VSIPL 1.3 API

David A. Schwartz
HRL Laboratories, LLC
Randall R. Judd
Space and Naval Warfare Systems Center, San Diego
William J. Harrod
Silicon Graphics Inc./Cray Research
Dwight P. Manley
Compaq Computer Corp./Digital Equipment Corp.

http://www.vsipl.org

As Approved by the VSIPL Forum

January 31, 2008

©1999-2008 Georgia Tech Research Corporation, all rights reserved.
A non-exclusive, non-royalty bearing license is hereby granted to all persons to copy, modify, distribute and produce derivative works for any purpose, provided that this copyright notice and following disclaimer appear on all copies: THIS LICENSE INCLUDES NO WARRANTIES, EXPRESSED OR IMPLIED, WHETHER ORAL OR WRITTEN, WITH RESPECT TO THE SOFTWARE OR OTHER MATERIAL INCLUDING, BUT NOT LIMITED TO, ANY IMPLIED WARRANTIES OF MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE, OR ARISING FROM A COURSE OF PERFORMANCE OR DEALING, OR FROM USAGE OR TRADE, OR OF NON-INFRINGEMENT OF ANY PATENTS OF THIRD PARTIES. THE INFORMATION IN THIS DOCUMENT SHOULD NOT BE CONSTRUED AS A COMMITMENT OF DEVELOPMENT BY ANY OF THE ABOVE PARTIES.

This material is based in part upon work supported by Ft. Huachuca/DARPA under contract No. DABT63-96-C-0060. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of Ft. Huachuca/DARPA.

The US Government has a license under these copyrights, and this material may be reproduced by or for the US Government.

ACKNOWLEDGMENT

The VSIPL Forum would like to thank the participation of the following organizations and individuals (**bold** indicates a primary contributor):

Alacron	Tom Charlesworth Leo Mirkin Joe Sgro
CSP Inc, Multicomputer Division	**Sharon Sacco** Jim Waggett
Compaq Computer Corp. (Digital Equipment Corp.)	**Dwight Manley**
Defense Advanced Research Projects Agency	**Jose Munoz** **Col. Mark Swinson**
DNA Computing Solutions, Inc.	Win Smith
Georgia Institute of Technology	**Dan Campbell** **Tom Egolf** **Randy Janka** **Mark Richards**
Hughes Electronic	**E. Robert Tisdale**
HRL Laboratories, LLC	**David Schwartz**
Khoral Research	Joe Fogler **Tom Robey** **Jeremy Worley**
LNK Corp.	**Carl Stevens**
Lockheed Martin	Billy Chin Mark Law **Rick Pancoast**
Mercury Computer Systems	Jon Green Barry Isenstein **Jamie Kenny** **Arlan Pool**
Mississippi State University	**Purushotham (Puri) Bangalore** **Avi Purkayastha** Tony Skjellum
MIT Lincoln Laboratory	Bob Bond **James Lebak**
MITRE	Mike Butler Richard Games Brian Sroka
MPI Software Technology, Inc.	Clay Taylor
Northrop Grumman	Marc Campbell
Ohio State University	Stan Ahalt
ORINCON	John Carbone **Robert Redfield** **Tom Johnson**
Raytheon Systems Company	Frank Miville

	Gary Corbitt
Science and Technology Associates	**Jon Hiller** Ed Monachino
Silicon Graphics Inc.	**Bill Harrod**
SKY Computers	**Dimitris Christodoulou** **Joe Germann** Richard Jaenicke **Michael LeBlanc** Leo Mirkin **Steve Paavola** **Sharon Sacco**
Texas Instruments (Spectron)	David Friedland
Texas Instruments	Jacqueline Brenner **Dave Syiek**
Texas Memory Systems	**John Marsh** Brian Shoemaker Jonathon Brauer
U.S. Air Force	Ralph Kohler Rich Linderman
Univ. of San Diego Supercomputing Center	Tim Kaiser Stuart Johnson
U.S. Navy	Kris Briggs **Keith Bromley** **Chuck Browne** **Jim Broughton** **Rob Ginn** **Clair Guthrie** **Roger Hillson** **Randall Judd** **Tony Lee** **Tim Singleton**

Special thanks to DARPA's ITO Embedded High Performance Computing program and the Navy's TASP (PMS411) program for partial support of the VSIPL Forum.

TABLE OF CONTENTS

VSIPL 1.3 API i
ACKNOWLEDGMENT iii
TABLE OF CONTENTS v
GLOSSARY xvii
NOTATION NOTES xx
CODE STYLE NOTES xxi
VSIPL INTRODUCTION 1
Introduction to the VSIPL Standard 1
Platform Requirements 1
VSIPL Functionality 1
VSIPL Objects 2
Other Features of the Specification 2
Basic VSIPL Concepts 3
General Library Design Principles 3
Memory Management 3
Structure of a VSIPL application 6
Introduction to VSIPL Data Types 7
VSIPL Naming Conventions 7
Non-Standard Scalar Data Types 8
Data Array Layout 8
Summary of Scalar and View Types 9
Description of the API Specification Pages 11
Function Name 11
SUMMARY OF VSIPL TYPES 13
Type - Introduction 13
Scalar 15
Real Scalar 15
Complex Scalar 16
Boolean 16
Index 16
Offset, Stride, Length 16
Random Number Generator 16
Block Object 17
Vector View 18
Real Vector View Object 18
Complex Vector View 19
Matrix View 20
Real Matrix View Object 20
Complex Matrix View Object 21
Tensor View 22

Real Tensor View Object....22
Complex Tensor View Object....23
Signal Processing....25
Signal Processing – FFT....25
Signal Processing – Filtering....27
Signal Processing – Convolution....28
Signal Processing – Correlation....28
Signal Processing – Histogram....29
Linear Algebra....30
Solvers....30
SUPPORT FUNCTIONS....33
Support - Introduction....33
Library Initialization and Finalization....33
Object Creation and Destruction....33
Complex Blocks and Views....35
Real Views of Real or Imaginary Parts of Complex Views....35
Examples....35
Initialization/Finalization Operations....36
vsip_init Initialize the Library....37
vsip_finalize Terminate the Library....39
Array and Block Object Functions....40
vsip_*d*blockadmit_*p* Block Admit....41
vsip_blockbind_*p* Memory Block Bind....43
vsip_cblockbind_*p* Complex Memory Block Bind....45
vsip_*d*blockcreate_*p* Memory Block Create....47
vsip_*d*blockdestroy_*p* Memory Block Destroy....49
vsip_blockfind_*p* Memory Block Find....51
vsip_cblockfind_*p* Memory Complex Block Find....52
vsip_blockrebind_*p* Block Rebind....53
vsip_cblockrebind_*p* Complex Block Rebind....55
vsip_blockrelease_*p* Block Release....57
vsip_cblockrelease_*p* Complex Block Release....59
vsip_complete Complete Deferred Execution....63
vsip_cstorage_*p* Complex Storage....64
Vector View Object Functions....66
vsip_*d*valldestroy_*p* Destroy Vector and Block....67
vsip_*d*vbind_*p* Create and Bind a Vector View....69
vsip_*d*vcloneview_*p* Create Vector View Clone....71
vsip_*d*vcreate_*p* Create Vector....73
vsip_*d*vdestroy_*p* Destroy Vector View....75
vsip_*d*vget_*p* Vector Get Element....77
vsip_*d*vgetattrib_*p* Vector Get View Attributes....78
vsip_*d*vgetblock_*p* Vector Get Block....80
vsip_*d*vgetlength_*p* Vector Get Length....81

vsip_*d*vgetoffset_*p* Vector Get Offset....82
vsip_*d*vgetstride_*p* Vector Get Stride....83
vsip_vimagview_*p* Create Imaginary Vector View....84
vsip_*d*vput_*p* Vector Put Element....86
vsip_*d*vputattrib_*p* Put Vector View Attributes....88
vsip_*d*vputlength_*p* Vector Put Length....90
vsip_*d*vputoffset_*p* Vector Put Offset....91
vsip_*d*vputstride_*p* Vector Put Stride....92
vsip_vrealview_*p* Create Real Vector View....93
vsip_*d*vsubview_*p* Create Subview Vector View....95
Matrix View Object Functions....97
vsip_*d*malldestroy_*p* Destroy Matrix and Block....99
vsip_*d*mbind_*p* Create and Bind a Matrix View....101
vsip_*d*mcloneview_*p* Create Matrix View Clone....103
vsip_*d*mcolview_*p* Create Column-View Matrix View....104
vsip_*d*mcreate_*p* Create Matrix....105
vsip_*d*mdestroy_*p* Destroy Matrix View....107
vsip_*d*mdiagview_*p* Create Matrix Diagonal View....108
vsip_*d*mget_*p* Matrix Get Element....110
vsip_*d*mgetattrib_*p* Matrix Get View Attributes....111
vsip_*d*mgetblock_*p* Matrix Get Block....113
vsip_*d*mgetcollength_*p* Matrix Get Column Length....114
vsip_*d*mgetcolstride_*p* Matrix Get Column Stride....115
vsip_*d*mgetoffset_*p* Matrix Get Offset....116
vsip_*d*mgetrowlength_*p* Matrix Get Row Length....117
vsip_*d*mgetrowstride_*p* Matrix Get Row Stride....118
vsip_mimagview_*p* Create Imaginary Matrix View....119
vsip_*d*mput_*p* Matrix Put Element....121
vsip_*d*mputattrib_*p* Matrix Put View Attributes....123
vsip_*d*mputcollength_*p* Matrix Put Column Length....125
vsip_*d*mputcolstride_*p* Matrix Put Column Stride....126
vsip_*d*mputoffset_*p* Matrix Put Offset....127
vsip_*d*mputrowlength_*p* Matrix Put Row Length....128
vsip_*d*mputrowstride_*p* Matrix Put Row Stride....129
vsip_mrealview_*p* Create Real Matrix View....130
vsip_*d*mrowview_*p* Create Matrix Row View....132
vsip_*d*msubview_*p* Create Subview Matrix View....133
vsip_*d*mtransview_*p* Create Matrix Transposed View....135
Tensor Views....136
vsip_*d*talldestroy_*p* Destroy Tensor And Block....138
vsip_*d*tbind_*p* Create and Bind a Tensor View....139
vsip_*d*tcloneview_*p* Create Tensor View Clone....141
vsip_*d*tcreate_*p* Create Tensor....142
vsip_*d*tdestroy_*p* Destroy Tensor View....144
vsip_*d*tget_*p* Tensor Get Element....145
vsip_*d*tgetattrib_*p* Tensor Get View Attributes....147

