

C:

`mod`        int. Specifies the full `mode` parameter.

### Discussion

The function `vmlGetMode()` returns the VML `mode` parameter which controls accuracy, FPU and error handling options. The `mod` variable value is some combination of the values listed in the [Table 6-10](#). The values of `mode` parameter are defined in the `mode.h` header file for C- interface and in the `VML.fi` file for Fortran- interface. You can obtain some of these values using the respective mask from the [Table 6-11](#), for example:

Fortran:

```
mod = vmlgetmode()
accm = IAND(mod, VML_ACCURACY_MASK)
fpum = IAND(mod, VML_FPUMODE_MASK)
errm = IAND(mod, VML_ERRMODE_MASK)
```

C:

```
accm = vmlGetMode(void )& VML_ACCURACY_MASK;
fpum = vmlGetMode(void )& VML_FPUMODE _MASK;
errm = vmlGetMode(void )& VML_ERRMODE _MASK;
```

**Table 6-11    Values of Mask for the *mode* Parameter**

---

Value of mask	Description
VML_ACCURACY_MASK	Specifies mask for accuracy <i>mode</i> selection.
VML_FPUMODE_MASK	Specifies mask for FPU <i>mode</i> selection.
VML_ERRMODE_MASK	Specifies mask for error <i>mode</i> selection.

---

## SetErrStatus

Sets the VML error status to *err* and stores the previous VML error status to *olderr*.

---

**Fortran:**

```
olderr = vmlseterrstatus( err )
```

C:

```
olderr = vmlSetErrStatus( err );
```

### Input Parameters

Fortran:

<i>err</i>	INTEGER, INTENT(IN). Specifies the VML error status to be set.
------------	--

C:

`err`      `int`. Specifies the VML error status to be set.

### Output Parameters

Fortran:

`olderr`      `INTEGER, INTENT(IN)`. Specifies the former VML error status.

C:

`olderr`      `int`. Specifies the former VML error status.

[Table 6-12](#) lists possible values of the `err` parameter.

**Table 6-12    Values of the VML Error Status**

Error Status	Description
<code>VML_STATUS_OK</code>	The execution was completed successfully.
<code>VML_STATUS_BADSIZE</code>	The array dimension is not positive.
<code>VML_STATUS_BADMEM</code>	NULL pointer is passed.
<code>VML_STATUS_ERRDOM</code>	At least one of array values is out of a range of definition.
<code>VML_STATUS_SING</code>	At least one of array values caused a singularity.
<code>VML_STATUS_OVERFLOW</code>	An overflow has happened during the calculation process.
<code>VML_STATUS_UNDERFLOW</code>	An underflow has happened during the calculation process.

### Examples:

```
vm.setStatus( VML_STATUS_OK );
vm.setStatus( VML_STATUS_ERRDOM );
vm.setStatus( VML_STATUS_UNDERFLOW );
```

---

## GetErrStatus

*Gets the VML error status.*

---

**Fortran:**

```
err = vmlgeterrstatus( )
```

**C:**

```
err = vmlGetErrStatus( void );
```

**Output Parameters**

Fortran:

*err*      **INTEGER**. Specifies the VML error status.

C:

*err*      **int**. Specifies the VML error status.

---

## ClearErrStatus

*Sets the VML error status to  
VML\_STATUS\_OK and stores the  
previous VML error status to olderr.*

---

**Fortran:**

```
olderr = vmlclearerrstatus( )
```

**C:**

```
olderr = vmlClearErrStatus( void );
```

**Output Parameters**

Fortran:

*olderr*      **INTEGER**. Specifies the former VML error status.

---

C:

*olderr*      int. Specifies the former VML error status.

---

## SetErrorCallBack

*Sets the additional error handler  
callback function and gets the old  
callback function.*

---

**Fortran:**

*oldcallback = vmlseterrorcallback( callback )*

**C:**

*oldcallback = vmlSetErrorCallBack( callback );*

### Input Parameters

Fortran:

*callback*      Pointer to the callback function.

The callback function has the following format:

```
INTEGER FUNCTION ERRFUNC(par)
  TYPE (ERROR_STRUCTURE) par
  ! ...
  ! user error processing
  ! ...
  ERRFUNC = 0
  ! if ERRFUNC = 0 - standard VML error
  ! handler
  ! is called after the callback
  ! if ERRFUNC != 0 - standard VML error
  ! handler
  ! is not called
END
```

The passed error structure is defined as follows:

```
TYPE ERROR_STRUCTURE
SEQUENCE
  INTEGER*4 ICODE
  INTEGER*4 IINDEX
  REAL*8 DBA1
  REAL*8 DBA2
  REAL*8 DBR1
  REAL*8 DBR2
  CHARACTER(64) CFUNCNAME
  INTEGER*4 IFUNCNAMELEN
END TYPE ERROR_STRUCTURE
```

C:

*callback*

Pointer to the callback function.

The callback function has the following format:

```
static int __stdcall MyHandler(DefVmlErrorContext*
pContext)
{
/* Handler body */
};
```

The passed error structure is defined as follows:

```
typedef struct _DefVmlErrorContext
{
  int iCode;           /* Error status value */
  int iIndex;          /* Index for bad array
                        element, or bad array
                        dimension, or bad
                        array pointer */

  double dbA1;         /* Error argument 1 */
  double dbA2;         /* Error argument 2 */
  double dbR1;         /* Error result 1 */
  double dbR2;         /* Error result 2 */
  char cFuncName[64];  /* Function name */
  int iFuncNameLen;    /* Length of function name*/
} DefVmlErrorContext;
```

## Output Parameters

Fortran:

`oldcallback` Pointer to the former callback function.

C:

`oldcallback` Pointer to the former callback function.

## Discussion

The callback function is called on each VML mathematical function error if `VML_ERRMODE_CALLBACK` error mode is set (see [Table 6-10](#)).

Use the `vmlSetErrorCallBack( )` function if you need to define your own callback function instead of default empty callback function.

The input structure for a callback function contains the following information

about the encountered error:

- the input value which caused an error
- location (array index) of this value
- the computed result value
- error code
- name of the function in which the error occurred.

You can insert your own error processing into the callback function. This may include correcting the passed result values in order to pass them back and resume computation. The standard error handler is called after the callback function only if it returns 0.

---

## GetErrorCallBack

*Gets the additional error handler callback function.*

---

**Fortran:**

`fun = vmlgeterrorcallback( )`

**C:**

```
fun = vmlGetErrorCallBack( void );
```

**Output Parameters**

Fortran:

*fun*      Pointer to the callback function.

**C:**

*fun*      Pointer to the callback function.

---

## ClearErrorCallBack

*Deletes the additional error handler  
callback function and retrieves the  
former callback function.*

---

**Fortran:**

```
oldcallback = vmlclearerrorcallback( )
```

**C:**

```
oldcallback = vmlClearErrorCallBack( void );
```

**Output Parameters**

Fortran:

*oldcallback*    INTEGER. Pointer to the former callback function.

**C:**

*oldcallback*    int. Pointer to the former callback function.

# *Routine and Function Arguments*

A

The major arguments in the BLAS routines are vector and matrix, whereas VML functions work on vector arguments only.

The sections that follow discuss each of these arguments and provide examples.

## **Vector Arguments in BLAS**

Vector arguments are passed in one-dimensional arrays. The array dimension (length) and vector increment are passed as integer variables. The length determines the number of elements in the vector. The increment (also called stride) determines the spacing between vector elements and the order of the elements in the array in which the vector is passed.

A vector of length *n* and increment *incx* is passed in a one-dimensional array *x* whose values are defined as

*x(1), x(1+|incx|), ..., x(1+(n-1)\* |incx| )*

If *incx* is positive, then the elements in array *x* are stored in increasing order. If *incx* is negative, the elements in array *x* are stored in decreasing order with the first element defined as *x(1+(n-1)\* |incx| )*. If *incx* is zero, then all elements of the vector have the same value, *x(1)*. The dimension of the one-dimensional array that stores the vector must always be at least

*idimx = 1 + (n-1)\* |incx|*

**Example A-1 One-dimensional Real Array**

---

Let  $x(1:7)$  be the one-dimensional real array

$x = (1.0, 3.0, 5.0, 7.0, 9.0, 11.0, 13.0)$ .

If  $incx = 2$  and  $n = 3$ , then the vector argument with elements in order from first to last is  $(1.0, 5.0, 9.0)$ .

If  $incx = -2$  and  $n = 4$ , then the vector elements in order from first to last is  $(13.0, 9.0, 5.0, 1.0)$ .

If  $incx = 0$  and  $n = 4$ , then the vector elements in order from first to last is  $(1.0, 1.0, 1.0, 1.0)$ .

---

One-dimensional substructures of a matrix, such as the rows, columns, and diagonals, can be passed as vector arguments with the starting address and increment specified. In Fortran, storing the  $m$  by  $n$  matrix is based on column-major ordering where the increment between elements in the same column is  $1$ , the increment between elements in the same row is  $m$ , and the increment between elements on the same diagonal is  $m + 1$ .

**Example A-2 Two-dimensional Real Matrix**

---

Let  $a$  be the real  $5 \times 4$  matrix declared as `REAL A (5,4)`.

To scale the third column of  $a$  by 2.0, use the BLAS routine `sscal` with the following calling sequence:

`call sscal (5, 2.0, a(1,3), 1).`

To scale the second row, use the statement:

`call sscal (4, 2.0, a(2,1), 5).`

To scale the main diagonal of  $A$  by 2.0, use the statement:

`call sscal (5, 2.0, a(1,1), 6).`

---



---

**NOTE.** *The default vector argument is assumed to be 1.*

---

## Vector Arguments in VML

Vector arguments of VML mathematical functions are passed in one-dimensional arrays with unit vector increment. It means that a vector of length  $n$  is passed contiguously in an array  $a$  whose values are defined as  $a[0], a[1], \dots, a[n-1]$  (for C-interface).

To accommodate for arrays with other increments, or more complicated indexing, VML contains auxiliary pack/unpack functions that gather the array elements into a contiguous vector and then scatter them after the computation is complete.

Generally, if the vector elements are stored in a one-dimensional array  $a$  as

$a[m_0], a[m_1], \dots, a[m_{n-1}]$

and need to be regrouped into an array  $y$  as

$y[k_0], y[k_1], \dots, y[k_{n-1}],$

VML pack/unpack functions can use one of the following indexing methods:

### Positive Increment Indexing

$k_j = incy * j, m_j = inca * j, j = 0, \dots, n-1$

Constraint:  $incy > 0$  and  $inca > 0$ .

For example, setting  $incy = 1$  specifies gathering array elements into a contiguous vector.

This method is similar to that used in BLAS, with the exception that negative and zero increments are not permitted.

### Index Vector Indexing

$k_j = iy[j], m_j = ia[j], j = 0, \dots, n-1,$

where  $ia$  and  $iy$  are arrays of length  $n$  that contain index vectors for the input and output arrays  $a$  and  $y$ , respectively.

### Mask Vector Indexing

Indices  $k_j, m_j$  are such that:

$my[k_j] \neq 0, ma[m_j] \neq 0, j = 0, \dots, n-1,$

where  $ma$  and  $my$  are arrays that contain mask vectors for the input and output arrays  $a$  and  $y$ , respectively.

## Matrix Arguments

Matrix arguments of the Math Kernel Library routines can be stored in either one- or two-dimensional arrays, using the following storage schemes:

- conventional full storage (in a two-dimensional array)
- packed storage for Hermitian, symmetric, or triangular matrices (in a one-dimensional array)
- band storage for band matrices (in a two-dimensional array).

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

If a matrix is *triangular* (upper or lower, as specified by the argument  $uplo$ ), only the elements of the relevant triangle are stored; the remaining elements of the array need not be set.

Routines that handle symmetric or Hermitian matrices allow for either the upper or lower triangle of the matrix to be stored in the corresponding elements of the array:

if  $uplo = 'U'$ ,       $a_{ij}$  is stored in  $a(i, j)$  for  $i \leq j$ ,  
                          other elements of  $a$  need not be set.

if  $uplo = 'L'$ ,       $a_{ij}$  is stored in  $a(i, j)$  for  $j \leq i$ ,  
                          other elements of  $a$  need not be set.

**Packed storage** allows you to store symmetric, Hermitian, or triangular matrices more compactly: the relevant triangle (again, as specified by the argument  $uplo$ ) is packed by columns in a one-dimensional array  $ap$ :

if  $uplo = 'U'$ ,  $a_{ij}$  is stored in  $ap(i+j(j-1)/2)$  for  $i \leq j$

if  $uplo = 'L'$ ,  $a_{ij}$  is stored in  $ap(i+(2*n-j)*(j-1)/2)$  for  $j \leq i$ .

In descriptions of LAPACK routines, arrays with packed matrices have names ending in  $p$ .

**Band storage** is as follows: an  $m$  by  $n$  band matrix with  $kl$  non-zero sub-diagonals and  $ku$  non-zero super-diagonals is stored compactly in a two-dimensional array  $ab$  with  $kl+ku+1$  rows and  $n$  columns. Columns of the matrix are stored in the corresponding columns of the array, and diagonals of the matrix are stored in rows of the array. Thus,

$a_{ij}$  is stored in  $ab(ku+1+i-j, j)$  for  $\max(n, j-ku) \leq i \leq \min(n, j+kl)$ .

Use the band storage scheme only when  $k_1$  and  $k_u$  are much less than the matrix size  $n$ . (Although the routines work correctly for all values of  $k_1$  and  $k_u$ , it's inefficient to use the band storage if your matrices are not really banded).

When a general band matrix is supplied for *LU factorization*, space must be allowed to store  $k_1$  additional super-diagonals generated by fill-in as a result of row interchanges. This means that the matrix is stored according to the above scheme, but with  $k_1 + k_u$  super-diagonals.

The band storage scheme is illustrated by the following example, when

$m = n = 6$ ,  $k_1 = 2$ ,  $k_u = 1$ :

Banded matrix A

$$\begin{bmatrix} \mathbf{a}_{11} & \mathbf{a}_{12} & 0 & 0 & 0 & 0 \\ \mathbf{a}_{21} & \mathbf{a}_{22} & \mathbf{a}_{23} & 0 & 0 & 0 \\ \mathbf{a}_{31} & \mathbf{a}_{32} & \mathbf{a}_{33} & \mathbf{a}_{34} & 0 & 0 \\ 0 & \mathbf{a}_{42} & \mathbf{a}_{43} & \mathbf{a}_{44} & \mathbf{a}_{45} & 0 \\ 0 & 0 & \mathbf{a}_{53} & \mathbf{a}_{54} & \mathbf{a}_{55} & \mathbf{a}_{56} \\ 0 & 0 & 0 & \mathbf{a}_{64} & \mathbf{a}_{65} & \mathbf{a}_{66} \end{bmatrix}$$

Band storage of A

$$\begin{array}{ccccccc} * & * & * & + & + & + \\ * & * & + & + & + & + \\ * & \mathbf{a}_{12} & \mathbf{a}_{23} & \mathbf{a}_{34} & \mathbf{a}_{45} & \mathbf{a}_{56} \\ \mathbf{a}_{11} & \mathbf{a}_{22} & \mathbf{a}_{33} & \mathbf{a}_{44} & \mathbf{a}_{55} & \mathbf{a}_{66} \\ \mathbf{a}_{21} & \mathbf{a}_{32} & \mathbf{a}_{43} & \mathbf{a}_{54} & \mathbf{a}_{65} & * \\ \mathbf{a}_{31} & \mathbf{a}_{42} & \mathbf{a}_{53} & \mathbf{a}_{64} & * & * \end{array}$$

Array elements marked \* are not used by the routines; elements marked + need not be set on entry, but are required by the LU factorization routines to store the results. The input array will be overwritten on exit by the details of the LU factorization as follows:

$$\begin{array}{cccccc} * & * & * & \mathbf{u}_{14} & \mathbf{u}_{25} & \mathbf{u}_{36} \\ * & * & \mathbf{u}_{13} & \mathbf{u}_{24} & \mathbf{u}_{35} & \mathbf{u}_{46} \\ * & \mathbf{u}_{12} & \mathbf{u}_{23} & \mathbf{u}_{34} & \mathbf{u}_{45} & \mathbf{u}_{56} \\ \mathbf{u}_{11} & \mathbf{u}_{22} & \mathbf{u}_{33} & \mathbf{u}_{44} & \mathbf{u}_{55} & \mathbf{u}_{66} \\ \mathbf{m}_{21} & \mathbf{m}_{32} & \mathbf{m}_{43} & \mathbf{m}_{54} & \mathbf{m}_{65} & * \\ \mathbf{m}_{31} & \mathbf{m}_{42} & \mathbf{m}_{53} & \mathbf{m}_{64} & * & * \end{array}$$

where  $\mathbf{u}_{ij}$  are the elements of the upper triangular matrix U, and  $\mathbf{m}_{ij}$  are the multipliers used during factorization.

Triangular band matrices are stored in the same format, with either  $k_1 = 0$  if upper triangular, or  $k_u = 0$  if lower triangular. For symmetric or Hermitian band matrices with  $k$  sub-diagonals or super-diagonals, you need to store only the upper or lower triangle, as specified by the argument *uplo*:  
if *uplo* = 'U',  $a_{ij}$  is stored in  $ab(k+1+i-j, j)$  for  $\max(1, j-k) \leq i \leq j$   
if *uplo* = 'L',  $a_{ij}$  is stored in  $ab(1+i-j, j)$  for  $j \leq i \leq \min(n, j+k)$ .

In descriptions of LAPACK routines, arrays that hold matrices in band storage have names ending in *b*.

In Fortran, column-major ordering of storage is assumed. This means that elements of the same column occupy successive storage locations.

Three quantities are usually associated with a two-dimensional array argument: its leading dimension, which specifies the number of storage locations between elements in the same row, its number of rows, and its number of columns. For a matrix in full storage, the leading dimension of the array must be at least as large as the number of rows in the matrix.

A character transposition parameter is often passed to indicate whether the matrix argument is to be used in normal or transposed form or, for a complex matrix, if the conjugate transpose of the matrix is to be used. The values of the transposition parameter for these three cases are the following:

'N' or 'n'	normal (no conjugation, no transposition)
'T' or 't'	transpose
'C' or 'c'	conjugate transpose.

**Example A-3 Two-Dimensional Complex Array**

Suppose `A (1:5, 1:4)` is the complex two-dimensional array presented by matrix

$$\begin{bmatrix} (1.1, 0.11) & (1.2, 0.12) & (1.3, 0.13) & (1.4, 0.14) \\ (2.1, 0.21) & (2.2, 0.22) & (2.3, 0.23) & (2.4, 0.24) \\ (3.1, 0.31) & (3.2, 0.32) & (3.3, 0.33) & (3.4, 0.34) \\ (4.1, 0.41) & (4.2, 0.42) & (4.3, 0.43) & (4.4, 0.44) \\ (5.1, 0.51) & (5.2, 0.52) & (5.3, 0.53) & (5.4, 0.54) \end{bmatrix}$$

Let `transa` be the transposition parameter, `m` be the number of rows, `n` be the number of columns, and `lda` be the leading dimension. Then if `transa = 'N'`, `m = 4`, `n = 2`, and `lda = 5`, the matrix argument would be

$$\begin{bmatrix} (1.1, 0.11) & (1.2, 0.12) \\ (2.1, 0.21) & (2.2, 0.22) \\ (3.1, 0.31) & (3.2, 0.32) \\ (4.1, 0.41) & (4.2, 0.42) \end{bmatrix}$$

If `transa = 'T'`, `m = 4`, `n = 2`, and `lda = 5`, the matrix argument would be

$$\begin{bmatrix} (1.1, 0.11) & (2.1, 0.21) & (3.1, 0.31) & (4.1, 0.41) \\ (1.2, 0.12) & (2.2, 0.22) & (3.2, 0.32) & (4.2, 0.42) \end{bmatrix}$$

If `transa = 'C'`, `m = 4`, `n = 2`, and `lda = 5`, the matrix argument would be

$$\begin{bmatrix} (1.1, -0.11) & (2.1, -0.21) & (3.1, -0.31) & (4.1, -0.41) \\ (1.2, -0.12) & (2.2, -0.22) & (3.2, -0.32) & (4.2, -0.42) \end{bmatrix}$$

Note that care should be taken when using a leading dimension value which is different from the number of rows specified in the declaration of the two-dimensional array. For example, suppose the array `A` above is declared as `COMPLEX A (5, 4)`.

continued \*

Then if `transa = 'N'` , `m = 3` , `n = 4`, and `lda = 4`, the matrix argument will be

$$\begin{bmatrix} (1.1, 0.11) & (5.1, 0.51) & (4.2, 0.42) & (3.3, 0.33) \\ (2.1, 0.21) & (1.2, 0.12) & (5.2, 0.52) & (4.3, 0.43) \\ (3.1, 0.31) & (2.2, 0.22) & (1.3, 0.13) & (5.3, 0.53) \end{bmatrix}$$

---

# *Code Examples*

B

This appendix presents code examples of using BLAS routines and functions.

## **Example B-1 Using BLAS Level 1 Function**



[?dot](#)  
[description](#)

The following example illustrates a call to the BLAS Level 1 function `sdot`. This function performs a vector-vector operation of computing a scalar product of two single-precision real vectors `x` and `y`.

### **Parameters**

`n`              Specifies the order of vectors `x` and `y`.  
`incx`            Specifies the increment for the elements of `x`.  
`incy`            Specifies the increment for the elements of `y`.

```
program dot_main
real x(10), y(10), sdot, res
integer n, incx, incy, i
external sdot
n = 5
incx = 2
incy = 1
do i = 1, 10
    x(i) = 2.0e0
    y(i) = 1.0e0
end do
```

---

continued \*

## Example B-1 Using BLAS Level 1 Function (continued)

---

```
res = sdot (n, x, incx, y, incy)
print*, 'SDOT = ', res
end
```

As a result of this program execution, the following line is printed:

SDOT = 10.000

---

## Example B-2 Using BLAS Level 1 Routine

---



[?copy](#)  
[description](#)

The following example illustrates a call to the BLAS Level 1 routine `scopy`. This routine performs a vector-vector operation of copying a single-precision real vector `x` to a vector `y`.

### Parameters

<code>n</code>	Specifies the order of vectors <code>x</code> and <code>y</code> .
<code>incx</code>	Specifies the increment for the elements of <code>x</code> .
<code>incy</code>	Specifies the increment for the elements of <code>y</code> .

```
program copy_main
real x(10), y(10)
integer n, incx, incy, i
n = 3
incx = 3
incy = 1
do i = 1, 10
    x(i) = i
end do
call scopy (n, x, incx, y, incy)
print*, 'Y = ', (y(i), i = 1, n)
end
```

As a result of this program execution, the following line is printed:

Y = 1.00000 4.00000 7.00000

---

---

### Example B-3 Using BLAS Level 2 Routine

---



[?ger](#)  
[description](#)

The following example illustrates a call to the BLAS Level 2 routine `sger`.

This routine performs a matrix-vector operation

`a := alpha*x*y' + a.`

#### Parameters

`alpha`      Specifies a scalar `alpha`.

`x`              `m`-element vector.

`y`              `n`-element vector.

`a`              `m` by `n` matrix.

```
program ger_main
real a(5,3), x(10), y(10), alpha
integer m, n, incx, incy, i, j, lda
m = 2
n = 3
lda = 5
incx = 2
incy = 1
alpha = 0.5
do i = 1, 10
    x(i) = 1.0
    y(i) = 1.0
end do
do i = 1, m
    do j = 1, n
        a(i,j) = j
    end do
end do
call sger (m, n, alpha, x, incx, y, incy, a, lda)
print*, 'Matrix A: '
do i = 1, m
    print*, (a(i,j), j = 1, n)
end do
end
```

---

continued \*

## Example B-3 Using BLAS Level 2 Routine (continued)

---

As a result of this program execution, matrix *a* is printed as follows:

Matrix A:

1.50000 2.50000 3.50000

1.50000 2.50000 3.50000

---

## Example B-4 Using BLAS Level 3 Routine

---



[?symm  
description](#)

The following example illustrates a call to the BLAS Level 3 routine *ssymm*. This routine performs a matrix-matrix operation

*c* := *alpha*\**a*\**b*' + *beta*\**c*.

### Parameters

*alpha*              Specifies a scalar *alpha*.

*beta*              Specifies a scalar *beta*.

*a*                  Symmetric matrix.

*b*                  *m* by *n* matrix.

*c*                  *m* by *n* matrix.

```
program symm_main
real a(3,3), b(3,2), c(3,3), alpha, beta
integer m, n, lda, ldb, ldc, i, j
character uplo, side
uplo = 'u'
side = 'l'
m = 3
n = 2
lda = 3
ldb = 3
ldc = 3
alpha = 0.5
beta = 2.0
```

---

continued \*

---

**Example B-4 Using BLAS Level 3 Routine (continued)**

---

```
do i = 1, m
    do j = 1, m
        a(i,j) = 1.0
    end do
end do
do i = 1, m
    do j = 1, n
        c(i,j) = 1.0
        b(i,j) = 2.0
    end do
end do
call ssymm (side, uplo, m, n, alpha, a, lda, b, ldb,
beta, c, ldc)
print*, 'Matrix C: '
do i = 1, m
    print*, (c(i,j), j = 1, n)
end do
end
```

As a result of this program execution, matrix *c* is printed as follows:

Matrix C:

```
5.00000 5.00000
5.00000 5.00000
5.00000 5.00000
```

---

### Example B-5 Calling a Complex BLAS Level 1 Function from C

---

The following example illustrates a call from a C program to the complex BLAS Level 1 function `zdotc()`. This function computes the dot product of two double-precision complex vectors.