vsip_*d*tgetblock_*p* Tensor Get Block 149
vsip_*d*tgetoffset_*p* Tensor Get Offset 150
vsip_*d*tgetxlength_*p* Tensor Get X Length 151
vsip_*d*tgetxstride_*p* Tensor Get X Stride 152
vsip_*d*tgetylength_*p* Tensor Get Y Length 153
vsip_*d*tgetystride_*p* Tensor Get Y Stride 154
vsip_*d*tgetzlength_*p* Tensor Get Z Length 155
vsip_*d*tgetzstride_*p* Tensor Get Z Stride 156
vsip_timagview_*p* Create Imaginary Tensor View 157
vsip_*d*tmatrixview_*p* Create Tensor Matrix View 159
vsip_*d*tput_*p* Tensor Put Element 161
vsip_*d*tputattrib_*p* Tensor Put View Attributes 163
vsip_*d*tputoffset_*p* Tensor Put Offset 165
vsip_*d*tputxlength_*p* Tensor Put X Length 166
vsip_*d*tputxstride_*p* Tensor Put X Stride 167
vsip_*d*tputylength_*p* Tensor Put Y Length 168
vsip_*d*tputystride_*p* Tensor Put Row Stride 169
vsip_*d*tputzlength_*p* Tensor Put Z Length 170
vsip_*d*tputzstride_*p* Tensor Put Z Stride 171
vsip_trealview_*p* Create Real Tensor View 172
vsip_*d*tsubview_*p* Create Subview Tensor View 174
vsip_*d*ttransview_*p* Create Tensor Transposed View 176
vsip_*d*tvectview_*p* Create Tensor Vector View 178
SCALAR FUNCTIONS 181
Introduction To Scalar Functions 181
Domain And Range Errors 181
Notes To Implementors 181
Real Scalar Functions 182
Complex Scalar Functions 183
Real Scalar Functions 184
vsip_acos_*p* Scalar Arccosine 185
vsip_asin_*p* Scalar Arcsine 186
vsip_atan_*p* Scalar Arctangent 187
vsip_atan2_*p* Scalar Arctangent of Two Arguments 188
vsip_ceil_*p* Ceiling 190
vsip_cos_*p* Scalar Cosine 191
vsip_cosh_*p* Scalar Hyperbolic Cosine 192
vsip_exp_*p* Scalar Exponential 193
vsip_exp10_*p* Scalar Exponential Base 10 194
vsip_floor_*p* Floor 195
vsip_fmod_*p* Modulo 196
vsip_hypot_*p* Scalar Hypotenuse 197
vsip_log_*p* Scalar Log 198
vsip_log10_*p* Scalar Log Base Ten 199
vsip_mag_*p* Scalar Magnitude (Abs) 200
vsip_max_*p* Scalar Maximum 201

vsip_min_*p* Scalar Minimum....202
vsip_pow_*p* Scalar Power....203
vsip_rsqrt_*p* Scalar Reciprocal Square Root....204
vsip_sin_*p* Scalar Sine....205
vsip_sinh_*p* Scalar Hyperbolic Sine....206
vsip_sqrt_*p* Scalar Square Root....207
vsip_tan_*p* Scalar Tangent....208
vsip_tanh_*p* Scalar Hyperbolic Tangent....209
Complex Scalar Functions....210
vsip_arg_*p* Complex Scalar Argument....212
vsip_cadd_*p* Complex Scalar Add....213
vsip_cdiv_*p* Complex Scalar Divide....214
vsip_cexp_*p* Complex Scalar Exponential....215
vsip_cjmul_*p* Complex Conjugate Scalar Multiply....216
vsip_clog_*p* Complex Scalar Log....217
vsip_cmag_*p* Complex Scalar Magnitude....218
vsip_cmagsq_*p* Complex Scalar Magnitude Squared....219
vsip_cmplx_*p* Complex Scalar....220
vsip_cmul_*p* Complex Scalar Multiply....221
vsip_cneg_*p* Complex Scalar Negate....222
vsip_conj_*p* Complex Scalar Conjugate....223
vsip_crecip_*p* Complex Scalar Reciprocal....224
vsip_csqrt_*p* Complex Scalar Square Root....225
vsip_csub_*p* Complex Scalar Subtract....226
vsip_imag_*p* Complex Scalar Imaginary....228
vsip_polar_*p* Complex Scalar Polar....229
vsip_real_*p* Complex Scalar Real....230
vsip_rect_*p* Complex Scalar Rectangular....231
Index Scalar Functions....232
vsip_matindex Matrix Index....233
vsip_mcolindex Matrix Column Index....234
vsip_mrowindex Matrix Row Index....235
vsip_tenindex Tensor Index....236
vsip_txindex Tensor X Index....237
vsip_tyindex Tensor Y Index....238
vsip_tzindex Tensor Z Index....239
RANDOM NUMBER GENERATION....241
Introduction....241
Random Numbers....241
VSIPL Random Number Generator Functions....241
Sample Implementation....244
Random Number Functions....248
vsip_randcreate Create Random State....249
vsip_randdestroy Destroy Random State....251
vsip_*ds*randu_*p* Uniform Random Numbers....252

vsip_*ds*randn_*p* Gaussian Random Numbers 254
VECTOR & ELEMENTWISE OPERATIONS....257
Vector & Elementwise - Introduction....257
Name Space 257
In-Place Functionality 261
Example Programs 265
Elementary Math Functions....266
vsip_*s*acos_*p* Vector/Matrix Arccosine....267
vsip_*s*asin_*p* Vector/Matrix Arcsine....269
vsip_*s*atan_*p* Vector/Matrix Arctangent....271
vsip_*s*atan2_*p* Vector/Matrix Arctangent of Two Arguments....273
vsip_*s*cos_*p* Vector/Matrix Cosine 276
vsip_*s*cosh_*p* Vector/Matrix Hyperbolic Cosine 278
vsip_*ds*exp_*p* Vector/Matrix Exponential 280
vsip_*s*exp10_*p* Vector/Matrix Exponential Base 10....282
vsip_*ds*log_*p* Vector/Matrix Log 284
vsip_*s*log10_*p* Vector/Matrix Log Base 10....286
vsip_*s*sin_*p* Vector/Matrix Sine....288
vsip_*s*sinh_*p* Vector/Matrix Hyperbolic Sine....290
vsip_*ds*sqrt_*p* Vector/Matrix Square Root 292
vsip_*s*tan_*p* Vector/Matrix Tangent....295
vsip_*s*tanh_*p* Vector/Matrix Hyperbolic Tangent....297
Unary Operations 299
vsip_*s*arg_*p* Vector/Matrix Argument....300
vsip_*s*ceil_*p*_*p* Vector/Matrix Ceiling 303
vsip_c*s*conj_*p* Vector/Matrix Conjugate 304
vsip_*ds*cumsum_*p* Cumulative Sum 306
vsip_*s*euler_*p* Vector/Matrix Euler 308
vsip_*s*floor_*p*_*p* Vector/Matrix Floor....310
vsip_*ds*mag_*p* Vector/Matrix Magnitude....311
vsip_*s*cmagsq_*p* Vector/Matrix Complex Magnitude Squared....313
vsip_*ds*meanval_*p* Vector/Matrix Mean Value 315
vsip_*ds*meansqval_*p* Vector/Matrix Mean Square Value 317
vsip_*d*vmodulate_*p* Vector Modulate 319
vsip_*ds*neg_*p* Vector/Matrix Negate....321
vsip_*ds*recip_*p* Vector/Matrix Reciprocal....324
vsip_*s*round_*p*_*p* Vector/Matrix Round 327
vsip_*ds*rsqrt_*p* Vector/Matrix Reciprocal Square Root....328
vsip_*ds*sq_*p* Vector/Matrix Square 330
vsip_*ds*sumval_*p* Vector/Matrix Sum Value....332
vsip_*ds*sumsqval_*p* Vector/Matrix Sum of Squares Value 334
Binary Operations 335
vsip_*ds*add_*p* Vector/Matrix Add 336
vsip_*ds*add_*p* Scalar Vector/Matrix Add 339
vsip_*ds*div_*p* Vector/Matrix Divide....342

vsip_*dss*div_*p* Scalar Vector/Matrix Divide....345
vsip_*ds*sdiv_*p* Vector/Matrix Scalar Divide....347
vsip_*ds*expoavg_*p* Vector/Matrix Exponential Average....349
vsip_*s*hypot_*p* Vector/Matrix Hypotenuse....351
vsip_c*s*jmul_*p* Vector/Matrix Conjugate Multiply (Elementwise)....353
vsip_*ds*mul_*p* Vector/Matrix Multiply (Elementwise)....355
vsip_*dss*mul_*p* Scalar Vector/Matrix Multiply....358
vsip_*dvd*mmul_*p* Vector-Matrix Multiply (Elementwise)....361
vsip_*ds*sub_*p* Vector/Matrix Subtract....363
vsip_*dss*sub_*p* Scalar Vector/Matrix Subtract....366
Ternary Operations....368
vsip_*d*vam_*p* Vector Add and Multiply....369
vsip_*d*vma_*p* Vector Multiply and Add....371
vsip_*d*vmsa_*p* Vector Multiply, Scalar Add....373
vsip_*d*vmsb_*p* Vector Multiply and Subtract....375
vsip_*d*vsam_*p* Vector Scalar Add, Vector Multiply....377
vsip_*d*vsbm_*p* Vector Subtract and Multiply....379
vsip_*d*vsma_*p* Vector Scalar Multiply, Vector Add....381
vsip_*d*vsmsa_*p* Vector Scalar Multiply, Scalar Add....383
Logical Operations....385
vsip_*s*alltrue_bl Vector/Matrix All True....386
vsip_*s*anytrue_bl Vector/Matrix Any True....388
vsip_*ds*leq_*p* Vector/Matrix Logical Equal....390
vsip_*dss*leq_*p* Scalar Vector/Matrix Logical Equal....393
vsip_*s*lge_*p* Vector/Matrix Logical Greater Than or Equal....395
vsip_*ss*lge_*p* Scalar Vector/Matrix Logical Greater Than or Equal....397
vsip_*s*lgt_*p* Vector/Matrix Logical Greater Than....399
vsip_*ss*lgt_*p* Scalar Vector/Matrix Logical Greater Than....401
vsip_*s*lle_*p* Vector/Matrix Logical Less Than or Equal....403
vsip_*ss*lle_*p* Scalar Vector/Matrix Logical Less Than or Equal....405
vsip_*s*llt_*p* Vector/Matrix Logical Less Than....407
vsip_*ss*llt_*p* Scalar Vector/Matrix Logical Less Than....409
vsip_*ds*lne_*p* Vector/Matrix Logical Not Equal....411
vsip_*ds*lne_*p* Scalar Vector/Matrix Logical Not Equal....413
Selection Operations....414
vsip_*s*clip_*p* Vector/Matrix Clip....415
vsip_*d*vfirst_*p* Find First Vector Index....418
vsip_*s*invclip_*p* Vector/Matrix Inverted Clip....420
vsip_*s*indexbool Vector/Matrix Index a Boolean....422
vsip_*s*max_*p* Vector/Matrix Maximum....423
vsip_*s*maxmg_*p* Vector/Matrix Maximum Magnitude....425
vsip_*s*cmaxmgsq_*p* Vector/Matrix Complex Max Magnitude Squared....427
vsip_*s*cmaxmgsqval_*p* Vector/Matrix Complex Max Mag Squared Value....429
vsip_*s*maxmgval_*p* Vector/Matrix Maximum Magnitude Value....432
vsip_*s*maxval_*p* Vector/Matrix Maximum Value....434
vsip_*s*min_*p* Vector/Matrix Minimum....436

vsip_*s*minmg_*p* Vector/Matrix Minimum Magnitude 437
vsip_*s*cminmgsq_*p* Vector/Matrix Complex Min Magnitude Squared....438
vsip_*s*cminmgsqval_*p* Vector/Matrix Complex Min Mag Squared Value 439
vsip_*s*minmgval_*p* Vector/Matrix Minimum Magnitude Value 440
vsip_*s*minval_*p* Vector/Matrix Minimum Value....441
Bitwise and Boolean Logical Operators....442
vsip_*s*and_*p* Vector/Matrix AND 443
vsip_*s*not_*p* Vector/Matrix NOT 446
vsip_*s*or_*p* Vector/Matrix OR....448
vsip_*s*xor_*p* Vector/Matrix Exclusive OR....450
Element Generation and Copy....453
vsip_*ds*copy_*p*_*p* Vector/Matrix/Tensor Copy 454
vsip_*ds*copyto_user_*p* Vector/Matrix Copy To User Memory....458
vsip_*ds*copyfrom_user_*p* Vector/Matrix Copy From User Memory....461
vsip_*ds*fill_*p* Vector/Matrix Fill 464
vsip_vramp_*p* Vector Ramp 465
Manipulation Operations....466
vsip_*s*cmplx_*p* Vector/Matrix Complex 467
vsip_*ds*gather_*p* Vector/Matrix Gather 468
vsip_*d*tgather_*p* Tensor Gather....471
vsip_*s*imag_*p* Vector/Matrix Imaginary....473
vsip_*s*polar_*p* Vector/Matrix Polar 475
vsip_*s*real_*p* Vector/Matrix Real 477
vsip_*s*rect_*p* Vector/Matrix Rectangular....478
vsip_*ds*scatter_*p* Vector/Matrix Scatter....480
vsip_*d*tscatter_*p* Tensor Scatter 483
vsip_*ds*swap_*p* Vector/Matrix Swap....485
User-Specified By Element Functions....487
vsip_*s*binary_*p* User-Specified Binary Function 488
vsip_*s*bool_*p* User-Specified Boolean Binary Function 490
vsip_*s*mary_*p* User-Specified M-ary Function....492
vsip_*s*nary_*p* User-Specified Stream Function....496
vsip_*s*serialmary_*p* User-Specified Serial M-ary Function....498
vsip_*s*unary_*p* User-Specified Unary Function....501
SIGNAL PROCESSING FUNCTIONS....503
VSIPL Signal Processing - Introduction....503
VSIPL FFT Routines....503
VSIPL Window Routines 506
VSIPL Filter Routines 507
VSIPL Convolution/Correlation Routines....508
VSIPL Miscellaneous Routines 509
FFT Functions....510
vsip_ccfft*x*_*f* FFT Complex to Complex 511
vsip_crfftop_*f* FFT Complex to Real 514

vsip_rcfftop_*f* FFT Real to Complex 517
vsip_*d*ft*x*_create_*f* Create 1D FFT Object....520
vsip_fft_setwindow_*f* Set a FFT Window....523
vsip_ccfftm*x*_*f* FFT Multiple Complex to Complex....524
vsip_crfftmop_*f* FFT Multiple Complex to Real....527
vsip_rcfftmop_*f* FFT Multiple Real to Complex....529
vsip_*d*fftm*x*_create_*f* Create Multiple FFT Object....531
vsip_fftm_setwindow_*f* Set a Multiple FFT Window....535
vsip_ccfft2d*x*_*f* 2D FFT Complex to Complex....536
vsip_crfft2dop_*f* 2D FFT Complex to Real....538
vsip_rcfft2dop_*f* 2D FFT Real to Complex....540
vsip_*d*fft2d*x*_create_*f* Create 2D FFT Object....542
vsip_ccfft3d*x*_*f* 3D FFT Complex to Complex....545
vsip_crfft3dop_*f* 3D FFT Complex to Real....547
vsip_rcfft3dop_*f* 3D FFT Real to Complex....549
vsip_*d*fft3d*x*_create_*f* Create 3D FFT Object....551
vsip_fft*n*_destroy_*f* Destroy FFT Object....554
vsip_fft*n*_getattr_*f* FFT Get Attributes....555
Convolution/Correlation Functions....558
vsip_*d*conv1d_create_*f* Create 1D Convolution Object....559
vsip_*d*conv1d_destroy_*f* Destroy Conv1D Object....562
vsip_conv1d_getattr_*f* Conv1D Get Attributes....563
vsip_*d*convolve1d_*f* 1D Convolution....565
vsip_conv2d_create_*f* Create 2D Convolution Object....567
vsip_conv2d_destroy_*f* Destroy Conv2d Object....570
vsip_conv2d_getattr_*f* Conv2d Get Attributes....571
vsip_convolve2d_*f* 2D Convolution....573
vsip_*d*corr1d_create_*f* Create 1D Correlation Object....576
vsip_*d*corr1d_destroy_*f* Destroy Corr1D Object....579
vsip_*d*corr1d_getattr_*f* Corr1D Get Attributes....580
vsip_*d*correlate1d_*f* 1D Correlation....581
vsip_*d*corr2d_create_*f* Create 2D Correlation Object....584
vsip_*d*corr2d_destroy_*f* Destroy Corr2d Object....587
vsip_*d*corr2d_getattr_*f* Corr2d Get Attributes....588
vsip_*d*correlate2d_*f* 2D Correlation....589
Window Functions....592
vsip_vcreate_blackman_*f* Create Blackman Window....593
vsip_vcreate_cheby_*f* Create Chebyshev Window....594
vsip_vcreate_hanning_*f* Create Hanning Window....596
vsip_vcreate_kaiser_*f* Create Kaiser Window....598
Filter Functions....600
vsip_*d*fir_create_*f* Create Decimated FIR Filter....601
vsip_*d*fir_destroy_*f* Destroy FIR Filter Object....604
vsip_*d*firflt_*f* Decimated FIR Filter....605
vsip_*d*fir_getattr_*f* FIR Get Attributes....608
vsip_*d*fir_reset_*f* FIR Reset....610

vsip_*d*iir_create_*f* Create Cascaded IIR Filter 611
vsip_*d*iir_destroy_*f* Destroy IIR Filter Object 613
vsip_*d*iirflt_*f* Cascaded IIR Filter 614
vsip_*d*iir_getattr_*f* Get IIR Attributes 615
vsip_*d*iir_reset_*f* IIR Reset 616
Miscellaneous Signal Processing Functions 617
vsip_*s*histo_*p* Histogram 618
vsip_*ds*freqswap_*f* Frequency Swap 620
LINEAR ALGEBRA FUNCTIONS 621
Introduction 621
Matrix and Vector Operations 622
vsip_cmherm_*p* Matrix Hermitian 623
vsip_cvjdot_*p* Complex Vector Conjugate Dot Product 624
vsip_*d*gemp_*p* General Matrix Product 627
vsip_*d*gems_*p* General Matrix Sum 629
vsip_*ds*kron_*p* Kronecker Product 631
vsip_*d*mprod3_*p* 3 by 3 Matrix Product 633
vsip_*d*mprod4_*p* 4 by 4 Matrix Product 634
vsip_*d*mprod_*p* Matrix Product 635
vsip_cmprodh_*p* Matrix Hermitian Product 636
vsip_cmprodj_*p* Matrix Conjugate Product 637
vsip_*d*mprodt_*p* Matrix Transpose Product 638
vsip_*d*mvprod3_*p* 3 by 3 Matrix Vector Product 639
vsip_*d*mvprod4_*p* 4 by 4 Matrix Vector Product 640
vsip_*d*mvprod_*p* Matrix Vector Product 641
vsip_*d*mtrans_*p* Matrix Transpose 642
vsip_*d*vdot_*p* Vector Dot Product 643
vsip_*d*vmprod_*p* Vector Matrix Product 645
vsip_*d*vouter_*p* Vector Outer Product 646
Special Linear System Solvers 648
vsip_*d*covsol_*p* Solve Covariance System 649
vsip_*d*llsqsol_*p* Solve Linear Least Squares Problem 650
vsip_*d*toepsol_*p* Solve Toeplitz System 651
General Square Linear System Solver 653
vsip_*d*lud_*p* LU Decomposition 654
vsip_*d*lud_create_*p* Create LU Decomposition Object 655
vsip_*d*lud_destroy_*p* Destroy LUD Object 656
vsip_*d*lud_getattr_*p* LUD Get Attributes 657
vsip_*d*lusol_*p* Solve General Linear System 658
Symmetric Positive Definite Linear System Solver 660
vsip_*d*chold_*p* Cholesky Decomposition 661
vsip_*d*chold_create_*p* Create Cholesky Decomposition Object 662
vsip_*d*chold_destroy_*p* Destroy CHOLD Object 663
vsip_*d*chold_getattr_*p* CHOLD Get Attributes 664
vsip_*d*cholsol_*p* Solve SPD Linear System 665

Over-determined Linear System Solver 666
vsip_*d*qrd_*p* QR Decomposition 667
vsip_*d*qrd_create_*p* Create QR Decomposition Object 668
vsip_*d*qrd_destroy_*p* Destroy QRD Object 670
vsip_*d*qrd_getattr_*p* QRD Get Attributes 671
vsip_*d*qrdprodq_*p* Product with Q from QR Decomposition 672
vsip_*d*qrdsolr_*p* Solve Linear System Based on R from QR Dec. 675
vsip_*d*qrsol_*p* Solve Covariance or LLSQ System 677
Singular Value Decomposition 679
vsip_*d*svd_*p* Singular Value Decomposition 680
vsip_*d*svd_create_*p* Create Singular Value Decomposition Object 682
vsip_*d*svd_destroy_*p* Destroy SVD Object 684
vsip_*d*svd_getattr_*p* SVD Get Attributes 685
vsip_*d*svdprodu_*p* Product with U from SV Decomposition 686
vsip_*d*svdprodv_*p* Product with V from SV Decomposition 689
vsip_*d*svdmatu_*p* Return with U from SV Decomposition 692
vsip_*d*svdmatv_*p* Return with V from SV Decomposition 694
IMPLEMENTATION DEPENDENT INPUT AND OUTPUT 697
Introduction 697
Methodology 697
Functionality and Naming 697
I/O Functionality 699
Signal Processing Prototypes 699
Linear Algebra Prototypes 705
vsip_*d*<*object*>_export_*p* Export 708
vsip_*d*<*object*>_find_*p* Find 709
vsip_*d*<*object*>_import_*p* Import 710
vsip_*d*<*object*>_size_*p* Size 711
Examples 712
NOTES TO IMPLEMENTORS 715
VSIPL Implementation Notes 715
Incomplete Type Definitions 715
Checking for Object Validity 715
Change Notes 716
Introduction 716
Errata 716
Changes 720
Additional Functions 721
Deprecated Functions 722
vsip_cstorage Complex Storage 723
THE END 724

GLOSSARY

Admitted	*Block* state where the *data array* (memory) and associated *views* are available for VSIPL computations, and not available for user I/O or access.
Attribute	Characteristic or state of an object, such as *admitted/released, stride,* or *length.*
Binary Function	A function with two input arguments.
Block	A data storage abstraction representing contiguous data elements consisting of a data array and a VSIPL block object.
Block Object	Descriptor for a *data array* and its attributes, including a reference to the data array, the state of the block, data type and size.
Block Offset	The number of *elements* from the start of a *block.* A view with a block offset of zero starts at the beginning of the block.
Boolean	Used to represent the values of true and false, where false is always zero, and true is non-zero.
Bound	A *view* or *block* is bound to a *data array* if it references the data array.
Cloned View	An exact duplicate of a *view object.*
Column	Rightmost dimension in a *matrix.*
Column Stride	The number of *block elements* between successive elements within a *column*.
Complex Block	*Block* containing only complex *elements*. There are two formats for released complex blocks – *split* and *interleaved.* The complex data format for admitted complex blocks are not specified by this standard.
Conformant Views	Views that are the correct shape/size for a given computation.
const Object	An object that is not modified by the function, although data referenced by the const object may be modified.
Create	To allocate memory for an object and initialize it (if appropriate).
Data Array	Memory where data is stored.
Derived Block	A *real block* derived from a *complex block*. Note that the only way to create a derived block is to create a *derived view* of the real or complex component of a *split complex* view. In all other cases, retrieving the block from a view returns a reference to the original block.
Derived View	A *derived view* is a view created using a VSIPL function whose arguments include another view (a parent view). The derived view's data is some subset of the parent view's data. The data subset depends on the function call, and is physically co-located in memory with the parent view's data.
Destroy	To release the memory allocated to an object.
Development Library	An implementation of VSIPL that maximizes error reporting at the possible expense of performance.