In this example, the complex dot product is returned in the structure `c`.

```
#define N 5
void main()
{
    int n, inca = 1, incb = 1, i;
    typedef struct{ double re; double im; } complex16;
    complex16 a[N], b[N], c;
    void zdotc();
    n = N;
    for( i = 0; i < n; i++ ){
        a[i].re = (double)i; a[i].im = (double)i * 2.0;
        b[i].re = (double)(n - i); b[i].im = (double)i * 2.0;
    }
    zdotc( &c, &n, a, &inca, b, &incb );
    printf( "The complex dot product is: ( %6.2f, %6.2f
)\n", c.re, c.im );
}
```

---



**NOTE.** Instead of calling BLAS directly from C programs, you might wish to use the CBLAS interface; this is the supported way of calling BLAS from C. For more information about CBLAS, see Appendix C, [“CBLAS Interface to the BLAS”](#).

---

# *CBLAS Interface to the BLAS*



This appendix presents CBLAS, the C interface to the Basic Linear Algebra Subprograms (BLAS).

Similar to BLAS, the CBLAS interface includes three levels of functions:

- [Level 1 CBLAS](#) (vector-vector operations)
- [Level 2 CBLAS](#) (matrix-vector operations)
- [Level 3 CBLAS](#) (matrix-matrix operations).

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

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

## CBLAS Arguments

The arguments of CBLAS functions obey the following rules:

- Input arguments are declared with the `const` modifier.
- Non-complex scalar input arguments are passed by value.
- Complex scalar input arguments are passed as void pointers.
- Array arguments are passed by address.
- Output scalar arguments are passed by address.
- BLAS character arguments are replaced by the appropriate enumerated type.

- Level 2 and Level 3 routines acquire an additional parameter of type `CBLAS_ORDER` as their first argument. This parameter specifies whether two-dimensional arrays are row-major (`CblasRowMajor`) or column-major (`CblasColMajor`).

## Enumerated Types

The CBLAS interface uses the following enumerated types:

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

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

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

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

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

## Level 1 CBLAS

This is an interface to [BLAS Level 1 Routines and Functions](#), which perform basic vector-vector operations.

### ?asum

```
float cblas_sasum(const int N, const float *X, const int incX);
double cblas_dasum(const int N, const double *X, const int
incX);
float cblas_scasum(const int N, const void *X, const int incX);
double cblas_dzasum(const int N, const void *X, const int
incX);
```

### ?axpy

```
void cblas_saxpy(const int N, const float alpha, const float
*X, const int incX, float *Y, const int incY);
void cblas_daxpy(const int N, const double alpha, const double
*X, const int incX, double *Y, const int incY);
void cblas_caxpy(const int N, const void *alpha, const void *X,
const int incX, void *Y, const int incY);
void cblas_zaxpy(const int N, const void *alpha, const void *X,
const int incX, void *Y, const int incY);
```

### ?copy

```
void cblas_scopy(const int N, const float *X, const int incX,
float *Y, const int incY);
void cblas_dcopy(const int N, const double *X, const int incX,
double *Y, const int incY);
void cblas_ccopy(const int N, const void *X, const int incX,
void *Y, const int incY);
void cblas_zcopy(const int N, const void *X, const int incX,
void *Y, const int incY);
```

### ?dot

```
float cblas_sdot(const int N, const float *X, const int incX,
const float *Y, const int incY);
double cblas_ddot(const int N, const double *X, const int incX,
const double *Y, const int incY);
```

**?dotc**

```
void cblas_cdotc_sub(const int N, const void *X, const int
incX, const void *Y, const int incY, void *dotc);
void cblas_zdotc_sub(const int N, const void *X, const int
incX, const void *Y, const int incY, void *dotc);
```

**?dotu**

```
void cblas_cdotu_sub(const int N, const void *X, const int
incX, const void *Y, const int incY, void *dotu);
void cblas_zdotu_sub(const int N, const void *X, const int
incX, const void *Y, const int incY, void *dotu);
```

**?nrm2**

```
float cblas_snrm2(const int N, const float *X, const int incX);
double cblas_dnrm2(const int N, const double *X, const int
incX);
float cblas_scnrm2(const int N, const void *X, const int incX);
double cblas_dznrm2(const int N, const void *X, const int
incX);
```

**?rot**

```
void cblas_srot(const int N, float *X, const int incX, float
*Y, const int incY, const float c, const float s);
void cblas_drot(const int N, double *X, const int incX, double
*Y, const int incY, const double c, const double s);
```

**?rotg**

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

**?rotm**

```
void cblas_srotm(const int N, float *X, const int incX, float
*Y, const int incY, const float *P);
void cblas_drotm(const int N, double *X, const int incX, double
*Y, const int incY, const double *P);
```

**?rotmg**

```
void cblas_srotmg(float *d1, float *d2, float *b1, const float
b2, float *P);
void cblas_drotmg(double *d1, double *d2, double *b1, const
double b2, double *P);
```

**?scal**

```
void cblas_sscal(const int N, const float alpha, float *X,
const int incX);
void cblas_dscal(const int N, const double alpha, double *X,
const int incX);
void cblas_cscal(const int N, const void *alpha, void *X, const
int incX);
void cblas_zscal(const int N, const void *alpha, void *X, const
int incX);
void cblas_csscal(const int N, const float alpha, void *X,
const int incX);
void cblas_zdscal(const int N, const double alpha, void *X,
const int incX);
```

**?swap**

```
void cblas_sswap(const int N, float *X, const int incX, float
*Y, const int incY);
void cblas_dswap(const int N, double *X, const int incX, double
*Y, const int incY);
void cblas_cswap(const int N, void *X, const int incX, void *Y,
const int incY);
void cblas_zswap(const int N, void *X, const int incX, void *Y,
const int incY);
```

**i?amax**

```
CBLAS_INDEX cblas_isamax(const int N, const float *X, const int
incX);
CBLAS_INDEX cblas_idamax(const int N, const double *X, const
int incX);
CBLAS_INDEX cblas_icamax(const int N, const void *X, const int
incX);
CBLAS_INDEX cblas_izamax(const int N, const void *X, const int
incX);
```

**i?amin**

```
CBLAS_INDEX cblas_isamin(const int N, const float *X, const int
incX);
CBLAS_INDEX cblas_idamin(const int N, const double *X, const
int incX);
CBLAS_INDEX cblas_icamin(const int N, const void *X, const int
incX);
CBLAS_INDEX cblas_izamin(const int N, const void *X, const int
incX);
```

## Level 2 CBLAS

This is an interface to [BLAS Level 2 Routines](#), which perform basic matrix-vector operations. Each C routine in this group has an additional parameter of type `CBLAS_ORDER` (the first argument) that determines whether the two-dimensional arrays use column-major or row-major storage.

### ?gbmv

```
void cblas_sgbmv(const enum CBLAS_ORDER order, const enum
CBLAS_TRANSPOSE TransA, const int M, const int N, const int KL,
const int KU, const float alpha, const float *A, const int lda,
const float *X, const int incX, const float beta, float *Y,
const int incY);

void cblas_dgbmv(const enum CBLAS_ORDER order, const enum
CBLAS_TRANSPOSE TransA, const int M, const int N, const int KL,
const int KU, const double alpha, const double *A, const int lda,
const double *X, const int incX, const double beta, double *Y,
const int incY);

void cblas_cgbmv(const enum CBLAS_ORDER order, const enum
CBLAS_TRANSPOSE TransA, const int M, const int N, const int KL,
const int KU, const void *alpha, const void *A, const int lda,
const void *X, const int incX, const void *beta, void *Y, const
int incY);

void cblas_zgbmv(const enum CBLAS_ORDER order, const enum
CBLAS_TRANSPOSE TransA, const int M, const int N, const int KL,
const int KU, const void *alpha, const void *A, const int lda,
const void *X, const int incX, const void *beta, void *Y, const
int incY);
```

### ?gemv

```
void cblas_sgmv(const enum CBLAS_ORDER order, const enum
CBLAS_TRANSPOSE TransA, const int M, const int N, const float
alpha, const float *A, const int lda, const float *X, const int
incX, const float beta, float *Y, const int incY);

void cblas_dgmv(const enum CBLAS_ORDER order, const enum
CBLAS_TRANSPOSE TransA, const int M, const int N, const double
alpha, const double *A, const int lda, const double *X, const
int incX, const double beta, double *Y, const int incY);

void cblas_cgmv(const enum CBLAS_ORDER order, const enum
CBLAS_TRANSPOSE TransA, const int M, const int N, const void
*alpha, const void *A, const int lda, const void *X, const int
incX, const void *beta, void *Y, const int incY);
```

```
void cblas_zgemv(const enum CBLAS_ORDER order, const enum
CBLAS_TRANSPOSE TransA, const int M, const int N, const void
*alpha, const void *A, const int lda, const void *X, const int
incX, const void *beta, void *Y, const int incY);
```

#### ?ger

```
void cblas_sger(const enum CBLAS_ORDER order, const int M,
const int N, const float alpha, const float *X, const int incX,
const float *Y, const int incY, float *A, const int lda);
void cblas_dger(const enum CBLAS_ORDER order, const int M,
const int N, const double alpha, const double *X, const int
incX, const double *Y, const int incY, double *A, const int
lda);
```

#### ?gerc

```
void cblas_cgerc(const enum CBLAS_ORDER order, const int M,
const int N, const void *alpha, const void *X, const int incX,
const void *Y, const int incY, void *A, const int lda);
void cblas_zgerc(const enum CBLAS_ORDER order, const int M,
const int N, const void *alpha, const void *X, const int incX,
const void *Y, const int incY, void *A, const int lda);
```

#### ?geru

```
void cblas_cgerus(const enum CBLAS_ORDER order, const int M,
const int N, const void *alpha, const void *X, const int incX,
const void *Y, const int incY, void *A, const int lda);
void cblas_zgeru(const enum CBLAS_ORDER order, const int M,
const int N, const void *alpha, const void *X, const int incX,
const void *Y, const int incY, void *A, const int lda);
```

#### ?hbmv

```
void cblas_chbmv(const enum CBLAS_ORDER order, const enum
CBLAS_UPLO Uplo, const int N, const int K, const void *alpha,
const void *A, const int lda, const void *X, const int incX,
const void *beta, void *Y, const int incY);
void cblas_zhbmv(const enum CBLAS_ORDER order, const enum
CBLAS_UPLO Uplo, const int N, const int K, const void *alpha,
const void *A, const int lda, const void *X, const int incX,
const void *beta, void *Y, const int incY);
```

#### ?hemv

```
void cblas_chemv(const enum CBLAS_ORDER order, const enum
CBLAS_UPLO Uplo, const int N, const void *alpha, const void *A,
const int lda, const void *X, const int incX, const void *beta,
void *Y, const int incY);
```

```
void cblas_zhemv(const enum CBLAS_ORDER order, const enum  
CBLAS_UPLO Uplo, const int N, const void *alpha, const void *A,  
const int lda, const void *X, const int incX, const void *beta,  
void *Y, const int incY);
```

### ?her

```
void cblas_cher(const enum CBLAS_ORDER order, const enum  
CBLAS_UPLO Uplo, const int N, const float alpha, const void *X,  
const int incX, void *A, const int lda);
```

```
void cblas_zher(const enum CBLAS_ORDER order, const enum  
CBLAS_UPLO Uplo, const int N, const double alpha, const void  
*X, const int incX, void *A, const int lda);
```

### ?her2

```
void cblas_cher2(const enum CBLAS_ORDER order, const enum  
CBLAS_UPLO Uplo, const int N, const void *alpha, const void *X,  
const int incX, const void *Y, const int incY, void *A, const  
int lda);
```

```
void cblas_zher2(const enum CBLAS_ORDER order, const enum  
CBLAS_UPLO Uplo, const int N, const void *alpha, const void *X,  
const int incX, const void *Y, const int incY, void *A, const  
int lda);
```

### ?hpmv

```
void cblas_chpmv(const enum CBLAS_ORDER order, const enum  
CBLAS_UPLO Uplo, const int N, const void *alpha, const void  
*Ap, const void *X, const int incX, const void *beta, void *Y,  
const int incY);
```

```
void cblas_zhpmv(const enum CBLAS_ORDER order, const enum  
CBLAS_UPLO Uplo, const int N, const void *alpha, const void  
*Ap, const void *X, const int incX, const void *beta, void *Y,  
const int incY);
```

### ?hpr

```
void cblas_chpr(const enum CBLAS_ORDER order, const enum  
CBLAS_UPLO Uplo, const int N, const float alpha, const void *X,  
const int incX, void *A);
```

```
void cblas_zhpr(const enum CBLAS_ORDER order, const enum  
CBLAS_UPLO Uplo, const int N, const double alpha, const void  
*X, const int incX, void *A);
```

### ?hpr2

```
void cblas_chpr2(const enum CBLAS_ORDER order, const enum  
CBLAS_UPLO Uplo, const int N, const void *alpha, const void *X,  
const int incX, const void *Y, const int incY, void *Ap);
```

```
void cblas_zhpr2(const enum CBLAS_ORDER order, const enum
CBLAS_UPLO Uplo, const int N, const void *alpha, const void *X,
const int incX, const void *Y, const int incY, void *Ap);
```