Domain	The set of all valid input values to a function.
Element	The atomic portion of data associated with a *block* or a *view*. For example, an element of a complex *block* of precision double is a complex number of precision double; for a *view* of type float an element is a single float number.
Hermitian Transpose	Conjugate transpose.
Hint	Information provided by the user to some VSIPL functions to aid optimization. *Hints* are optional and may be ignored by the implementation. Wrong hints may result in incorrect behavior.
Implementor	The individual or group creating a VSIPL implementation.
In-Place	A type of algorithm implementation in which the memory used to hold the input to an algorithm is overwritten (completely or partially) with the output data. Often referred to in the context of an FFT algorithm.
Interleaved Complex	Storage format for *user data arrays* where the real and complex *element* components alternate in physical memory.
Kernel	The filter vector used in a FIR filter, or the vector or matrix used as the weights in a convolution.
Length	Number of *elements* in a *view* along a *view dimension.*
M-ary Function	A function with *M* arguments.
Matrix	A two dimensional *view*.
N-ary Function	A *null*-ary function without input arguments (for by-element functions, function of element index, not element value).
Opaque	An *opaque* object may not be manipulated by simple assignment statements. Its attributes must be set/retrieved through access functions. All VSIPL objects are opaque.
Out-of-place	If none of the output views in a function call overlap the input views, the function is considered *out-of-place.*
Overlapped	Indicates that two or more *views* or *blocks* share one or more memory locations.
Portable Precision	A data type with a guarantee of a specified minimum precision or an exact precision on all supported implementations.
Production Library	A VSIPL implementation that maximizes performance at the possible expense of not detecting user errors.
Range	Valid output values from a function.
Real Block	A *block* containing only real *elements.*
Region of Support	For neighborhood operations (*i.e.* FIR filtering, convolution, …), the non-zero values in the kernel, or the output. [3×3 FIR filter has a "kernel region of support" of 3×3.]
Released	*Block* state where the associated *data array* is available for user I/O and application access, but not available for VSIPL computations.

Row	Left-most dimension of a *matrix.*
Row Stride	The number of *block elements* between successive elements within a *row*.
Split Complex	Storage format for released *complex blocks* where the real *element* components are stored in one physically contiguous *data array*, and the imaginary components are stored in a separate physically contiguous data array.
Stride	Distance between successive *elements* of the block data array in a *view* along a *view* dimension. Strides can be positive, negative, or zero.
Subview	A *derived view* that describes a subset of the data from the original view, and is the same type as the original view.
Tensor	An *n*-dimensional matrix. VSIPL only supports 3 dimensional tensors (3-tensor). The three dimensions are referred to as X, Y and Z.
Ternary Function	A function with 3 input arguments.
Unary Function	A function with a single input argument.
User Block	A block which is associated with user data arrays. User blocks are created in the released state and may be admitted and released.
User Data Array	Memory that has been allocated by the application for the storage of data using some functionality not part of the VSIPL standard.
Vector	A one dimensional view.
View	A portion of a *block*, and a *view object* describing it. The view object has structural information allowing the data to be interpreted as a one, two or three-dimensional array for arithmetic processing.
View Dimension	A *view* represents a one, two, or three dimensional data organization termed respectively a vector, matrix or tensor. A *view dimension* represents one of the standard directions of these data representations.
View Object	A description of a portion of a *block* including structural information that allows the data to be interpreted as a one, two or three-dimensional array for arithmetic processing. Attributes of the *view object* include *offset, stride(s)* and *length(s).*
VSIPL Block	*Block* referencing or bound to VSIPL data. A VSIPL block is created in the admitted state and may not be released.
VSIPL Data Array	Memory that has been allocated for the storage of data using some functionality that is part of the VSIPL standard.

NOTATION NOTES

Matrix, tensor, image	$\mathbf{A}$, $\mathbf{B}$, $\mathbf{C}$
Column Vector	$\mathbf{x}$, $\mathbf{y}$, $\mathbf{z}$
Real or complex scalars	α, β, γ
Real or complex scalars	r, s, t
Long scalar names	mean
vector element	x_i, y_i, z_i
matrix (tensor) element	$a_{i,j}$, $b_{i,j}$, $c_{i,j}$
Functions	cosh θ or cosh(θ)
Text	text
Minimum, Maximum	$\min\{\cdots\}$,$\max\{\cdots\}$
Product	$\alpha \cdot \beta$ $\alpha \cdot \mathbf{v}$ $\mathbf{AB}$ or $\mathbf{A} \cdot \mathbf{B}$ $\mathbf{Av}$ or $\mathbf{A} \cdot \mathbf{v}$
Matrix shape	M by N
Absolute value	$\lvert x \rvert$ or $\lvert x \rvert^2$
Norms	$\lVert \mathbf{v} \rVert_p$ or $\lVert \mathbf{A} \rVert_p$
Floor, largest integer less than or equal to x.	$\lfloor x \rfloor$
Ceiling, smallest integer greater than or equal to x.	$\lceil x \rceil$
Transpose	$\mathbf{A}^T$, $\mathbf{v}^T$
Hermitian (conjugate) transpose	$\mathbf{A}^H$, $\mathbf{v}^H$
Conjugate	$\mathbf{A}^*$, $\mathbf{v}^*$
The result of the right hand side expression is x.	$x \leftarrow expression$
Index Range	for $i = 0,1,\cdots,N-1$ for $i = 0,1,\cdots,M-1$; for $j = 0,1,\cdots,N-1$

CODE STYLE NOTES

Function return value	Returns an object of type *object_type* by reference	`object_type *func_name(…)`
Function return value	Returns a pointer to type *data_type*	`data_type* func_name(…)`
Function return value	Returns a pointer to an object of type *object_type* by reference	`object_type* *func_name(…)`

VSIPL INTRODUCTION

Introduction to the VSIPL Standard

The purpose of the Vector, Signal, and Image Processing Library (VSIPL) standard is to support portable, high performance application programs. The standard is based upon existing libraries that have evolved and matured over decades of scientific and engineering computing. A layer of abstraction is added to support portability across diverse memory and processor architectures. The primary design focus of the standard has been embedded signal processing platforms. Enhanced portability of workstation applications is a side benefit.

Platform Requirements

VSIPL was specified so that it could be implemented on a wide variety of hardware. In order to use VSIPL functions on a given platform, a VSIPL compliant library must be available for the particular hardware and tool-set (linker) available for the operating system. Since the library API is defined for ANSI C, for practical applications, an ANSI C compiler will also be necessary.

VSIPL Functionality

The VSIPL standard provides a number of functions to the programmer to support high performance numerical computation on dense rectangular arrays. These are organized in the VSIPL documentation according to category. The available categories include:

- Support
  - Library initialization and finalization
  - Object creation and interaction
  - Memory management
- Basic Scalar Operations
- Basic Vector Operations
- Random Number Generation
- Signal Processing
  - FFT operations
  - Filtering
  - Correlation and convolution
- Linear Algebra
  - Basic matrix operations
  - Linear system solution
  - Least-squares problem solution

Although there are many functions in the VSIPL specification, not all functions are available in all libraries. The contents of a specific VSIPL library subset are defined in a *profile*. As of the completion of VSIPL 1.0 two profiles have been approved by the VSIPL Forum, referred to as the

“Core” and “Core Lite” profiles. The “Core” profile includes most of the signal processing and matrix algebra functionality of the library. The “Core Lite” profile includes a smaller subset,

suitable for vector-based signal processing applications. The VSIPL specification defines more functions than are present in either of these profiles.

VSIPL Objects

The main difference between the proposed VSIPL standard and existing libraries is a cleaner encapsulation of memory management through an "object-based" design. In VSIPL, a *block* can be thought of as a contiguous area of memory for storage of data. A *block* consists of a *data array*, which is the memory used for data storage; and a *block object*, which is an abstract data type which stores information necessary for VSIPL to access the *data array*. VSIPL allows the user to construct a *view* of the data in a block as a vector, matrix, or higher dimensional object. A *view* consists of a *block*, which contains the data of interest; and a *view object*, which is an abstract data type which stores information necessary for VSIPL to access the data of interest.

Blocks and views are opaque. They can only be created, accessed and destroyed via library functions. Object data members are private to hide the details of non-portable memory hierarchy management. VSIPL library developers may hide information peculiar to their implementations in the objects in order to prevent the application programmer from accidentally writing code that is neither portable nor compatible.

Data arrays in VSIPL exist in one of two logical data spaces. These are the user data space, and VSIPL data space. VSIPL functions may only operate on data in VSIPL space. User supplied functions may only operate on data in user space. Data may be moved between these logical spaces. Depending on the specific implementation, this move may incur actual data movement penalties or may simply be a bookkeeping procedure. The user should consider the data in VSIPL space to be inaccessible except through VSIPL functions.

Other Features of the Specification

The VSIPL specification provides support for multiple floating-point and integer data types. In addition, methods are defined for a vendor to supply tools to allow the user to specify precision requirements in an application. For example, a user may specify that *at least* 16 bits of precision are needed. On an embedded platform, this might translate exactly to a 16-bit type, while on a workstation during development the actual type used might have 32 bits of precision. VSIPL allows the same code to work on both platforms.

Two versions of the library are described, referred to as development and performance libraries. These libraries operate the same with the exception of error reporting and timing. Performance versions of a VSIPL library are not guaranteed to provide any error detection or handling except in the case of memory allocation. Other programming errors under a VSIPL performance library may have unpredictable results, up to and including complete system crashes. Development libraries are expected to run slower than performance libraries but include more error detection capabilities. Suppliers of VSIPL compliant libraries are not required to provide both versions; they may choose to supply either version, both versions, or a single library that supports both development and performance modes, as desired.

Basic VSIPL Concepts

General Library Design Principles

The VSIPL standard supports high performance numerical computation on dense rectangular arrays. The standard incorporates the following well-established characteristics of existing scientific and engineering libraries:

1. Elements are stored in one dimensional data arrays, which appear to the application programmer as a single contiguous block of memory.
2. Data arrays can be viewed as either real or complex vectors, matrices, or tensors.
3. All operations on data arrays are performed indirectly through view objects, each of which specify a particular view of a data array with a particular offset, length(s) and stride(s).
4. In general, the application programmer cannot combine operators in a single statement to evaluate expressions. Operators which return scalar may be combined, but most operators will return a view type or are void and may not be combined.

Operators are restricted to views of a data array that can be specified by an offset, lengths and strides. Views that are more arbitrary are converted into these simple views by functions like gather and back again by functions like scatter. VSIPL does not support triangular or sparse matrices very well, though future extensions might address these.

The main difference between the proposed VSIPL standard and existing libraries is a cleaner encapsulation of the above principles through an "object-based" design. All of the view attributes are encapsulated in opaque objects[1]. The object can only be created, accessed and destroyed via library functions, which reference it via a pointer.

Memory Management

The management of memory is important to efficient algorithm development. This is especially true in embedded systems, many of which are memory limited. In VSIPL, memory management is handled by the implementation. This section describes VSIPL memory management and how the user interacts with VSIPL objects.

Terminology

The terms *user data*, *VSIPL data*, *admitted*, and *released* are used throughout this document when describing memory allocation. It is important that the reader understand the terms that are defined in this section below, and in the Glossary above.

Object Memory Allocation

All objects in VSIPL consist of abstract data types (ADT) which contain attributes defining the underlying data accessed by the object. Certain of the attributes are accessible to the application programmer via access functions; however, there may be any number of attributes assigned by the VSIPL library developer for its internal use. Each time an object is defined, memory must be allocated for the ADT. All VSIPL objects are allocated by VSIPL library functions. There is no method by which the application programmer may allocate space for these objects outside of VSIPL.

[1] Object opacity is achieved through the technique of "incomplete typedef," described in the section on implementation.

Most VSIPL objects are relatively small and of fixed size; however, some of the objects created for signal processing or linear algebra may allocate large workspaces.

Data Memory Allocation

A *data array* is an area of memory where data is stored. Data arrays in VSIPL exist in one of two logical data spaces. These are the user data space, and VSIPL data space. VSIPL functions may only operate on data in VSIPL space. User supplied functions may only operate on data in user space. Data may be moved between these logical spaces. Depending on the specific implementation, this move may incur actual data movement penalties or may simply be a bookkeeping procedure. The user should consider the data in VSIPL space to be inaccessible except through VSIPL functions.