#### **?sbmv**

```
void cblas_ssbbmv(const enum CBLAS_ORDER order, const enum
CBLAS_UPLO Uplo, const int N, const int K, const float alpha,
const float *A, const int lda, const float *X, const int incX,
const float beta, float *Y, const int incY);
```

```
void cblas_dsbmv(const enum CBLAS_ORDER order, const enum
CBLAS_UPLO Uplo, const int N, const int K, const double alpha,
const double *A, const int lda, const double *X, const int
incX, const double beta, double *Y, const int incY);
```

#### **?spmv**

```
void cblas_sspmv(const enum CBLAS_ORDER order, const enum
CBLAS_UPLO Uplo, const int N, const float alpha, const float
*Ap, const float *X, const int incX, const float beta, float
*Y, const int incY);
```

```
void cblas_dspmv(const enum CBLAS_ORDER order, const enum
CBLAS_UPLO Uplo, const int N, const double alpha, const double
*Ap, const double *X, const int incX, const double beta, double
*Y, const int incY);
```

#### **?spr**

```
void cblas_sspr(const enum CBLAS_ORDER order, const enum
CBLAS_UPLO Uplo, const int N, const float alpha, const float
*X, const int incX, float *Ap);
```

```
void cblas_dspr(const enum CBLAS_ORDER order, const enum
CBLAS_UPLO Uplo, const int N, const double alpha, const double
*X, const int incX, double *Ap);
```

#### **?spr2**

```
void cblas_sspr2(const enum CBLAS_ORDER order, const enum
CBLAS_UPLO Uplo, const int N, const float alpha, const float
*X, const int incX, const float *Y, const int incY, float *A);
```

```
void cblas.dspr2(const enum CBLAS_ORDER order, const enum
CBLAS_UPLO Uplo, const int N, const double alpha, const double
*X, const int incX, const double *Y, const int incY, double
*A);
```

**?symv**

```
void cblas_ssymv(const enum CBLAS_ORDER order, const enum  
CBLAS_UPLO Uplo, const int N, const float alpha, const float  
*A, const int lda, const float *X, const int incX, const float  
beta, float *Y, const int incY);  
void cblas_dsymv(const enum CBLAS_ORDER order, const enum  
CBLAS_UPLO Uplo, const int N, const double alpha, const double  
*A, const int lda, const double *X, const int incX, const  
double beta, double *Y, const int incY);
```

**?syr**

```
void cblas_ssyr(const enum CBLAS_ORDER order, const enum  
CBLAS_UPLO Uplo, const int N, const float alpha, const float  
*X, const int incX, float *A, const int lda);  
void cblas_dsyrr(const enum CBLAS_ORDER order, const enum  
CBLAS_UPLO Uplo, const int N, const double alpha, const double  
*X, const int incX, double *A, const int lda);
```

**?syr2**

```
void cblas_ssyr2(const enum CBLAS_ORDER order, const enum  
CBLAS_UPLO Uplo, const int N, const float alpha, const float  
*X, const int incX, const float *Y, const int incY, float *A,  
const int lda);  
void cblas_dsyrr2(const enum CBLAS_ORDER order, const enum  
CBLAS_UPLO Uplo, const int N, const double alpha, const double  
*X, const int incX, const double *Y, const int incY, double *A,  
const int lda);
```

**?tbmv**

```
void cblas_stbmv(const enum CBLAS_ORDER order, const enum  
CBLAS_UPLO Uplo, const enum CBLAS_TRANSPOSE TransA, const enum  
CBLAS_DIAG Diag, const int N, const int K, const float *A,  
const int lda, float *X, const int incX);  
void cblas_dtbmv(const enum CBLAS_ORDER order, const enum  
CBLAS_UPLO Uplo, const enum CBLAS_TRANSPOSE TransA, const enum  
CBLAS_DIAG Diag, const int N, const int K, const double *A,  
const int lda, double *X, const int incX);  
void cblas_ctbmv(const enum CBLAS_ORDER order, const enum  
CBLAS_UPLO Uplo, const enum CBLAS_TRANSPOSE TransA, const enum  
CBLAS_DIAG Diag, const int N, const int K, const void *A, const  
int lda, void *X, const int incX);
```

```
void cblas_ztgemv(const enum CBLAS_ORDER order, const enum
CBLAS_UPLO Uplo, const enum CBLAS_TRANSPOSE TransA, const enum
CBLAS_DIAG Diag, const int N, const int K, const void *A, const
int lda, void *X, const int incX);
```

#### ?tbsv

```
void cblas_stbsv(const enum CBLAS_ORDER order, const enum
CBLAS_UPLO Uplo, const enum CBLAS_TRANSPOSE TransA, const enum
CBLAS_DIAG Diag, const int N, const int K, const float *A,
const int lda, float *X, const int incX);
```

```
void cblas_dtbsv(const enum CBLAS_ORDER order, const enum
CBLAS_UPLO Uplo, const enum CBLAS_TRANSPOSE TransA, const enum
CBLAS_DIAG Diag, const int N, const int K, const double *A,
const int lda, double *X, const int incX);
```

```
void cblas_ctbsv(const enum CBLAS_ORDER order, const enum
CBLAS_UPLO Uplo, const enum CBLAS_TRANSPOSE TransA, const enum
CBLAS_DIAG Diag, const int N, const int K, const void *A, const
int lda, void *X, const int incX);
```

```
void cblas_ztbsv(const enum CBLAS_ORDER order, const enum
CBLAS_UPLO Uplo, const enum CBLAS_TRANSPOSE TransA, const enum
CBLAS_DIAG Diag, const int N, const int K, const void *A, const
int lda, void *X, const int incX);
```

#### ?tpmv

```
void cblas_stpmv(const enum CBLAS_ORDER order, const enum
CBLAS_UPLO Uplo, const enum CBLAS_TRANSPOSE TransA, const enum
CBLAS_DIAG Diag, const int N, const float *Ap, float *X, const
int incX);
```

```
void cblas_dtpmv(const enum CBLAS_ORDER order, const enum
CBLAS_UPLO Uplo, const enum CBLAS_TRANSPOSE TransA, const enum
CBLAS_DIAG Diag, const int N, const double *Ap, double *X,
const int incX);
```

```
void cblas_ctpmv(const enum CBLAS_ORDER order, const enum
CBLAS_UPLO Uplo, const enum CBLAS_TRANSPOSE TransA, const enum
CBLAS_DIAG Diag, const int N, const void *Ap, void *X, const int
incX);
```

```
void cblas_ztpmv(const enum CBLAS_ORDER order, const enum
CBLAS_UPLO Uplo, const enum CBLAS_TRANSPOSE TransA, const enum
CBLAS_DIAG Diag, const int N, const void *Ap, void *X, const int
incX);
```

**?tpsv**

```
void cblas_stpsv(const enum CBLAS_ORDER order, const enum
CBLAS_UPLO Uplo, const enum CBLAS_TRANSPOSE TransA, const enum
CBLAS_DIAG Diag, const int N,const float *Ap, float *X, const
int incX);

void cblas_dtpsv(const enum CBLAS_ORDER order, const enum
CBLAS_UPLO Uplo, const enum CBLAS_TRANSPOSE TransA, const enum
CBLAS_DIAG Diag, const int N,const double *Ap, double *X, const
int incX);

void cblas_ctpsv(const enum CBLAS_ORDER order, const enum
CBLAS_UPLO Uplo, const enum CBLAS_TRANSPOSE TransA, const enum
CBLAS_DIAG Diag, const int N,const void *Ap, void *X, const int
incX);

void cblas_ztpsv(const enum CBLAS_ORDER order, const enum
CBLAS_UPLO Uplo, const enum CBLAS_TRANSPOSE TransA, const enum
CBLAS_DIAG Diag, const int N,const void *Ap, void *X, const int
incX);
```

**?trmv**

```
void cblas_strmv(const enum CBLAS_ORDER order, const enum
CBLAS_UPLO Uplo, const enum CBLAS_TRANSPOSE TransA, const enum
CBLAS_DIAG Diag, const int N,const float *A, const int lda,
float *X, const int incX);

void cblas_dtrmv(const enum CBLAS_ORDER order, const enum
CBLAS_UPLO Uplo, const enum CBLAS_TRANSPOSE TransA, const enum
CBLAS_DIAG Diag, const int N,const double *A, const int lda,
double *X, const int incX);

void cblas_ctrmv(const enum CBLAS_ORDER order, const enum
CBLAS_UPLO Uplo, const enum CBLAS_TRANSPOSE TransA, const enum
CBLAS_DIAG Diag, const int N,const void *A, const int lda, void
*X, const int incX);

void cblas_ztrmv(const enum CBLAS_ORDER order, const enum
CBLAS_UPLO Uplo, const enum CBLAS_TRANSPOSE TransA, const enum
CBLAS_DIAG Diag, const int N,const void *A, const int lda, void
*X, const int incX);
```

**?trsv**

```
void cblas_strsv(const enum CBLAS_ORDER order, const enum
CBLAS_UPLO Uplo, const enum CBLAS_TRANSPOSE TransA, const enum
CBLAS_DIAG Diag, const int N,const float *A, const int lda,
float *X, const int incX);
```

```
void cblas_dtrsv(const enum CBLAS_ORDER order, const enum
CBLAS_UPLO Uplo, const enum CBLAS_TRANSPOSE TransA, const enum
CBLAS_DIAG Diag, const int N,const double *A, const int lda,
double *X, const int incX);

void cblas_ctrsv(const enum CBLAS_ORDER order, const enum
CBLAS_UPLO Uplo, const enum CBLAS_TRANSPOSE TransA, const enum
CBLAS_DIAG Diag, const int N,const void *A, const int lda, void
*X, const int incX);

void cblas_ztrsv(const enum CBLAS_ORDER order, const enum
CBLAS_UPLO Uplo, const enum CBLAS_TRANSPOSE TransA, const enum
CBLAS_DIAG Diag, const int N,const void *A, const int lda, void
*X, const int incX);
```

## Level 3 CBLAS

This is an interface to [BLAS Level 3 Routines](#), which perform basic matrix-matrix operations. Each C routine in this group has an additional parameter of type `CBLAS_ORDER` (the first argument) that determines whether the two-dimensional arrays use column-major or row-major storage.

### ?gemm

```
void cblas_sgemm(const enum CBLAS_ORDER Order, const enum
CBLAS_TRANSPOSE TransA, const enum CBLAS_TRANSPOSE TransB,
const int M, const int N, const int K, const float alpha, const
float *A, const int lda, const float *B, const int ldb, const
float beta, float *C, const int ldc);
void cblas_dgemm(const enum CBLAS_ORDER Order, const enum
CBLAS_TRANSPOSE TransA, const enum CBLAS_TRANSPOSE TransB,
const int M, const int N, const int K, const double alpha,
const double *A, const int lda, const double *B, const int ldb,
const double beta, double *C, const int ldc);
void cblas_cgemm(const enum CBLAS_ORDER Order, const enum
CBLAS_TRANSPOSE TransA, const enum CBLAS_TRANSPOSE TransB,
const int M, const int N, const int K, const void *alpha, const
void *A, const int lda, const void *B, const int ldb, const
void *beta, void *C, const int ldc);
void cblas_zgemm(const enum CBLAS_ORDER Order, const enum
CBLAS_TRANSPOSE TransA, const enum CBLAS_TRANSPOSE TransB,
const int M, const int N, const int K, const void *alpha, const
void *A, const int lda, const void *B, const int ldb, const
void *beta, void *C, const int ldc);
```

### ?hemm

```
void cblas_chemm(const enum CBLAS_ORDER Order, const enum
CBLAS_SIDE Side, const enum CBLAS_UPLO Uplo, const int M, const
int N, const void *alpha, const void *A, const int lda, const
void *B, const int ldb, const void *beta, void *C, const int
ldc);
void cblas_zhemm(const enum CBLAS_ORDER Order, const enum
CBLAS_SIDE Side, const enum CBLAS_UPLO Uplo, const int M, const
int N, const void *alpha, const void *A, const int lda, const
void *B, const int ldb, const void *beta, void *C, const int
ldc);
```

### ?herk

```
void cblas_cherk(const enum CBLAS_ORDER Order, const enum
CBLAS_UPLO Uplo, const enum CBLAS_TRANSPOSE Trans, const int N,
const int K, const float alpha, const void *A, const int lda,
const float beta, void *C, const int ldc);
void cblas_zherk(const enum CBLAS_ORDER Order, const enum
CBLAS_UPLO Uplo, const enum CBLAS_TRANSPOSE Trans, const int N,
const int K, const double alpha, const void *A, const int lda,
const double beta, void *C, const int ldc);
```