A data array allocated by the application, using any method not part of the VSIPL standard, is considered to be a user data array. The application has a pointer to the user data array and knowledge of its type and size. Therefore the application can access a user data array directly using pointers, although it is not always correct to do so.

A data array allocated by a VSIPL function call is referred to as a *VSIPL data array*. The user has no proper method to retrieve a pointer to such a data array; it may only be accessed via VSIPL function calls.

Users may access data arrays in VSIPL space using an entity referred to as a *block*. The data array associated with a block is a contiguous series of elements of a given type. There is one block type for each type of data processed by VSIPL.

There are two categories of blocks, *user blocks* and *VSIPL blocks*. A user block is one that has been associated with a user data array. A VSIPL block is one that has been associated with a VSIPL data array. The data array referenced by the block is referred to as being “bound” to the block. The user must provide a pointer to the associated data for a user block. The VSIPL library will allocate space for the data associated with a VSIPL block. Blocks can also be created without any data and then associated with data in user space. The process of associating user space data with a block is called “binding.” A block which does not have data bound to it may not be used, as there is no data to operate on.

A block that has been associated with data may exist in one of two states, *admitted* and *released*. The data in an admitted block is in the logical VSIPL data space, and the data in a released block is in the logical user data space. The process of moving data from the logical user data space to the logical VSIPL data space is called admission; the reverse process is called release.

Data in an admitted block is owned by the VSIPL library, and VSIPL functions operate on this data under the assumption that the data will only be modified using VSIPL functions. VSIPL blocks are always in the admitted state. User blocks may be in an admitted state. User data in an admitted block shall not be operated on except by VSIPL functions. Direct manipulation of user data bound to an admitted block via pointers to the allocated memory is incorrect and may cause erroneous behavior.

Data in a released block may be accessed by the user, but VSIPL functions should not perform computation on it. User blocks are created in the released state. The block must be admitted to VSIPL before VSIPL functions can operate on the data bound to the block. A user block may be admitted for use by VSIPL and released when direct access to the data is needed by the application program. A VSIPL block may not be released.

Blocks represent logically contiguous data areas in memory (physical layout is undefined for VSIPL space), but users often wish to operate on non-contiguous sub-sets of these data areas. To provide support for such operations, VSIPL requires that users operate on the data in a block through another object type called a *view*. Views allow the user to specify noncontiguous subsets of a data array and inform VSIPL how the data will be accessed (for example, as a vector or matrix). When creating a vector view, the user specifies an offset into the block, a view length, and a stride value which specifies the number of elements (defined in the type of the block) to advance between each access. Thus, for a block whose corresponding data array contains four elements, a view with an offset value of zero, a stride of two, and a length of two represents a logical data set consisting of members zero and two of the original block. For a matrix view, stride and length parameters are specified in each dimension, and a single offset is specified. By varying the stride, row-major or column-major matrices can be created.

A block may have any number of views created on it: this allows the user to use vector views to access particular rows or columns of a matrix view, for example. Since the blocks are typed, views are also typed; however, because views also include usage information (e.g. vector or matrix), there are multiple view types for each block type corresponding to how the data will be accessed. These types are immutable; thus for example, a block cannot have both integer and float views associated with it. This would not be useful in any event because the data layout inside VSIPL space is vendor specific.

New views of a block may be created directly using a block object, or indirectly using a previously created view of the block. Except for finding the real or imaginary view of a complex view all views may be created directly using the block object.

Structure of a VSIPL application

Although there are a number of ways to program an application, the basic VSIPL program consists of the following sequence:

A VSIPL program must initialize the VSIPL library with a call to `vsip_init` before calling any other VSIPL function. Any program that uses VSIPL and that terminates must call `vsip_finalize` before terminating. See the Support chapter for additional conditions and restrictions on these functions.

Introduction to VSIPL Data Types

VSIPL Naming Conventions

While there is nothing to prevent a programmer from writing VSIPL-compatible functions, only those functions that are approved and included in formal VSIPL documentation are a part of VSIPL. Functions outside the standard should not use the VSIPL naming conventions in order to avoid confusion and application porting problems. In particular, names outside of VSIPL should not start with "`vsip`" or "`vsipl`", either in caps or lower case.

The exact names of VSIPL functions depend on the precision of the operation. The base names of VSIPL functions are explicitly specified in the functionality section. The VSIPL function name consists of the base name plus a precision affix that specifies the data precision. This is done to allow wide variation in precision to support diverse hardware. The allowed affixes are covered in the summary of VSIPL types below. Except for copy functions, the precision affix is usually a suffix. The precision affix for a copy function consists of two precision affixes corresponding to the data types of the source and destination data arrays.

In addition to the precision affix one or more precision *depth* qualifiers (real or complex) and *shape* qualifiers (scalar, vector, matrix, tensor) are placed as a prefix in the front of the root name.

The actual qualifier for *real* or *complex* will be an "`r`" or "`c`". For a generalized name an italic ***d*** is used to indicate either real or complex. Generally the *real* (`r`) prefix is understood (not included as part of the name) if there can be no confusion.

The qualifier for scalar, vector, matrix and tensor are prefixes of "`s`", "`v`", "`m`", and "`t`" respectively. For a generalized name the shape is indicated using an italic ***s*** for any shape. Generally the *scalar* prefix (`s`) is understood (not included as part of the name) if there can be no confusion.

All VSIPL type declarations and function names have the data type encoded into the name. Throughout the VSIPL documentation, a *generalized affix* of ***_p*** is used to denote a general precision of any type. This is a method to name functions or data types without spelling out every single prefix that might be needed for that function or data type. Other generalized affixes used are ***_i*** to denote any integer, or an ***_f*** to denote any float. The generalized affix is in an italic font style. To produce a valid VSIPL name use a specified name from the functionality section, and replace the generalized data type affix with the selected affix from the *Summary of VSIPL Types*.

For example, the function `vsip_mag`***_p*** listed in the functionality section takes the magnitude (absolute value) of its argument. A specific instance of this function which operates on single-precision floating-point data would be called be `vsip_mag_f`, and a version that operates on integer data would be called `vsip_mag_i`. As a further example, consider the complex FFT function `vsip_ccfftop`***_f***. The "***_f***" suffix indicates that instances of the function that operate on single-precision, double-precision, and extra-precision floating-point data are part of the specification, while an instance that operates on integer data is not currently included.

VSIPL affix notations for use in encoding type data in the names and type declarations are described in the *Summary of VSIPL Types*. It is not expected that any implementation will support all possible VSIPL data types. The data types supported will depend in part on the hardware for which the library was developed, and the expected use of the hardware.

Non-Standard Scalar Data Types

In general, VSIPL scalar data types correspond to particular C data types depending on the underlying implementation. However, ANSI C does not define boolean or complex scalar types, both of which are defined in VSIPL. This section summarizes requirements for these data types.

Boolean Data Types

The VSIPL boolean data type (bl) is either true or false when used by a VSIPL function which sets or uses the boolean type. If a numeric vector or matrix is copied to a boolean vector or matrix, then the value zero is copied to the boolean as false. Any other value is copied as true. If a boolean vector or matrix is copied to a numeric vector or matrix then the value false is copied as a zero, and the value true is copied as positive one. If a VSIPL function returns a boolean scalar then a false is returned as zero and a true is non-zero. The actual value of the non-zero true is implementation dependent. If a scalar is tested as boolean using a VSIPL function then a zero is tested as false and a non-zero is tested as true.

Complex Data Types

The definition of the complex scalar is available in public header files, and has the usual structure for complex data as normally defined in ANSI C programs. The exact form of complex for VSIPL is defined in the chapter *Summary of VSIPL Types*. In general, users are encouraged to not use the structure directly, but to instead use VSIPL scalar functions for manipulating complex scalars. This should enhance portability of user code.

Data Array Layout

A user data array that is bound to a block has a particular required layout, depending on the type of the block. This section describes the required layout of the user data array for various block types. The implementor must support, and the application programmer must use, the data array formats for user data. These formats allow portable input of user data into VSIPL, and portable output of VSIPL results to the application.

For basic VSIPL types, the user data array is simply contiguous memory of the corresponding VSIPL type. This applies to floating-point (`vsip_scalar_f`), integer (`vsip_scalar_i`), boolean (`vsip_scalar_bl`), and vector index (`vsip_scalar_vi`) types.

For either matrix or tensor index data, the user data array is contiguous memory of type `vsip_scalar_vi`. In the case of matrix index data, each element is two consecutive elements of type `vsip_scalar_vi`. The first element is the row, the second is the column. In the case of tensor index data, each element is three consecutive elements of type `vsip_scalar_vi`. The first value in the element is the z index, the second is the y index, and the third is the x index. Note that the matrix index element in a user data array is not the same as `vsip_scalar_mi`, and the tensor index element in a user data array is not the same as `vsip_scalar_ti`.

For complex float or complex integer data, the user data array is either *interleaved* or *split* as described below. Both the interleaved and split formats must be supported for user data. Internal storage of complex data is implementation dependent. Note that the data format for complex user data arrays is **not** of type `vsip_cscalar_p`.

Interleaved: The user data array is contiguous memory of type `vsip_scalar_p`. The complex element is two consecutive elements of type `vsip_scalar_p`. The first element is the real component and the second is the imaginary component.

Split: The user data array consists of two contiguous memory regions of equal length, each of type `vsip_scalar_p`. The real and the imaginary region are determined when the

memory is bound to the block. A complex element consists of corresponding elements from the real and imaginary regions.

Summary of Scalar and View Types

The following table summarizes common VSIPL data types used in function prototypes. (For a complete set of VSIPL types see the *Summary of VSIPL Types* chapter.) Note that letters in italic indicate that more than one data type may be implemented. For instance, `vsip_scalar_`***f*** indicates a scalar float value. The implementor, depending upon their hardware and other vendor specific reasons, may implement one or several float types. The actual declarations available for different precision may be implemented as `vsip_scalar_f`, `vsip_scalar_d`, and `vsip_scalar_l`; corresponding to float, double and long double. A more limited implementation may only supply a `vsip_scalar_f`.

Typedef	Description	Precision
`vsip_scalar_`***i***	Type definition of an integer scalar	***i***
`vsip_cscalar_`*i*	Type definition of a complex integer scalar	***i***
`vsip_vview_`***i***	Type definition of an integer vector view	***i***
`vsip_mview_`***i***	Type definition of an integer matrix view	***i***
`vsip_tview_`***i***	Type definition of an integer tensor view	***i***
`vsip_cvview_`***i***	Type definition of a complex integer vector view	***i***
`vsip_cmview_`***i***	Type definition of a complex matrix view	***i***
`vsip_ctview_`***i***	Type definition of a complex tensor view	***i***
`vsip_scalar_`***f***	Type definition of a real float scalar.	***f***
`vsip_cscalar_`***f***	Type definition of a complex float scalar	***f***
`vsip_vview_`***f***	Type definition of a real vector view	***f***
`vsip_mview_`***f***	Type definition of a real floating point matrix view.	***f***
`vsip_tview_`***f***	Type definition of a real floating point tensor view.	***f***
`vsip_cvview_`***f***	Type definition of a complex vector view	***f***
`vsip_cmview_`***f***	Type definition of a complex floating point matrix view.	***f***
`vsip_ctview_`***f***	Type definition of a complex floating point tensor view.	***f***
`vsip_scalar_bl`	Type definition of a single boolean value	Vendor Dependent
`vsip_vview_bl`	Type definition of a boolean vector view	`bl`
`vsip_mview_bl`	Type definition of a boolean matrix view	`bl`
`vsip_tview_bl`	Type definition of a boolean tensor view	`bl`
`vsip_scalar_vi`	Type definition of a vector index with a single element.	Vendor Dependent
`vsip_vview_vi`	Type definition of a vector index vector view	`vi`

Typedef	Description	Precision
`vsip_scalar_mi`	Type definition of a matrix index with a single element	`vi`
`vsip_vview_mi`	Type definition of a matrix index vector view	`vi`
`vsip_scalar_ti`	Type definition of a tensor index with a single element	`vi`
`vsip_vview_ti`	Type definition of a tensor index vector view	`vi`
`vsip_offset`	Type definition of an offset for a vector/matrix/tensor view	`vi`
`vsip_length`	Type definition of a length for a vector/matrix/tensor view	`vi`
`vsip_stride`	Declaration of a stride for a vector/matrix/tensor view	Vendor Dependent

Description of the API Specification Pages

Most of the VSIPL documentation consists of specifications of VSIPL functions. The API specification is done in a format similar to a man page in UNIX and so these pages are generally described as manual pages, or man pages. This section describes the format of these manual pages in more detail.

Function Name

At the top of each page is the function name. Typically, this name will include one or more generalized precision affixes, as described in the types section, that indicate the data that this function operates on (e.g. *_f* for all types of floating-point data). Where a function operates on more than one data type, example functions for each of the data types are given. When similar functions operate on both real and complex data, they may be given separate man pages.

Functionality

A brief description of the function, including a mathematical definition where appropriate.

Prototypes

This gives the C prototype for the function. Note defined prototypes also delineate the defined VSIPL functions. For instance if a prototype is defined with a precision of *_p* then the function may be written (by the implementor) for any defined VSIPL type. However, if the function is defined with a precision of *_f* and no prototypes appear with *_i*, then the function is only defined for float types and is not defined for integers. If implementors provide functionality which is not defined in the specification using this functionality in application code may hinder portability if a new implementation is chosen.

Arguments

The arguments section describes each of the arguments to the function in more detail.

Return Value

The return value section notes possible values that may be returned. Functions that allocate memory return NULL if the memory allocation fails. Functions that return integers for success or failure return zero on success.

Restrictions

The restrictions section is used to denote special restrictions such as domain limitations, lack of in-place functionality, etc. It may also indicate other error conditions such as overflow and underflow. These errors are not checked for specifically by the VSIPL library functions. The results of such errors are implementation dependent.

Errors

This section describes conditions that if not met, are considered an error.

Note that memory allocation failure is not an error. It is always signaled via the function's return value, and should always be checked by the application.

In development mode:

1. All implementations are required to check all of the specified error conditions.
2. When an error is detected, an informative error message should be produced. Error messages should be sent to `stderr`, if it exists, otherwise logging is implementation dependent.

3. All error conditions are considered fatal. After logging the error message, the implementation should call the ANSI C exit function with a (implementation dependent) return code value. (The behavior of `exit` is system dependent.)
4. VSIPL does not specify any *warning* conditions. Implementations may provide *warnings* in development mode. A *warning* is limited to the production or logging of a warning message.

Because of the cost of checking views for overlap errors in the general case, overlap checking is only required to test for the case of identical blocks with the same offset. However, complete overlap checking may be provided.

In production mode:

No error checking of any kind is required.

Implementations may define additional error detecting and handling behavior, but it shall not use names within the VSIPL namespace (may not start with a leading "vsip" or "vsipl", either in caps or lower case).

Conformant. The most common required condition is *conformance*. This is used to denote objects and or views that have compatible attributes. For example, to add two vectors together, we require the two inputs and the output vector (view objects) to be of the same length.

Valid. The second common required condition is *valid object*. A *valid* object is one that has been initialized and not destroyed. In the context of a function that operates on a data array, it also requires that the corresponding block be *admitted*. A NULL object pointer is not considered valid.

Notes/References

This section may contain references to preferred implementations of the described algorithm, or notes about special cases and circumstances for the particular function.

Examples

This section gives examples of use for the function. The details available in examples vary widely. The most complete examples include input and output data. Other examples are merely code fragments.

See Also

This section lists functions whose use is related to the described function.

SUMMARY OF VSIPL TYPES

Type - Introduction

Notes: Highlight (in the VSIPL TYPES section only) denotes an implementation dependent name, type or value.

ANSI C macros found in this section are defined in `float.h` and `limits.h`.

The basic scalar types are:

- Floating point
  - Denoted with suffix *_f*, which is one or more of the types:
    - `_f (float)`
    - `_d (double)`
    - and `_l (long double)`
  - Portable precision types
    - `_f6` Floating point types with at least 6 decimal digits of accuracy. IEEE 754 single precision (32 bit) has 6 decimal digits of accuracy.
    - `_f15` Floating point types with at least 15 decimal digits of accuracy. IEEE 754 double precision (64 bit) has 15 decimal digits of accuracy.
    - `_f`*n* Floating point type with at least *n* decimal digits of accuracy. If the system supports such a precision, it resolves to the smallest C type based on the values of `FLT_MANT_DIG`, `DBL_MANT_DIG`, or `LDBL_MANT_DIG`.
- Integer
  - Denoted with suffix *_i* which is one or more of the types:
    - `_c (char)`
    - `_uc (unsigned char)`
    - `_si (short int)`
    - `_us (unsigned short int)`
    - `_i (int)`
    - `_u (unsigned int)`
    - `_li (long int)`
    - `_ul (unsigned long int)`
    - `_ll (long long int)`
    - `_ull (unsigned long long int)`
    - `_vi (unsigned long int)`
  - Portable precision types
    - Of at least *n* bits:
      `_il8` *int* of at least 8 bits
      `_il16` *int* of at least 16 bits
      `_il32` *int* of at least 32 bits
      `_il64` *int* of at least 64 bits
      `_il`*n* *int* of at least *n* bits

`_ul8` unsigned *int* of at least 8 bits
`_ul16` unsigned *int* of at least 16 bits
`_ul32` unsigned *int* of at least 32 bits
`_ul64` unsigned *int* of at least 64 bits
`_ul`*`n`* unsigned *int* of at least *`n`* bits

- Of exactly *n* bits:
  `_ie8` *int* of exactly 8 bits
  `_ie16` *int* of exactly 16 bits
  `_ie32` *int* of exactly 32 bits
  `_ie64` *int* of exactly 64 bits
  `_ie`*`n`* *int* of exactly *`n`* bits
  `_ue8` unsigned *int* of exactly 8 bits
  `_ue16` unsigned *int* of exactly 16 bits
  `_ue32` unsigned *int* of exactly 32 bits
  `_ue64` unsigned *int* of exactly 64 bits
  `_ue`*`n`* unsigned *int* of exactly *`n`* bits
- Fastest type of at least *n* bits:
  `_if8` fastest *int* of at least 8 bits
  `_if16` fastest *int* of at least 16 bits
  `_if32` fastest *int* of at least 32 bits
  `_if64` fastest *int* of at least 64 bits
  `_if`*`n`* fastest *int* of at least *`n`* bits
  `_uf8` unsigned fastest *int* of at least 8 bits
  `_uf16` unsigned fastest *int* of at least 16 bits
  `_uf32` unsigned fastest *int* of at least 32 bits
  `_uf64` unsigned fastest *int* of at least 64 bits
  `_uf`*`n`* unsigned fastest *int* of at least *`n`* bits

- Integer or Floating Point
  - Denoted with suffix *`_p`*, which is the union of *`_f`* and *`_i`*.
- Boolean
  - Logical *false* for zero, and logical *true* for non-zero.
  - Denoted with suffix: `_bl`
- Vector Index
  - Unsigned integer denoting the index of a vector element. The *vector index* of the element x_i is i.
  - Denoted with suffix: `_vi`
- Matrix Index
  - Unsigned integer denoting the index of a matrix element. The *matrix index* of the element $x_{i,j}$ is the 2-tuple $\{i, j\}$.
  - Denoted with suffix: `_mi`
- Tensor Index
  - Unsigned integer denoting the index of a 3-tensor element. The *tensor index* of the element $x_{i,j,k}$ is the 3-tuple $\{i, j, k\}$.
  - Denoted with suffix: `_ti`

Scalar

Real Scalar

```
#define  VSIP_PI              M_PI

#define  VSIP_MAX_SCALAR_L    LDBL_MAX
#define  VSIP_MAX_SCALAR_D    DBL_MAX
#define  VSIP_MAX_SCALAR_F    FLT_MAX
#define  VSIP_MAX_SCALAR_LI   LONG_MAX
#define  VSIP_MAX_SCALAR_UL   ULONG_MAX
#define  VSIP_MAX_SCALAR_I    INT_MAX
#define  VSIP_MAX_SCALAR_U    UINT_MAX
#define  VSIP_MAX_SCALAR_SI   SHRT_MAX
#define  VSIP_MAX_SCALAR_US   USHRT_MAX
#define  VSIP_MAX_SCALAR_C    CHAR_MAX
#define  VSIP_MAX_SCALAR_UC   UCHAR_MAX

#define  VSIP_MIN_SCALAR_L    LDBL_MIN
#define  VSIP_MIN_SCALAR_D    DBL_MIN
#define  VSIP_MIN_SCALAR_F    FLT_MIN
#define  VSIP_MIN_SCALAR_LI   LONG_MIN
#define  VSIP_MIN_SCALAR_UL   0
#define  VSIP_MIN_SCALAR_I    INT_MIN
#define  VSIP_MIN_SCALAR_U    0
#define  VSIP_MIN_SCALAR_SI   SHRT_MIN
#define  VSIP_MIN_SCALAR_US   0
#define  VSIP_MIN_SCALAR_C    CHAR_MIN
#define  VSIP_MIN_SCALAR_UC   0

#ifdef LLONG_MAX
#define  VSIP_MAX_SCALAR_LL   LLONG_MAX
#define  VSIP_MAX_SCALAR_ULL  ULLONG_MAX
#define  VSIP_MIN_SCALAR_LL   LLONG_MIN
#define  VSIP_MIN_SCALAR_ULL  0
#endif /* LLONG_MAX */
typedef              long  double   vsip_scalar_l;
typedef                    double   vsip_scalar_d;
typedef                    float    vsip_scalar_f;
typedef  signed      long  int      vsip_scalar_li;
typedef  unsigned    long  int      vsip_scalar_ul;
typedef  signed            int      vsip_scalar_i;
typedef  unsigned          int      vsip_scalar_u;
typedef  signed      short int      vsip_scalar_si;
typedef  unsigned    short int      vsip_scalar_us;
typedef  signed      char           vsip_scalar_c;
typedef  unsigned    char           vsip_scalar_uc;
#ifdef LLONG_MAX
typedef   signed     long long      vsip_scalar_ll;      Non-ANSI C
typedef unsigned long      long     vsip_scalar_ull;     Non-ANSI C
#endif /* LLONG_MAX */
```

Complex Scalar

```
typedef struct { vsip_scalar_p  r, i; } vsip_cscalar_p;
```

Boolean

```
typedef unsigned int vsip_scalar_bl;
typedef vsip_scalar_bl vsip_bool;

#define VSIP_FALSE 0
#define VSIP_TRUE 1
```

However, just as in ANSI C, testing equality of booleans that are *true* may not result in a logical true.

```
vsip_bool a;
vsip_bool b;
…
/* a and b may both be true, but the logical test may return false */
if(a == b) {
/* true */
…
} else {
/* false */
…
}
```

Index

Vector index

```
typedef unsigned long int vsip_scalar_vi;
```

Matrix index

```
typedef struct { vsip_scalar_vi r,c;} vsip_scalar_mi;
```

Tensor index

```
typedef struct { vsip_scalar_vi z,y,x;} vsip_scalar_ti;
```

Element index; a synonym for vector index

```
typedef vsip_scalar_vi vsip_index;
#define VSIP_MAX_SCALAR_VI ULONG_MAX
#define VSIP_MIN_SCALAR_VI 0
```