### ?her2k

```
void cblas_cher2k(const enum CBLAS_ORDER Order, const enum
CBLAS_UPLO Uplo, const enum CBLAS_TRANSPOSE Trans, const int N,
const int K, const void *alpha, const void *A, const int lda,
const void *B, const int ldb, const float beta, void *C, const
int ldc);
void cblas_zher2k(const enum CBLAS_ORDER Order, const enum
CBLAS_UPLO Uplo, const enum CBLAS_TRANSPOSE Trans, const int N,
const int K, const void *alpha, const void *A, const int lda,
const void *B, const int ldb, const double beta, void *C, const
int ldc);
```

### ?symm

```
void cblas_ssymmm(const enum CBLAS_ORDER Order, const enum
CBLAS_SIDE Side, const enum CBLAS_UPLO Uplo, const int M, const
int N, const float alpha, const float *A, const int lda, const
float *B, const int ldb, const float beta, float *C, const int
ldc);
void cblas_dsymmm(const enum CBLAS_ORDER Order, const enum
CBLAS_SIDE Side, const enum CBLAS_UPLO Uplo, const int M, const
int N, const double alpha, const double *A, const int lda,
const double *B, const int ldb, const double beta, double *C,
const int ldc);
void cblas_csymmm(const enum CBLAS_ORDER Order, const enum
CBLAS_SIDE Side, const enum CBLAS_UPLO Uplo, const int M, const
int N, const void *alpha, const void *A, const int lda, const
void *B, const int ldb, const void *beta, void *C, const int
ldc);
void cblas_zsymmm(const enum CBLAS_ORDER Order, const enum
CBLAS_SIDE Side, const enum CBLAS_UPLO Uplo, const int M, const
int N, const void *alpha, const void *A, const int lda, const
void *B, const int ldb, const void *beta, void *C, const int
ldc);
```

**?syrk**

```
void cblas_ssyrk(const enum CBLAS_ORDER Order, const enum
CBLAS_UPLO Uplo, const enum CBLAS_TRANSPOSE Trans, const int N,
const int K, const float alpha, const float *A, const int lda,
const float beta, float *C, const int ldc);
void cblas_dsyrk(const enum CBLAS_ORDER Order, const enum
CBLAS_UPLO Uplo, const enum CBLAS_TRANSPOSE Trans, const int N,
const int K, const double alpha, const double *A, const int lda,
const double beta, double *C, const int ldc);
void cblas_csyrk(const enum CBLAS_ORDER Order, const enum
CBLAS_UPLO Uplo, const enum CBLAS_TRANSPOSE Trans, const int N,
const int K, const void *alpha, const void *A, const int lda,
const void *beta, void *C, const int ldc);
void cblas_zsyrk(const enum CBLAS_ORDER Order, const enum
CBLAS_UPLO Uplo, const enum CBLAS_TRANSPOSE Trans, const int N,
const int K, const void *alpha, const void *A, const int lda,
const void *beta, void *C, const int ldc);
```

**?syr2k**

```
void cblas_ssyr2k(const enum CBLAS_ORDER Order, const enum
CBLAS_UPLO Uplo, const enum CBLAS_TRANSPOSE Trans, const int N,
const int K, const float alpha, const float *A, const int lda,
const float *B, const int ldb, const float beta, float *C,
const int ldc);
void cblas_dsyr2k(const enum CBLAS_ORDER Order, const enum
CBLAS_UPLO Uplo, const enum CBLAS_TRANSPOSE Trans, const int N,
const int K, const double alpha, const double *A, const int lda,
const double *B, const int ldb, const double beta, double
*C, const int ldc);
void cblas_csyr2k(const enum CBLAS_ORDER Order, const enum
CBLAS_UPLO Uplo, const enum CBLAS_TRANSPOSE Trans, const int N,
const int K, const void *alpha, const void *A, const int lda,
const void *B, const int ldb, const void *beta, void *C, const
int ldc);
void cblas_zsyr2k(const enum CBLAS_ORDER Order, const enum
CBLAS_UPLO Uplo, const enum CBLAS_TRANSPOSE Trans, const int N,
const int K, const void *alpha, const void *A, const int lda,
const void *B, const int ldb, const void *beta, void *C, const
int ldc);
```

### ?trmm

```
void cblas_strmm(const enum CBLAS_ORDER Order, const enum
CBLAS_SIDE Side, const enum CBLAS_UPLO Uplo, const enum
CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag, const int
M, const int N, const float alpha, const float *A, const int
lda, float *B, const int ldb);
void cblas_dtrmm(const enum CBLAS_ORDER Order, const enum
CBLAS_SIDE Side, const enum CBLAS_UPLO Uplo, const enum
CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag, const int
M, const int N, const double alpha, const double *A, const int
lda, double *B, const int ldb);
void cblas_ctrmm(const enum CBLAS_ORDER Order, const enum
CBLAS_SIDE Side, const enum CBLAS_UPLO Uplo, const enum
CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag, const int
M, const int N, const void *alpha, const void *A, const int
lda, void *B, const int ldb);
void cblas_ztrmm(const enum CBLAS_ORDER Order, const enum
CBLAS_SIDE Side, const enum CBLAS_UPLO Uplo, const enum
CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag, const int
M, const int N, const void *alpha, const void *A, const int
lda, void *B, const int ldb);
```

### ?trsm

```
void cblas_strsm(const enum CBLAS_ORDER Order, const enum
CBLAS_SIDE Side, const enum CBLAS_UPLO Uplo, const enum
CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag, const int
M, const int N, const float alpha, const float *A, const int
lda, float *B, const int ldb);
void cblas_dtrsm(const enum CBLAS_ORDER Order, const enum
CBLAS_SIDE Side, const enum CBLAS_UPLO Uplo, const enum
CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag, const int
M, const int N, const double alpha, const double *A, const int
lda, double *B, const int ldb);
void cblas_ctrsm(const enum CBLAS_ORDER Order, const enum
CBLAS_SIDE Side, const enum CBLAS_UPLO Uplo, const enum
CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag, const int
M, const int N, const void *alpha, const void *A, const int
lda, void *B, const int ldb);
void cblas_ztrsm(const enum CBLAS_ORDER Order, const enum
CBLAS_SIDE Side, const enum CBLAS_UPLO Uplo, const enum
CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag, const int
M, const int N, const void *alpha, const void *A, const int
lda, void *B, const int ldb);
```

# Glossary

---

$A^H$	Denotes the conjugate of a general matrix $A$ . <i>See also</i> conjugate matrix.
$A^T$	Denotes the transpose of a general matrix $A$ . <i>See also</i> transpose.
band matrix	A general $m$ by $n$ matrix $A$ such that $a_{ij} = 0$ for $ i - j  > l$ , where $1 < l < \min(m, n)$ . For example, any tridiagonal matrix is a band matrix.
band storage	A special storage scheme for band matrices. A matrix is stored in a two-dimensional array: columns of the matrix are stored in the corresponding columns of the array, and <i>diagonals</i> of the matrix are stored in rows of the array.
BLAS	Abbreviation for Basic Linear Algebra Subprograms. These subprograms implement vector, matrix-vector, and matrix-matrix operations.
Bunch-Kaufman factorization	Representation of a real symmetric or complex Hermitian matrix $A$ in the form $A = PUDU^H P^T$ (or $A = PLDL^H P^T$ ) where $P$ is a permutation matrix, $U$ and $L$ are upper and lower triangular matrices with unit diagonal, and $D$ is a Hermitian block-diagonal matrix with 1-by-1 and 2-by-2 diagonal blocks. $U$ and $L$ have 2-by-2 unit diagonal blocks corresponding to the 2-by-2 blocks of $D$ .

c	When found as the first letter of routine names, c indicates the usage of single-precision complex data type.
CBLAS	C interface to the BLAS. See BLAS.
Cholesky factorization	Representation of a symmetric positive-definite or, for complex data, Hermitian positive-definite matrix A in the form $A = U^H U$ or $A = LL^H$ , where L is a lower triangular matrix and U is an upper triangular matrix.
condition number	The number $\kappa(A)$ defined for a given square matrix A as follows: $\kappa(A) = \ A\  \ A^{-1}\ $ .
conjugate matrix	The matrix $A^H$ defined for a given general matrix A as follows: $(A^H)_{ij} = (a_{ji})^*$ .
conjugate number	The conjugate of a complex number $z = a + bi$ is $z^* = a - bi$ .
d	When found as the first letter of routine names, d indicates the usage of double-precision real data type.
dot product	The number denoted $x \cdot y$ and defined for given vectors x and y as follows: $x \cdot y = \sum_i x_i y_i$ . Here $x_i$ and $y_i$ stand for the $i$ th elements of x and y, respectively.
double precision	A floating-point data type. On Intel® processors, this data type allows you to store real numbers x such that $2.23 \times 10^{-308} <  x  < 1.79 \times 10^{308}$ . For this data type, the machine precision $\epsilon$ is approximately $10^{-15}$ , which means that double-precision numbers usually contain no more than 15 significant decimal digits. For more information, refer to <i>Pentium® Processor Family Developer's Manual, Volume 3: Architecture and Programming Manual</i> .
eigenvalue	See eigenvalue problem.

eigenvalue problem	A problem of finding non-zero vectors $x$ and numbers $\lambda$ (for a given square matrix $A$ ) such that $Ax = \lambda x$ . Here the numbers $\lambda$ are called the <i>eigenvalues</i> of the matrix $A$ and the vectors $x$ are called the <i>eigenvectors</i> of the matrix $A$ .
eigenvector	<i>See</i> eigenvalue problem.
elementary reflector (Householder matrix)	Matrix of a general form $H = I - \tau vv^T$ , where $v$ is a column vector and $\tau$ is a scalar. In LAPACK elementary reflectors are used, for example, to represent the matrix $Q$ in the $QR$ factorization (the matrix $Q$ is represented as a product of elementary reflectors).
factorization	Representation of a matrix as a product of matrices. <i>See also</i> Bunch-Kaufman factorization, Cholesky factorization, $LU$ factorization, $LQ$ factorization, $QR$ factorization, Schur factorization.
FFTs	Abbreviation for Fast Fourier Transforms. <i>See</i> Chapter 3 of this book.
full storage	A storage scheme allowing you to store matrices of any kind. A matrix $A$ is stored in a two-dimensional array $a$ , with the matrix element $a_{ij}$ stored in the array element $a(i, j)$ .
Hermitian matrix	A square matrix $A$ that is equal to its conjugate matrix $A^H$ . The conjugate $A^H$ is defined as follows: $(A^H)_{ij} = (a_{ji})^*$ .
$I$	<i>See</i> identity matrix.
identity matrix	A square matrix $I$ whose diagonal elements are 1, and off-diagonal elements are 0. For any matrix $A$ , $AI = A$ and $IA = A$ .
in-place	Qualifier of an operation. A function that performs its operation in-place takes its input from an array and returns its output to the same array.

inverse matrix	The matrix denoted as $A^{-1}$ and defined for a given square matrix $A$ as follows: $AA^{-1} = A^{-1}A = I$ . $A^{-1}$ does not exist for singular matrices $A$ .
$LQ$ factorization	Representation of an $m$ by $n$ matrix $A$ as $A = LQ$ or $A = (L \ 0)Q$ . Here $Q$ is an $n$ by $n$ orthogonal (unitary) matrix. For $m \leq n$ , $L$ is an $m$ by $m$ lower triangular matrix with real diagonal elements; for $m > n$ ,
	$L = \begin{bmatrix} L_1 \\ L_2 \end{bmatrix}$
	where $L_1$ is an $n$ by $n$ lower triangular matrix, and $L_2$ is a rectangular matrix.
$LU$ factorization	Representation of a general $m$ by $n$ matrix $A$ as $A = PLU$ , where $P$ is a permutation matrix, $L$ is lower triangular with unit diagonal elements (lower trapezoidal if $m > n$ ) and $U$ is upper triangular (upper trapezoidal if $m < n$ ).
machine precision	The number $\epsilon$ determining the precision of the machine representation of real numbers. For Intel® architecture, the machine precision is approximately $10^{-7}$ for single-precision data, and approximately $10^{-15}$ for double-precision data. The precision also determines the number of significant decimal digits in the machine representation of real numbers. <i>See also</i> double precision and single precision.
MKL	Abbreviation for Math Kernel Library.
orthogonal matrix	A real square matrix $A$ whose transpose and inverse are equal, that is, $A^T = A^{-1}$ , and therefore $AA^T = A^TA = I$ . All eigenvalues of an orthogonal matrix have the absolute value 1.
packed storage	A storage scheme allowing you to store symmetric, Hermitian, or triangular matrices more compactly. The upper or lower triangle of a matrix is packed by columns in a one-dimensional array.

positive-definite matrix	A square matrix $A$ such that $Ax \cdot x > 0$ for any non-zero vector $x$ . Here $\cdot$ denotes the dot product.
$QR$ factorization	Representation of an $m$ by $n$ matrix $A$ as $A = QR$ , where $Q$ is an $m$ by $m$ orthogonal (unitary) matrix, and $R$ is $n$ by $n$ upper triangular with real diagonal elements (if $m \geq n$ ) or trapezoidal (if $m < n$ ) matrix.
s	When found as the first letter of routine names, $s$ indicates the usage of single-precision real data type.
Schur factorization	Representation of a square matrix $A$ in the form $A = TZ^H$ . Here $T$ is an upper quasi-triangular matrix (for complex $A$ , triangular matrix) called the Schur form of $A$ ; the matrix $Z$ is orthogonal (for complex $A$ , unitary). Columns of $Z$ are called Schur vectors.
single precision	A floating-point data type. On Intel® processors, this data type allows you to store real numbers $x$ such that $1.18*10^{-38} <  x  < 3.40*10^{38}$ . For this data type, the machine precision ( $\varepsilon$ ) is approximately $10^{-7}$ , which means that single-precision numbers usually contain no more than 7 significant decimal digits. For more information, refer to <i>Pentium® Processor Family Developer's Manual, Volume 3: Architecture and Programming Manual</i> .
singular matrix	A matrix whose determinant is zero. If $A$ is a singular matrix, the inverse $A^{-1}$ does not exist, and the system of equations $Ax = b$ does not have a unique solution (that is, there exist no solutions or an infinite number of solutions).
singular value	The numbers defined for a given general matrix $A$ as the eigenvalues of the matrix $AA^H$ . See also SVD.
SMP	Abbreviation for Symmetric MultiProcessing. The MKL offers performance gains through parallelism provided by the SMP feature.

sparse BLAS	Routines performing basic vector operations on sparse vectors. Sparse BLAS routines take advantage of vectors' sparsity: they allow you to store only non-zero elements of vectors. <i>See</i> BLAS.
sparse vectors	Vectors in which most of the components are zeros.
storage scheme	The way of storing matrices. <i>See</i> full storage, packed storage, and band storage.
SVD	Abbreviation for Singular Value Decomposition. <i>See also</i> Singular value decomposition section in Chapter 5.
symmetric matrix	A square matrix $A$ such that $a_{ij} = a_{ji}$ .
transpose	The transpose of a given matrix $A$ is a matrix $A^T$ such that $(A^T)_{ij} = a_{ji}$ (rows of $A$ become columns of $A^T$ , and columns of $A$ become rows of $A^T$ ).
trapezoidal matrix	A matrix $A$ such that $A = (A_1 A_2)$ , where $A_1$ is an upper triangular matrix, $A_2$ is a rectangular matrix.
triangular matrix	A matrix $A$ is called an upper (lower) triangular matrix if all its subdiagonal elements (superdiagonal elements) are zeros. Thus, for an upper triangular matrix $a_{ij} = 0$ when $i > j$ ; for a lower triangular matrix $a_{ij} = 0$ when $i < j$ .
tridiagonal matrix	A matrix whose non-zero elements are in three diagonals only: the leading diagonal, the first subdiagonal, and the first super-diagonal.
unitary matrix	A complex square matrix $A$ whose conjugate and inverse are equal, that is, that is, $A^H = A^{-1}$ , and therefore $AA^H = A^HA = I$ . All eigenvalues of a unitary matrix have the absolute value 1.
VML	Abbreviation for Vector Mathematical Library. <i>See</i> Chapter 6 of this book.
z	When found as the first letter of routine names, z indicates the usage of double-precision complex data type.

# *Index*

---

## **Routines**

---

?asum, 2-5  
?axpy, 2-6  
?axpyi, 2-116  
?bdsqr, 5-94, 5-98  
?copy, 2-7  
?dot, 2-8  
?dotc, 2-9  
?dotci, 2-119  
?doti, 2-118  
?dotu, 2-10  
?dotui, 2-120  
?fft1d, 3-4, 3-8  
?fft1dc, 3-5, 3-10, 3-15  
?fft2d, 3-19, 3-22, 3-28  
?fft2dc, 3-20, 3-24, 3-29  
?gbbrd, 5-79  
?gbcon, 4-69  
?gbmv, 2-24  
?gbrfs, 4-100  
?gbtrf, 4-11  
?gbtrs, 4-37  
?gebak, 5-193  
?gebal, 5-190  
?gebrd, 5-76  
?gecon, 4-66  
?gees, 5-376  
?geesx, 5-381  
?geev, 5-387  
?geevx, 5-391  
?gehrd, 5-178  
?gelqf, 5-25, 5-36  
?gels, 5-276  
?gelsd, 5-286  
?gelss, 5-283  
?gelsy, 5-279  
?gemm, 2-83  
?gemv, 2-27  
?geqpf, 5-11, 5-14  
?geqrdf, 5-8, 5-48, 5-60, 5-68, 5-71  
?ger, 2-30  
?gerc, 2-31  
?gerfs, 4-97, 4-103  
?geru, 2-33  
?gesdd, 5-402  
?gesvd, 5-397  
?getrf, 4-9  
?getri, 4-138  
?getrs, 4-35  
?ggbak, 5-231  
?ggbal, 5-228  
?gges, 5-478

?ggesx, 5-485	?hetrd, 5-111
?ggev, 5-493	?hetrf, 4-26
?ggevx, 5-498	?hetri, 4-146
?ggglm, 5-293	?hetrs, 4-53
?gghrd, 5-225	?hgeqz, 5-233
?gglse, 5-290	?hpcon, 4-88
?ggsvd, 5-406	?hpev, 5-326
?ggsvp, 5-264	?hpevd, 5-331
?gtcon, 4-71	?hpevx, 5-339
?gthr, 2-121	?hpgst, 5-164
?gthrz, 2-122	?hpgv, 5-438
?gttrf, 4-13	?hpgvd, 5-444
?gttrs, 4-39	?hpgvx, 5-452
?hbev, 5-345	?hpmv, 2-44
?hbevd, 5-350	?hpr, 2-47
?hbevx, 5-358	?hpr2, 2-49
?hbgst, 5-169	?hprfs, 4-127
?hbgv, 5-459	?hptrd, 5-122
?hbgyd, 5-465	?hptrf, 4-32
?hbgyx, 5-473	?hptri, 4-150
?hbtrd, 5-130	?hptrs, 4-57
?hecon, 4-83	?hsein, 5-199
?heev, 5-298	?hseqr, 5-195
?heevd, 5-303	?nrm2, 2-11
?heevr, 5-319	?opgr, 5-119
?heevx, 5-310	?opmtr, 5-120
?hegst, 5-160	?orgbr, 5-82
?hegv, 5-415	?orghr, 5-180
?hegyd, 5-421	?orglq, 5-28, 5-38, 5-40
?hegyx, 5-430	?orgqr, 5-17, 5-50, 5-52
?hemm, 2-86	?orgtr, 5-107
?hemv, 2-38	?ormbr, 5-85
?her, 2-40	?ormhr, 5-182
?her2, 2-42	?ormlq, 5-30, 5-42, 5-45, 5-54, 5-57, 5-62, 5-65
?her2k, 2-92	?ormqr, 5-19
?herfs, 4-121	?ormtr, 5-109
?herk, 2-89	?pbcon, 4-77

?pbrfs, 4-112  
?pbstf, 5-172  
?pbtrf, 4-19  
?pbtrs, 4-47  
?pocon, 4-73  
?porfs, 4-106, 4-115  
?potrf, 4-15  
?potri, 4-140  
?potrs, 4-42  
?ppcon, 4-75  
?pprfs, 4-109  
?pptrf, 4-17  
?pptri, 4-142  
?pptrs, 4-45  
?ptcon, 4-79  
?pteqr, 5-146  
?pttrf, 4-21  
?pttrs, 4-49  
?rot, 2-12  
?rotg, 2-14  
?roti, 2-123  
?rotm, 2-15  
?rotmg, 2-17  
?sbev, 5-343  
?sbevd, 5-347  
?sbenvx, 5-354  
?sbgst, 5-166  
?sbgv, 5-456  
?sbgvd, 5-462  
?sbgvx, 5-469  
?sbmv, 2-51  
?sbtrd, 5-128  
?scal, 2-18  
?sctr, 2-124  
?spcon, 4-86  
?spev, 5-324  
?spevd, 5-328  
?spevx, 5-335  
?spgst, 5-162  
?spgv, 5-435  
?spgvd, 5-441  
?spgvx, 5-448  
?spmv, 2-54  
?spr, 2-56  
?spr2, 2-58  
?sprfs, 4-124  
?sptrd, 5-117  
?sptrf, 4-29  
?sptri, 4-148  
?sptrs, 4-55  
?stebz, 5-141, 5-149  
?stein, 5-152  
?steqr, 5-134, 5-137  
?sterf, 5-132  
?stev, 5-362  
?stevd, 5-364  
?stevr, 5-371  
?stevx, 5-367  
?swap, 2-20  
?sycon, 4-81, 5-154  
?syev, 5-296  
?syevd, 5-300  
?syevr, 5-314  
?syevx, 5-306  
?sygst, 5-158  
?sygv, 5-412  
?sygvd, 5-418  
?sygvx, 5-425  
?symm, 2-96  
?symv, 2-60  
?syr, 2-62  
?syr2, 2-64  
?syr2k, 2-103  
?syrfs, 4-118

- ?syrk, 2-100  
?sytrd, 5-105  
?sytrf, 4-23  
?sytri, 4-144  
?sytrs, 4-51  
?tbcon, 4-94  
?tbmv, 2-66  
?tbsv, 2-69  
?tbtrs, 4-64  
?tgevc, 5-239  
?tgexc, 5-244  
?tgsen, 5-247  
?tgsja, 5-268  
?tgnsa, 5-258  
?tgssyl, 5-253  
?tpcon, 4-92  
?tpmv, 2-72  
?tprhs, 4-132  
?tpsv, 2-75  
?tptri, 4-153  
?ptrs, 4-62  
?trcon, 4-90  
?trevc, 5-205  
?trexc, 5-214  
?trmm, 2-107  
?trmv, 2-77  
?trrfs, 4-129  
?trsens, 5-216  
?trsm, 2-110  
?trsna, 5-209  
?trsv, 2-79  
?trssyl, 5-221  
?trtri, 4-152  
?trtrs, 4-59  
?ungbr, 5-88  
?unghr, 5-185  
?unglq, 5-32  
?ungqr, 5-21  
?ungtr, 5-113  
?unmbr, 5-91  
?unmhr, 5-187  
?unmlq, 5-34  
?unmqr, 5-23  
?unmtr, 5-115  
?upgtr, 5-124  
?upmtr, 5-125
- A**
- absolute value of a vector element  
largest, 2-21  
smallest, 2-22  
accuracy modes, in VML, 6-2  
adding magnitudes of the vector elements, 2-5  
arguments  
matrix, A-4  
sparse vector, 2-114  
vector, A-1
- B**
- balancing a matrix, 5-190  
band storage scheme, A-4  
bidiagonal matrix, 5-74  
BLAS Level 1 functions  
?asum, 2-4, 2-5  
?dot, 2-4, 2-8  
?dotc, 2-4, 2-9  
?dotu, 2-4, 2-10  
?nrm2, 2-4, 2-11  
code example, B-1, B-2  
iamax, 2-4, 2-21  
iamin, 2-4, 2-22  
BLAS Level 1 routines  
?axpy, 2-4, 2-6  
?copy, 2-4, 2-7  
?rot, 2-4, 2-12

?rotg, 2-4, 2-14  
?rotm, 2-15  
?rotmg, 2-17  
?scal, 2-4, 2-18  
?swap, 2-4, 2-20  
code example, B-2

BLAS Level 2 routines  
?gbmv, 2-23, 2-24  
?gemv, 2-23, 2-27  
?ger, 2-23, 2-30  
?gerc, 2-23, 2-31  
?geru, 2-23, 2-33  
?hbmv, 2-23, 2-35  
?hemv, 2-23, 2-38  
?her, 2-23, 2-40  
?her2, 2-23, 2-42  
?hpmv, 2-23, 2-44  
?hpr, 2-23, 2-47  
?hpr2, 2-23, 2-49  
?sbmv, 2-23, 2-51  
?spmv, 2-23, 2-54  
?spr, 2-23, 2-56  
?spr2, 2-23, 2-58  
?symv, 2-23, 2-60  
?syr, 2-23, 2-62  
?syr2, 2-23, 2-64  
?tbmv, 2-24, 2-66  
?tbsv, 2-24, 2-69  
?tpmv, 2-24, 2-72  
?tpsv, 2-24, 2-75  
?trmv, 2-24, 2-77  
?trsv, 2-24, 2-79  
code example, B-3, B-4

BLAS Level 3 routines  
?gemm, 2-82, 2-83  
?hemm, 2-82, 2-86  
?her2k, 2-82, 2-92  
?herk, 2-82, 2-89  
?symm, 2-82, 2-96  
?syr2k, 2-82, 2-103  
?syrk, 2-82, 2-100  
?trmm, 2-82, 2-107  
?trsm, 2-82, 2-110  
code example, B-4, B-5

BLAS routines  
matrix arguments, A-4  
routine groups, 1-4, 2-1  
vector arguments, A-1

Bunch-Kaufman factorization, 4-8

Hermitian matrix, 4-26  
    packed storage, 4-32

symmetric matrix, 4-23  
    packed storage, 4-29

**C**

C interface, 3-2

CBLAS, C-1  
    arguments, C-1  
    level 1 (vector operations), C-3  
    level 2 (matrix-vector operations), C-6  
    level 3 (matrix-matrix operations), C-14