Offset, Stride, Length

Unsigned offset in elements

```
typedef vsip_scalar_vi vsip_offset;
```

Stride in elements between successive elements in memory along a dimensions (row, column for matrices, or X, Y, Z for tensors)

```
typedef signed long int vsip_stride;
```

Unsigned length in elements, a synonym for vector index

```
typedef vsip_scalar_vi vsip_length;
```

Random Number Generator

Object type for the random number generator's state information.

```
struct vsip_randomstate;
typedef struct vsip_randomstate vsip_randstate;
```

Enumerated type for the preferred random number generator.

```
typedef enum {
  VSIP_PRNG = 0,                    Portable random number generator
  VSIP_NPRNG = 1                    Non-portable random number generator
} vsip_rng;
```

Block Object

Hint used when allocating *VSIPL data* memory for a block object.

```
typedef enum {
  VSIP_MEM_NONE = 0,                No hint
  VSIP_MEM_RDONLY = 1,              Read Only
  VSIP_MEM_CONST = 2,               Constant
  VSIP_MEM_SHARED = 3,              Shared
  VSIP_MEM_SHARED_RDONLY = 4,       Shared, Read Only
  VSIP_MEM_SHARED_CONST = 5         Shared, Constant
} vsip_memory_hint;
```

Enumerated type for the preferred memory storage layout of complex data.

```
typedef enum {
  VSIP_CMPLX_INTERLEAVED = 0,       Interleaved
  VSIP_CMPLX_SPLIT = 1,             Split, separate real and imaginary
  VSIP_CMPLX_NONE = 2               No preferred storage layout
} vsip_cmplx_mem;
```

Object type for a block of boolean data.

```
struct vsip_blockobject_bl;
typedef struct vsip_blockobject_bl vsip_block_bl;
```

Object type for a block of vector index data.

```
struct vsip_blockobject_vi;
typedef struct vsip_blockobject_vi vsip_block_vi;
```

Object type for a block of matrix index data.

```
struct vsip_blockobject_mi;
typedef struct vsip_blockobject_mi vsip_block_mi
```

Object type for a block of tensor index data.

```
struct vsip_blockobject_ti;
typedef struct vsip_blockobject_ti vsip_block_ti;
```

Object type for a block of integer, floating point data.

```
struct vsip_blockobject_i;
typedef struct vsip_blockobject_i vsip_block_i;

struct vsip_blockobject_f;
typedef struct vsip_blockobject_f vsip_block_f;
```

Object type for a block of complex integer, or complex floating point data.

```
struct vsip_cblockobject_i;
typedef struct vsip_cblockobject_i vsip_cblock_i;

struct vsip_cblockobject_f;
typedef struct vsip_cblockobject_f vsip_cblock_f;
```

Vector View

Real Vector View Object

Attribute structure for a vector view of vector indices.

```
typedef struct {
  vsip_offset offset;
  vsip_stride stride;
  vsip_length length;
  vsip_block_vi *block;
} vsip_vattr_vi;
```

Object type for a vector view of vector indices.

```
struct vsip_vviewobject_vi;
typedef struct vsip_vviewobject_vi vsip_vview_vi;
```

Attribute structure for a vector view of matrix indices.

```
typedef struct {
  vsip_offset offset;
  vsip_stride stride;
  vsip_length length;
  vsip_block_mi *block;
} vsip_vattr_mi;
```

Object type for a vector view of matrix indices.

```
struct vsip_vviewobject_mi;
typedef struct vsip_vviewobject_mi vsip_vview_mi;
```

Attribute structure for a vector view of tensor indices.

```
typedef struct {
  vsip_offset offset;
  vsip_stride stride;
  vsip_length length;
  vsip_block_ti *block;
} vsip_vattr_ti;
```

Object type for a vector view of tensor indices.

```
struct vsip_vviewobject_ti;
typedef struct vsip_vviewobject_ti vsip_vview_ti;
```

Attribute structure for a vector view of booleans.

```
typedef struct {
  vsip_offset offset;
  vsip_stride stride;
  vsip_length length;
  vsip_block_bl *block;
} vsip_vattr_bl;
```

Object type for a vector view of booleans.

```
struct vsip_vviewobject_bl;
typedef struct vsip_vviewobject_bl vsip_vview_bl;
```

Attribute structure for a vector view of integer data.

```
typedef struct {
  vsip_offset offset;
  vsip_stride stride;
  vsip_length length;
  vsip_block_i *block;
} vsip_vattr_i;
```

Object type for a vector view of integer data.

```
struct vsip_vviewobject_i;
typedef struct vsip_vviewobject_i vsip_vview_i;
```

Attribute structure for a vector view of floating point data.

```
typedef struct {
  vsip_offset offset;
  vsip_stride stride;
  vsip_length length;
  vsip_block_f *block;
} vsip_vattr_f;
```

Object type for a vector view of floating point data.

```
struct vsip_vviewobject_f;
typedef struct vsip_vviewobject_f vsip_vview_f;
```

Complex Vector View

Attribute structure for a vector view of complex integer data.

```
typedef struct {
  vsip_offset offset;
  vsip_stride stride;
  vsip_length length;
  vsip_cblock_i *block;
} vsip_cvattr_i;
```

Object type for a vector view of complex integer data.

```
struct vsip_cvviewobject_i;
typedef struct vsip_cvviewobject_i vsip_cvview_i;
```

Attribute structure for a vector view of complex floating point data.

```
typedef struct {
  vsip_offset offset;
  vsip_stride stride;
  vsip_length length;
  vsip_cblock_f *block;
} vsip_cvattr_f;
```

Object type for a vector view of complex floating point data.

```
struct vsip_cvviewobject_f;
typedef struct vsip_cvviewobject_f vsip_cvview_f;
```

Matrix View

Indicates the *major* memory direction, along rows, or columns.

```
typedef enum {
  VSIP_ROW = 0,                    Row, C style
  VSIP_COL = 1                     Column, FORTRAN style
} vsip_major;
```

Real Matrix View Object

Attribute structure for a matrix view of booleans.

```
typedef struct {
  vsip_offset offset;
  vsip_stride col_stride;
  vsip_length col_length;
  vsip_stride row_stride;
  vsip_length row_length;
  vsip_block_bl *block;
} vsip_mattr_bl;
```

Object type for a matrix view of booleans.

```
struct vsip_mviewobject_bl;
typedef struct vsip_mviewobject_bl vsip_mview_bl;
```

Attribute structure for a matrix view of integer data.

```
typedef struct {
  vsip_offset offset;
  vsip_stride col_stride;
  vsip_length col_length;
  vsip_stride row_stride;
  vsip_length row_length;
  vsip_block_i *block;
} vsip_mattr_i;
```

Object type for a matrix view of integer data.

```
struct vsip_mviewobject_i;
typedef struct vsip_mviewobject_i vsip_mview_i;
```

Attribute structure for a matrix view of floating point data.

```
typedef struct {
  vsip_offset offset;
  vsip_stride col_stride;
  vsip_length col_length;
  vsip_stride row_stride;
  vsip_length row_length;
  vsip_block_f *block;
} vsip_mattr_f;
```

Object type for a matrix view of floating point data.

```
struct vsip_mviewobject_f;
typedef struct vsip_mviewobject_f vsip_mview_f;
```

Complex Matrix View Object

Object type for a matrix view of complex integer or floating point data.

```
struct vsip_cmviewobject_p;
typedef struct vsip_cmviewobject_p vsip_cmview_p;
```

Attribute structure for a matrix view of complex integer data.

```
typedef struct {
  vsip_offset offset;
  vsip_stride col_stride;
  vsip_length col_length;
  vsip_stride row_stride;
  vsip_length row_length;
  vsip_cblock_i *block;
} vsip_cmattr_i;
```

Object type for a matrix view of complex integer data.

```
struct vsip_cmviewobject_i;
typedef struct vsip_cmviewobject_i vsip_cmview_i;
```

Attribute structure for a matrix view of complex floating point data.

```
typedef struct {
  vsip_offset offset;
  vsip_stride col_stride;
  vsip_length col_length
  vsip_stride row_stride;
  vsip_length row_length;
  vsip_cblock_f *block;
} vsip_mattr_f;
```

Object type for a matrix view of complex floating point data.

```
struct vsip_cmviewobject_f;
typedef struct vsip_cmviewobject_f vsip_cmview_f;
```

Tensor View

Indicates the *major* memory direction for a tensor; C style with trailing dimension, or FORTRAN style with the leading dimension.

```
typedef enum {
  VSIP_TRAILING = 0,          Trailing dimension, C style
  VSIP_LEADING = 1            Leading Dimension, FORTRAN style
} vsip_tmajor;
```

Specifies a 2-D slice of a tensor.

```
typedef enum {
  VSIP_TMYX = 0,              Y– X Submatrix
  VSIP_TMZX = 1,              Z– X Submatrix
  VSIP_TMZY = 2               Z– Y Submatrix
} vsip_tmslice;
```

Specifies a 1-D slice of a tensor.

```
typedef enum {
  VSIP_TVX = 0,               X Subvector
  VSIP_TVY = 1,               Y Subvector
  VSIP_TVZ = 2                Z Subvector
} vsip_tvslice;
```

```
typedef enum {
  VSIP_TTRANS_NOP = 0,        No transpose
  VSIP_TTRANS_YX = 1,         Y – X transpose
  VSIP_TTRANS_ZY = 2,         Z – Y transpose
  VSIP_TTRANS_ZX = 3,         Z – X transpose
  VSIP_TTRANS_YXZY = 4,       Y – X & Z – Y transpose
  VSIP_TTRANS_YXZX = 5        Y – X & Z – X transpose
} vsip_ttrans;
```

Real Tensor View Object

Attribute structure for a tensor view of booleans.

```
typedef struct {
  vsip_offset offset;
  vsip_stride z_length;
  vsip_length z_stride;
  vsip_stride y_length;
  vsip_length y_stride;
  vsip_stride x_length;
  vsip_length x_stride;
  vsip_block_bl *block;
} vsip_tattr_bl;
```

Object type for a tensor view of booleans.

```
struct vsip_tviewobject_bl;
typedef struct vsip_tviewobject_bl vsip_tview_bl;
```

Attribute structure for a tensor view of integer data.

```
typedef struct {
  vsip_offset offset;
  vsip_stride z_length;
  vsip_length z_stride;
  vsip_stride y_length;
  vsip_length y_stride;
  vsip_stride x_length;
  vsip_length x_stride;
  vsip_block_i *block;
} vsip_tattr_i;
```

Object type for a tensor view of integer data.

```
struct vsip_tviewobject_i;
typedef struct vsip_tviewobject_i vsip_tview_i;
```

Attribute structure for a tensor view of floating point data.

```
typedef struct {
  vsip_offset offset;
  vsip_stride z_length;
  vsip_length z_stride;
  vsip_stride y_length;
  vsip_length y_stride;
  vsip_stride x_length;
  vsip_length x_stride;
  vsip_block_f *block;
} vsip_tattr_f;
```

Object type for a tensor view of floating point data.

```
struct vsip_tviewobject_f;
typedef struct vsip_tviewobject_f vsip_tview_f;
```

Complex Tensor View Object

Object type for a tensor view of complex integer or floating point data.

```
struct vsip_ctviewobject_p;
typedef struct vsip_ctviewobject_p vsip_ctview_p;
```

Attribute structure for a tensor view of complex integer data.

```
typedef struct {
  vsip_offset offset;
  vsip_stride z_length;
  vsip_length z_stride;
  vsip_stride y_length;
  vsip_length y_stride;
  vsip_stride x_length;
  vsip_length x_stride;
  vsip_cblock_i *block;
} vsip_ctattr_i;
```