Cholesky factorization  
    Hermitian positive-definite matrix, 4-15  
        band storage, 4-19  
        packed storage, 4-17  
    symmetric positive-definite matrix, 4-15  
        band storage, 4-19  
        packed storage, 4-17

code examples  
    BLAS Level 1 function, B-1  
    BLAS Level 1 routine, B-2  
    BLAS Level 2 routine, B-3  
    BLAS Level 3 routine, B-4

complex-to-complex one-dimensional FFTs, 3-3

complex-to-complex two-dimensional FFTs,  
    3-18

complex-to-real one-dimensional FFTs, 3-11

complex-to-real two-dimensional FFTs, 3-27

Computational Routines, 5-6

condition number  
    band matrix, 4-69  
    general matrix, 4-66

Hermitian matrix, 4-83  
    packed storage, 4-88

Hermitian positive-definite matrix, 4-73

- band storage, 4-77
  - packed storage, 4-75
  - tridiagonal, 4-79
  - symmetric matrix, 4-81, 5-154
    - packed storage, 4-86
  - symmetric positive-definite matrix, 4-73
    - band storage, 4-77
    - packed storage, 4-75
    - tridiagonal, 4-79
  - triangular matrix, 4-90
    - band storage, 4-94
    - packed storage, 4-92
  - tridiagonal matrix, 4-71
  - converting a sparse vector into compressed storage form, 2-121
    - and writing zeros to the original vector, 2-122
  - converting compressed sparse vectors into full storage form, 2-124
  - copying vectors, 2-7
- D**
- data structure requirements for FFTs, 3-2
  - data type
    - in VML, 6-2
    - shorthand, 1-6
  - dimension, A-1
  - dot product
    - complex vectors, conjugated, 2-9
    - complex vectors, unconjugated, 2-10
    - real vectors, 2-8
    - sparse complex vectors, 2-120
    - sparse complex vectors, conjugated, 2-119
    - sparse real vectors, 2-118
  - Driver Routines, 5-275
- E**
- eigenvalue problems
    - general matrix, 5-174, 5-224
    - generalized form, 5-157
- Hermitian matrix, 5-101
  - symmetric matrix, 5-101
  - eigenvalues. *See* eigenvalue problems
  - eigenvectors. *See* eigenvalue problems
  - error diagnostics, in VML, 6-6
  - Error reporting routine, XERBLA, 2-1
  - errors in solutions of linear equations
    - general matrix, 4-97, 4-103
      - band storage, 4-100
    - Hermitian matrix, 4-121
      - packed storage, 4-127
    - Hermitian positive-definite matrix, 4-106, 4-115
      - band storage, 4-112
      - packed storage, 4-109
    - symmetric matrix, 4-118
      - packed storage, 4-124
    - symmetric positive-definite matrix, 4-106, 4-115
      - band storage, 4-112
      - packed storage, 4-109
  - triangular matrix, 4-129
    - band storage, 4-135
    - packed storage, 4-132
  - Euclidean norm
    - of a vector, 2-11
- F**
- factorization
    - See also* triangular factorization
  - Bunch-Kaufman, 4-8
  - Cholesky, 4-8
  - LU, 4-8
  - orthogonal (LQ, QR), 5-7
  - fast Fourier transforms, 1-2
    - C interface, 3-2
    - data storage types, 3-2
    - data structure requirements, 3-2
    - routines
      - ?fft1d, 3-4, 3-8, 3-13
      - ?fft1dc, 3-5, 3-10, 3-15

?fft2d, 3-19, 3-22, 3-28  
?fft2dc, 3-20, 3-24, 3-29

**FFT.** *See* fast Fourier transforms

**finding**

- element of a vector with the largest absolute value, 2-21
- element of a vector with the smallest absolute value, 2-22

**font conventions**, 1-6

**forward or inverse FFTs**, 3-4, 3-5, 3-19, 3-20

**full storage scheme**, A-4

**function name conventions**, in VML, 6-2

**G**

---

**gathering sparse vector's elements into compressed form**, 2-121

- and writing zeros to these elements, 2-122

**general matrix**

- eigenvalue problems, 5-174, 5-224
- estimating the condition number, 4-66
  - band storage, 4-69
- inverting the matrix, 4-138
- LQ factorization, 5-25, 5-36
- LU factorization, 4-9
  - band storage, 4-11
- matrix-vector product, 2-27
  - band storage, 2-24
- QR factorization, 5-8, 5-48, 5-60, 5-68, 5-71
  - with pivoting, 5-11, 5-14
- rank-l update, 2-30
- rank-l update, conjugated, 2-31
- rank-l update, unconjugated, 2-33
- scalar-matrix-matrix product, 2-83
- solving systems of linear equations, 4-35
  - band storage, 4-37

**generalized eigenvalue problems**, 5-157

*See also* LAPACK routines, generalized eigenvalue problems

**complex Hermitian-definite problem**, 5-160

- band storage, 5-169
- packed storage, 5-164

real symmetric-definite problem, 5-158

- band storage, 5-166
- packed storage, 5-162

**Givens rotation**

- modified Givens transformation parameters, 2-17
- of sparse vectors, 2-123
- parameters, 2-14

**H**

---

**Hermitian matrix**, 5-101, 5-157

- Bunch-Kaufman factorization, 4-26
  - packed storage, 4-32
- estimating the condition number, 4-83
  - packed storage, 4-88
- generalized eigenvalue problems, 5-157
- inverting the matrix, 4-146
  - packed storage, 4-150
- matrix-vector product, 2-38
  - band storage, 2-35
  - packed storage, 2-44
- rank-1 update, 2-40
  - packed storage, 2-47
- rank-2 update, 2-42
  - packed storage, 2-49
- rank-2k update, 2-92
- rank-n update, 2-89
- scalar-matrix-matrix product, 2-86
- solving systems of linear equations, 4-53
  - packed storage, 4-57

**Hermitian positive-definite matrix**

- Cholesky factorization, 4-15
  - band storage, 4-19
  - packed storage, 4-17
- estimating the condition number, 4-73
  - band storage, 4-77
  - packed storage, 4-75
- inverting the matrix, 4-140
  - packed storage, 4-142
- solving systems of linear equations, 4-42
  - band storage, 4-47
  - packed storage, 4-45

**I**

i?amax, 2-21  
i?amin, 2-22  
increment, A-1  
inverse matrix. *See* inverting a matrix  
inverting a matrix  
    general matrix, 4-138  
    Hermitian matrix, 4-146  
        packed storage, 4-150  
    Hermitian positive-definite matrix, 4-140  
        packed storage, 4-142  
    symmetric matrix, 4-144  
        packed storage, 4-148  
    symmetric positive-definite matrix, 4-140  
        packed storage, 4-142  
triangular matrix, 4-152  
    packed storage, 4-153

?spgst, 5-162  
?sygst, 5-158  
LQ factorization  
    ?gelqf, 5-25, 5-36  
    ?orglq, 5-28, 5-38, 5-40  
    ?ormlq, 5-30, 5-42, 5-45, 5-45, 5-54, 5-57,  
        5-62, 5-65  
    ?unglq, 5-32  
    ?unmlq, 5-34  
matrix inversion  
    ?getri, 4-138  
    ?hetri, 4-146  
    ?hptri, 4-150  
    ?potri, 4-140  
    ?pptri, 4-142  
    ?sptri, 4-148  
    ?sytri, 4-144  
    ?tptri, 4-153  
    ?trtri, 4-152  
nonsymmetric eigenvalue problems  
    ?gebak, 5-193  
    ?gebal, 5-190  
    ?gehrd, 5-178  
    ?hsein, 5-199  
    ?hseqr, 5-195  
    ?orghr, 5-180  
    ?ormhr, 5-182  
    ?trevc, 5-205  
    ?trexc, 5-214  
    ?trsen, 5-216  
    ?trsns, 5-209  
    ?unghr, 5-185  
    ?unmhr, 5-187  
QR factorization  
    ?geqpf, 5-11, 5-14  
    ?geqr, 5-8, 5-48, 5-60, 5-68, 5-71  
    ?orgqr, 5-17, 5-50, 5-52  
    ?ormqr, 5-19  
    ?ungqr, 5-21  
    ?unmqr, 5-23  
singular value decomposition  
    ?bdsqr, 5-94, 5-98  
    ?gbrd, 5-79  
    ?gebrd, 5-76

**L**

LAPACK routines  
condition number estimation  
    ?gbcon, 4-69  
    ?gecon, 4-66  
    ?gtcon, 4-71  
    ?hecon, 4-83  
    ?hpcon, 4-88  
    ?pbcon, 4-77  
    ?pocon, 4-73  
    ?ppcon, 4-75  
    ?ptcon, 4-79  
    ?spcon, 4-86  
    ?sycon, 4-81, 5-154  
    ?tbcon, 4-94  
    ?tpcon, 4-92  
    ?trcon, 4-90  
generalized eigenvalue problems  
    ?hbgst, 5-169  
    ?hegst, 5-160  
    ?hpgst, 5-164  
    ?pbstf, 5-172  
    ?sbgst, 5-166

?orgbr, 5-82  
?ormbr, 5-85  
?ungbr, 5-88  
?unmbr, 5-91  
solution refinement and error estimation  
    ?gbrfs, 4-100  
    ?gerfs, 4-97, 4-103  
    ?herfs, 4-121  
    ?hprfs, 4-127  
    ?pbrfs, 4-112  
    ?porfs, 4-106, 4-115  
    ?pprfs, 4-109  
    ?sprfs, 4-124  
    ?syrfs, 4-118  
    ?tbrfs, 4-135  
    ?tprfs, 4-132  
    ?trrfs, 4-129  
solving linear equations  
    ?gbtrs, 4-37  
    ?getrs, 4-35  
    ?gttrs, 4-39  
    ?hetrs, 4-53  
    ?hptrs, 4-57  
    ?pbtrs, 4-47  
    ?potrs, 4-42  
    ?pptrs, 4-45  
    ?pttrs, 4-49  
    ?sptrs, 4-55  
    ?syrtrs, 4-51  
    ?tbtrs, 4-64  
    ?tptrs, 4-62  
    ?trtrs, 4-59  
Sylvester's equation  
    ?trsyl, 5-221  
symmetric eigenvalue problems  
    ?hbevd, 5-350  
    ?hbtrd, 5-130  
    ?heevd, 5-303  
    ?hetrd, 5-111  
    ?hpevd, 5-331  
    ?hpdrv, 5-122  
    ?opgtr, 5-119  
    ?opmtr, 5-120  
    ?orgtr, 5-107  
    ?ormtr, 5-109  
    ?pteqr, 5-146  
    ?sbevd, 5-347  
    ?sbtrd, 5-128  
    ?spevd, 5-328  
    ?sptrd, 5-117  
    ?stebz, 5-141, 5-149  
    ?stein, 5-152  
    ?steqr, 5-134, 5-137  
    ?sterf, 5-132  
    ?stevd, 5-364  
    ?syevd, 5-300  
    ?sytrd, 5-105  
    ?ungtr, 5-113  
    ?unmtr, 5-115  
    ?upgtr, 5-124  
    ?upmtr, 5-125  
triangular factorization  
    ?gbtrf, 4-11  
    ?getrf, 4-9  
    ?gttrf, 4-13  
    ?hetrf, 4-26  
    ?hptrf, 4-32  
    ?pbtrf, 4-19  
    ?potrf, 4-15  
    ?pptrf, 4-17  
    ?pttrf, 4-21  
    ?sptrf, 4-29  
    ?syrtrf, 4-23  
leading dimension, A-6  
length. *See* dimension  
linear combination of vectors, 2-6  
linear equations, solving  
    general matrix, 4-35  
        band storage, 4-37  
    Hermitian matrix, 4-53  
        packed storage, 4-57  
    Hermitian positive-definite matrix, 4-42  
        band storage, 4-47  
        packed storage, 4-45  
    symmetric matrix, 4-51  
        packed storage, 4-55  
    symmetric positive-definite matrix, 4-42

- band storage, 4-47
- packed storage, 4-45
- triangular matrix, 4-59
  - band storage, 4-64
  - packed storage, 4-62
- tridiagonal matrix, 4-39, 4-49
- LQ factorization, 5-6
  - computing the elements of
    - orthogonal matrix Q, 5-28, 5-38, 5-40
    - unitary matrix Q, 5-32
- LU factorization, 4-9
  - band matrix, 4-11
  - tridiagonal matrix, 4-13
- M**
  - matrix arguments, A-4
    - column-major ordering, A-2, A-6
    - example, A-7
    - leading dimension, A-6
    - number of columns, A-6
    - number of rows, A-6
    - transposition parameter, A-6
  - matrix equation
    - $AX = B$ , 2-110, 4-3, 4-34
  - matrix one-dimensional substructures, A-2
  - matrix-matrix operation
    - product
      - general matrix, 2-83
    - rank-2k update
      - Hermitian matrix, 2-92
      - symmetric matrix, 2-103
    - rank-n update
      - Hermitian matrix, 2-89
      - symmetric matrix, 2-100
    - scalar-matrix-matrix product
      - Hermitian matrix, 2-86
      - symmetric matrix, 2-96
      - triangular matrix, 2-107
  - matrix-vector operation
    - product, 2-24, 2-27
      - Hermitian matrix, 2-38
- band storage, 2-35
- packed storage, 2-44
- symmetric matrix, 2-60
  - band storage, 2-51
  - packed storage, 2-54
- triangular matrix, 2-77
  - band storage, 2-66
  - packed storage, 2-72
- rank-1 update, 2-30, 2-31, 2-33
  - Hermitian matrix, 2-40
  - packed storage, 2-47
- symmetric matrix, 2-62
  - packed storage, 2-56
- rank-2 update
  - Hermitian matrix, 2-42
  - packed storage, 2-49
- symmetric matrix, 2-64
  - packed storage, 2-58
- N**
  - naming conventions, 1-6
    - BLAS, 2-2
    - LAPACK, 4-2, 5-4
    - Sparse BLAS, 2-115
    - VML, 6-2
- one-dimensional FFTs, 3-1
  - complex sequence, 3-9, 3-11, 3-14, 3-16
  - complex-to-complex, 3-3
  - complex-to-real, 3-11
  - computing a forward FFT, real input data, 3-8, 3-10
  - computing a forward or inverse FFT of a complex vector, 3-4, 3-5
  - groups, 3-2
  - performing an inverse FFT, complex input data, 3-13, 3-15
- O**

real-to-complex, 3-6  
storage effects, 3-7, 3-12  
orthogonal matrix, 5-74, 5-101, 5-174, 5-224

**P**

---

packed storage scheme, A-4  
parameters  
    for a Givens rotation, 2-14  
    modified Givens transformation, 2-17  
platforms supported, 1-4  
points  
    rotation in the modified plane, 2-15  
    rotation in the plane, 2-12  
positive-definite matrix  
    generalized eigenvalue problems, 5-158  
product  
    *See also* dot product  
matrix-vector  
    general matrix, 2-27  
        band storage, 2-24  
    Hermitian matrix, 2-38  
        band storage, 2-35  
        packed storage, 2-44  
    symmetric matrix, 2-60  
        band storage, 2-51  
        packed storage, 2-54  
triangular matrix, 2-77  
    band storage, 2-66  
    packed storage, 2-72  
scalar-matrix  
    general matrix, 2-83  
    Hermitian matrix, 2-86  
scalar-matrix-matrix  
    general matrix, 2-83  
    Hermitian matrix, 2-86  
    symmetric matrix, 2-96  
    triangular matrix, 2-107  
vector-scalar, 2-18

**Q**

---

QR factorization, 5-6  
computing the elements of  
    orthogonal matrix Q, 5-17, 5-50, 5-52  
    unitary matrix Q, 5-21  
with pivoting, 5-11, 5-14  
quasi-triangular matrix, 5-174, 5-224

**R**

---

rank-1 update  
    conjugated, general matrix, 2-31  
    general matrix, 2-30  
    Hermitian matrix, 2-40  
        packed storage, 2-47  
    symmetric matrix, 2-62  
        packed storage, 2-56  
    unconjugated, general matrix, 2-33  
rank-2 update  
    Hermitian matrix, 2-42  
        packed storage, 2-49  
    symmetric matrix, 2-64  
        packed storage, 2-58  
rank-2k update  
    Hermitian matrix, 2-92  
    symmetric matrix, 2-103  
rank-n update  
    Hermitian matrix, 2-89  
    symmetric matrix, 2-100  
real-to-complex one-dimensional FFTs, 3-6  
real-to-complex two-dimensional FFTs, 3-21  
reducing generalized eigenvalue problems,  
    5-158  
refining solutions of linear equations  
    band matrix, 4-100  
    general matrix, 4-97, 4-103  
    Hermitian matrix, 4-121  
        packed storage, 4-127  
    Hermitian positive-definite matrix, 4-106,  
        4-115  
        band storage, 4-112

packed storage, 4-109  
    symmetric matrix, 4-118  
        packed storage, 4-124  
    symmetric positive-definite matrix, 4-106,  
        4-115  
        band storage, 4-112  
        packed storage, 4-109  
rotation  
    of points in the modified plane, 2-15  
    of points in the plane, 2-12  
    of sparse vectors, 2-123  
parameters for a Givens rotation, 2-14  
parameters of modified Givens  
    transformation, 2-17  
routine name conventions  
    BLAS, 2-2  
    Sparse BLAS, 2-115

**S**

---

scalar-matrix product, 2-83, 2-86, 2-96  
scalar-matrix-matrix product, 2-86  
    general matrix, 2-83  
    symmetric matrix, 2-96  
    triangular matrix, 2-107  
scattering compressed sparse vector's elements  
    into full storage form, 2-124  
singular value decomposition, 5-74  
    *See also* LAPACK routines, singular value  
    decomposition  
smallest absolute value of a vector element, 2-22  
solving linear equations. *See* linear equations  
Sparse BLAS, 2-114  
    data types, 2-115  
    naming conventions, 2-115  
Sparse BLAS routines and functions, 2-115  
    ?axpyi, 2-116  
    ?dotci, 2-119  
    ?doti, 2-118  
    ?dotui, 2-120  
    ?gthr, 2-121  
    ?gthrz, 2-122

    ?roti, 2-123  
    ?sctr, 2-124  
sparse vectors, 2-114  
    adding and scaling, 2-116  
    complex dot product, conjugated, 2-119  
    complex dot product, unconjugated, 2-120  
    compressed form, 2-114  
    converting to compressed form, 2-121, 2-122  
    converting to full-storage form, 2-124  
    full-storage form, 2-114  
    Givens rotation, 2-123  
    norm, 2-116  
    passed to BLAS level 1 routines, 2-116  
    real dot product, 2-118  
    scaling, 2-116  
split Cholesky factorization (band matrices),  
    5-172  
stride. *See* increment  
sum  
    of magnitudes of the vector elements, 2-5  
    of sparse vector and full-storage vector,  
        2-116  
    of vectors, 2-6  
SVD (singular value decomposition), 5-74  
swapping vectors, 2-20  
Sylvester's equation, 5-221  
symmetric matrix, 5-101, 5-157  
Bunch-Kaufman factorization, 4-23  
    packed storage, 4-29  
estimating the condition number, 4-81,  
    5-154  
    packed storage, 4-86  
generalized eigenvalue problems, 5-157  
inverting the matrix, 4-144  
    packed storage, 4-148  
matrix-vector product, 2-60  
    band storage, 2-51  
    packed storage, 2-54  
rank-1 update, 2-62  
    packed storage, 2-56  
rank-2 update, 2-64  
    packed storage, 2-58

rank-2k update, 2-103  
rank-n update, 2-100  
scalar-matrix-matrix product, 2-96  
solving systems of linear equations, 4-51  
    packed storage, 4-55  
symmetric positive-definite matrix  
    Cholesky factorization, 4-15  
        band storage, 4-19  
        packed storage, 4-17  
    estimating the condition number, 4-73  
        band storage, 4-77  
        packed storage, 4-75  
        tridiagonal matrix, 4-79  
    inverting the matrix, 4-140  
        packed storage, 4-142  
    solving systems of linear equations, 4-42  
        band storage, 4-47  
        packed storage, 4-45  
system of linear equations  
    with a triangular matrix, 2-79  
        band storage, 2-69  
        packed storage, 2-75  
systems of linear equations. *See* linear equations

**T**

---

transposition parameter, A-6  
triangular factorization  
    band matrix, 4-11  
    general matrix, 4-9  
    Hermitian matrix, 4-26  
        packed storage, 4-32  
    Hermitian positive-definite matrix, 4-15  
        band storage, 4-19  
        packed storage, 4-17  
        tridiagonal matrix, 4-21  
    symmetric matrix, 4-23  
        packed storage, 4-29  
    symmetric positive-definite matrix, 4-15  
        band storage, 4-19  
        packed storage, 4-17  
        tridiagonal matrix, 4-21  
    tridiagonal matrix, 4-13  
triangular matrix, 5-174, 5-224  
    estimating the condition number, 4-90  
        band storage, 4-94

    packed storage, 4-92  
    inverting the matrix, 4-152  
        packed storage, 4-153  
    matrix-vector product, 2-77  
        band storage, 2-66  
        packed storage, 2-72  
    scalar-matrix-matrix product, 2-107  
    solving systems of linear equations, 2-79,  
        4-59  
        band storage, 2-69, 4-64  
        packed storage, 2-75, 4-62  
tridiagonal matrix, 5-101  
    estimating the condition number, 4-71  
    solving systems of linear equations, 4-39,  
        4-49  
two-dimensional FFTs, 3-17  
    complex-to-complex, 3-18  
    complex-to-real, 3-27  
    computing a forward FFT, real input data,  
        3-22, 3-24  
    computing a forward or inverse FFT, 3-19,  
        3-20  
    computing an inverse FFT, complex input  
        data, 3-28, 3-29  
    data storage types, 3-18  
    data structure requirements, 3-18  
    equations, 3-18  
    groups, 3-17  
    real-to-complex, 3-21

**U**

---

unitary matrix, 5-74, 5-101, 5-174, 5-224  
updating  
    rank-1  
        general matrix, 2-30  
        Hermitian matrix, 2-40  
            packed storage, 2-47  
        symmetric matrix, 2-62  
            packed storage, 2-56  
    rank-1, conjugated  
        general matrix, 2-31  
    rank-1, unconjugated  
        general matrix, 2-33  
    rank-2

- Hermitian matrix, 2-42
  - packed storage, 2-49
- symmetric matrix, 2-64
  - packed storage, 2-58
- rank-2k
  - Hermitian matrix, 2-92
  - symmetric matrix, 2-103
- rank-n
  - Hermitian matrix, 2-89
  - symmetric matrix, 2-100
- upper Hessenberg matrix, 5-174, 5-224
  
- V**
- vector arguments, A-1
  - array dimension, A-1
  - default, A-2
  - examples, A-2
  - increment, A-1
  - length, A-1
  - matrix one-dimensional substructures, A-2
  - sparse vector, 2-114
- vector indexing, 6-6
- vector mathematical functions, 6-8
  - cosine, 6-22
  - cube root, 6-15
  - denary logarithm, 6-21
  - division, 6-11
  - exponential, 6-18
  - four-quadrant arctangent, 6-29
  - hyperbolic cosine, 6-30
  - hyperbolic sine, 6-31
  - hyperbolic tangent, 6-33
  - inverse cosine, 6-26
  - inverse cube root, 6-16
  - inverse hyperbolic cosine, 6-34
  - inverse hyperbolic sine, 6-35
  - inverse hyperbolic tangent, 6-36
  - inverse sine, 6-27
  - inverse square root, 6-13
  - inverse tangent, 6-28
  - inversion, 6-10
- natural logarithm, 6-20
- power, 6-17
- sine, 6-23
- sine and cosine, 6-24
- square root, 6-12
- tangent, 6-25
- vector pack function, 6-37
- vector unpack function, 6-39
- vectors
  - adding magnitudes of vector elements, 2-5
  - copying, 2-7
  - dot product
    - complex vectors, 2-10
    - complex vectors, conjugated, 2-9
    - real vectors, 2-8
- element with the largest absolute value, 2-21
- element with the smallest absolute value, 2-22
- Euclidean norm, 2-11
- Givens rotation, 2-14
- linear combination of vectors, 2-6
- modified Givens transformation parameters, 2-17
- rotation of points, 2-12
- rotation of points in the modified plane, 2-15
- sparse vectors, 2-115
- sum of vectors, 2-6
- swapping, 2-20
- vector-scalar product, 2-18
- vector-scalar product, 2-18
  - sparse vectors, 2-116
- VML, 6-1
- VML functions
  - mathematical functions
    - Acos, 6-26
    - Acosh, 6-34
    - Asin, 6-27
    - Asinh, 6-35
    - Atan, 6-28
    - Atan2, 6-29
    - Atanh, 6-36
    - Cbrt, 6-15
    - Cos, 6-22

Cosh, 6-30  
Div, 6-11  
Exp, 6-18  
Inv, 6-10  
InvCbrt, 6-16  
InvSqrt, 6-13  
Ln, 6-20  
Log10, 6-21  
Pow, 6-17  
Sin, 6-23  
SinCos, 6-24  
Sinh, 6-31  
Sqrt, 6-12  
Tan, 6-25  
Tanh, 6-33  
pack/unpack functions  
    Pack, 6-37  
    Unpack, 6-39  
service functions  
    ClearErrorCallBack, 6-52  
    ClearErrStatus, 6-48  
    GetErrorCallBack, 6-51  
    GetErrStatus, 6-48  
    GetMode, 6-45  
    SetErrorCallBack, 6-49  
    SetErrStatus, 6-46  
    SetMode, 6-42

---

**X**

XERBLA, error reporting routine, 2-1