Object type for a tensor view of complex integer data.

```
struct vsip_ctviewobject_i;
typedef struct vsip_ctviewobject_i vsip_ctview_i;
```

Attribute structure for a tensor view of complex floating point data.

```
typedef struct {
  vsip_offset offset;
  vsip_stride z_length;
  vsip_length z_stride;
  vsip_stride y_length;
  vsip_length y_stride;
  vsip_stride x_length;
  vsip_length x_stride;
  vsip_cblock_f *block;
} vsip_ctattr_f;
```

Object type for a tensor view of complex floating point data.

```
struct vsip_ctviewobject_f;
typedef struct vsip_ctviewobject_f vsip_ctview_f;
```

Signal Processing

Hint for how to optimize an object.

```
typedef enum {
  VSIP_ALG_TIME = 0,          Minimize execution time
  VSIP_ALG_SPACE = 1,         Minimize memory required
  VSIP_ALG_NOISE = 2          Minimize computational noise, Maximize Accuracy
} vsip_alg_hint;
```

For filter and convolution kernels, specifies the symmetry of the kernel.

```
typedef enum {
  VSIP_NONSYM = 0,            Non-symmetric
  VSIP_SYM_EVEN_LEN_ODD = 1,  (Even) Symmetric, odd length
  VSIP_SYM_EVEN_LEN_EVEN = 2  (Even) Symmetric, even length
} vsip_symmetry;
```

For filters, convolutions, and correlations; specifies the region over which the output result is computed.

```
typedef enum {
  VSIP_SUPPORT_FULL = 0,      Maximum region
  VSIP_SUPPORT_SAME = 1,      Input and output same size
  VSIP_SUPPORT_MIN = 2        Region without zero extending the kernel, reference
} vsip_support_region;
```

Signal Processing – FFT

Enumerated type to indicate forward or inverse FFT.

```
typedef enum {
  VSIP_FFT_FWD = -1,          Forward
  VSIP_FFT_INV = 1            Inverse (or reverse)
} vsip_fft_dir;
```

Enumerated type to indicate in place or out of place computation.

```
typedef enum {
  VSIP_FFT_IP = 0,            In-Place
  VSIP_FFT_OP = 1             Out-of-Place
} vsip_fft_place;
```

Attribute structure for a 1-D FFT object.

```
typedef struct {
  vsip_scalar_vi input;       Input length
  vsip_scalar_vi output;      Output length
  vsip_fft_place place;       In-Place/Out-of-Place
  vsip_scalar_p scale;        Scale factor
  vsip_fft_dir dir;           Forward or Inverse
} vsip_fft_attr_p;
```

Attribute structure for a multiple 1-D FFT.

```
typedef struct {
  vsip_scalar_mi input;          Input size, M by N
  vsip_scalar_mi output;         Output size, P by Q
  vsip_fft_place place;          In-Place/Out-of-Place
  vsip_scalar_p scale;           Scale factor
  vsip_fft_dir dir;              Forward or Inverse
  vsip_major major;              By Row/Col
} vsip_fftm_attr_p;
```

Attribute structure for a 2-D FFT.

```
typedef struct {
  vsip_scalar_mi input;          Input size, M by N
  vsip_scalar_mi output;         Output size, P by Q
  vsip_fft_place place;          In-Place/Out-of-Place
  vsip_scalar_p scale;           Scale factor
  vsip_fft_dir dir;              Forward or Inverse
} vsip_fft2d_attr_p;
```

Attribute structure for a 3-D FFT.

```
typedef struct {
  vsip_scalar_ti input;          Input size, P by M by N
  vsip_scalar_ti output;         Output size, S by Q by R
  vsip_fft_place place;          In-Place/Out-of-Place
  vsip_scalar_p scale;           Scale factor
  vsip_fft_dir dir;              Forward or Inverse
} vsip_fft3d_attr_p;
```

Object type for a 1-D FFT.

```
struct vsip_fftobject_p;
typedef struct vsip_fftobject_p vsip_fft_p;
```

Object type for a multiple 1-D FFT.

```
struct vsip_fftmobject_p;
typedef struct vsip_fftmobject_p vsip_fftm_p;
```

Object type for a 2-D FFT.

```
struct vsip_fft2dobject_p;
typedef struct vsipfft2d_object_p vsip_fft2d_p;
```

Object type for a 3-D FFT.

```
struct vsip_fft3dobject_p;
typedef struct vsip_fft3dobject_p vsip_fft3d_p;
```

Signal Processing – Filtering

Object type for an FIR filter.

```
struct vsip_firobject_p;
typedef struct vsip_firobject_p vsip_fir_p;
```

Object type for a complex FIR filter.

```
struct vsip_cfirobject_p;
typedef struct vsip_cfirobject_p vsip_cfir_p;
```

Object type for an IIR filter.

```
struct vsip_iirobject_p;
typedef struct vsip_iirobject_p vsip_iir_p;
```

Enumerated type indicating if state information should be saved in the filter object.

```
Typedef enum {
  VSIP_STATE_NO_SAVE = 1,          Don't save state, single call filter
  VSIP_STATE_SAVE = 2              Save state for continuous filtering
} vsip_obj_state;
```

Attribute structure for a real FIR filter.

```
typedef struct {
  vsip_scalar_vi kernel_len;       Kernel length
  vsip_symmetry symm;              Kernel symmetry
  vsip_scalar_vi in_len;           Filter input segment length
  vsip_scalar_vi out_len;          Filter output segment length
  vsip_length decimation;          Decimation factor
  vsip_obj_state state;            Save state information
} vsip_fir_attr_f;
```

Attribute structure for a complex FIR filter.

```
typedef struct {
  vsip_scalar_vi kernel_len;       Kernel length
  vsip_symmetry symm;              Kernel symmetry
  vsip_scalar_vi in_len;           Filter input segment length
  vsip_scalar_vi out_len;          Filter output segment length
  vsip_length decimation;          Decimation factor
  vsip_obj_state state;            Save state information
} vsip_cfir_attr_f;
```

Attribute structure for an IIR filter.

```
typedef struct {
  vsip_length n2nd;                Number of 2nd order sections
  vsip_scalar_vi seg_len;          Filter input/output segment length
  vsip_obj_state state;            Save state information
} vsip_iir_attr_f;
```

Signal Processing – Convolution

Object type for a 1-D convolution.

```
struct vsip_conv1dobject_p;
typedef struct vsip_conv1dobject_p vsip_conv1d_p;
```

Object type for a 2-D convolution.

```
struct vsip_conv2dobject_p;
typedef struct vsip_conv2dobject_p vsip_conv2d_p;
```

Attribute structure for a 1-D convolution.

```
typedef struct {
  vsip_scalar_vi kernel_len;          Kernel length, M
  vsip_symmetry symm;                 Kernel symmetry
  vsip_scalar_vi data_len;            Data input length
  vsip_support_region support;        Output region of support
  vsip_scalar_vi out_len;             Output length
  vsip_length decimation;             Output decimation factor, D
} vsip_conv1d_attr_f;
```

Attribute structure for a 2-D convolution.

```
typedef struct {
  vsip_scalar_mi kernel_size;         Kernel size, M by N
  vsip_symmetry symm;                 Kernel symmetry
  vsip_scalar_mi in_size;             Data input size, P by Q
  vsip_support_region support;        Output region of support
  vsip_scalar_mi out_size;            Output size, S by T
  vsip_length decimation;             Output decimation factor, D
} vsip_conv2d_attr_f;
```

Signal Processing – Correlation

Object type for a 1-D correlation.

```
struct vsip_corr1dobject_p;
typedef struct vsip_corr1dobject_p vsip_corr1d_p;
```

Object type for a 2-D correlation.

```
struct vsip_corr2dobject_p;
typedef struct vsip_corr2dobject_p vsip_corr2d_p;
```

Object type for a complex 1-D correlation.

```
struct vsip_ccorr1dobject_p;
typedef struct vsip_ccorr1dobject_p vsip_ccorr1d_p;
```

Object type for a complex 2-D correlation.

```
struct vsip_ccorr2dobject_p;
typedef struct vsip_ccorr2dobject_p vsip_ccorr2d_p;
```

Enumerated type to indicate calculation of biased or unbiased correlation estimate.

```
typedef enum {
  VSIP_BIASED = 0,                    Biased
  VSIP_UNBIASED = 1                   Unbiased
} vsip_bias;
```

Attribute structure for a 1-D correlation.

```
typedef struct {
  vsip_scalar_vi ref_len;              Reference length
  vsip_scalar_vi data_len;             Data input length
  vsip_support_region support;         Output region of support
  vsip_scalar_vi lag_len;              Output (lags) length
} vsip_corr1d_attr_p;
```

Attribute structure for a 1-D complex correlation.

```
typedef struct {
  vsip_scalar_vi ref_len;              Reference length
  vsip_scalar_vi data_len;             Data input length
  vsip_support_region support;         Output region of support
  vsip_scalar_vi lag_len;              Output (lags) length
} vsip_ccorr1d_attr_p;
```

Attribute structure for a 2-D correlation.

```
typedef struct {
  vsip_scalar_mi ref_size;             Reference size, M by N
  vsip_scalar_mi data_size;            Data input size, P by Q
  vsip_support_region support;         Output region of support
  vsip_scalar_mi out_size;             Output size, S by T
} vsip_corr2d_attr_p;
```

Attribute structure for a 2-D complex correlation.

```
typedef struct {
  vsip_scalar_mi ref_size;             Reference size, M by N
  vsip_scalar_mi data_size;            Data input size, P by Q
  vsip_support_region support;         Output region of support
  vsip_scalar_mi out_size;             Output size, S by T
} vsip_ccorr2d_attr_p;
```

Signal Processing – Histogram

Histogram accumulate option.

```
typedef enum {
  VSIP_HIST_RESET = 1,                 Histogram Reset
  VSIP_HIST_ACCUM = 2                  Histogram Accumulate
} vsip_hist_opt;
```

Linear Algebra

Matrix transformation operation.

```
typedef enum {
  VSIP_MAT_NTRANS = 0,        No transformation
  VSIP_MAT_TRANS = 1,         Matrix Transpose
  VSIP_MAT_HERM = 2,          Matrix Hermitian (Conjugate Transpose)
  VSIP_MAT_CONJ = 3           Matrix Conjugate
} vsip_mat_op;
```

Solvers

LU

Attribute structure for an LU matrix decomposition object.

```
typedef struct {
  vsip_length n;              Matrix size is N by N
} vsip_lu_attr_f;
```

Attribute structure for a complex LU matrix decomposition object.

```
typedef struct {
  vsip_length n;              Matrix size is N by N
} vsip_clu_attr_f;
```

Object type for an LU matrix decomposition.

```
struct vsip_luobject_f;
typedef struct vsip_luobject_f vsip_lu_f;
```

Object type for a complex LU matrix decomposition.

```
struct vsip_cluobject_f;
typedef struct vsip_cluobject_f vsip_clu_f;
```

Cholesky

Attribute structure for a Cholesky matrix object.

```
typedef struct {
  vsip_length n;              Matrix size is N by N
} vsip_chol_attr_f;
```

Attribute structure for a complex Cholesky matrix object.

```
typedef struct {
  vsip_length n;              Matrix size is N by N
} vsip_cchol_attr_f;
```

Object type for a Cholesky matrix system.

```
struct vsip_choldobject_f;
typedef struct vsip_choldobject_f vsip_chol_f;
```

Object type for a complex Cholesky matrix system.

```
struct vsip_ccholdobject_f;
typedef struct vsip_ccholdobject_f vsip_cchol_f;
```

QR

Enumerated type to indicate if op(Q) is applied on the left or right

```
typedef enum {
  VSIP_MAT_LSIDE = 0,          Left side
  VSIP_MAT_RSIDE = 1           Right side
} vsip_mat_side;
```

Enumerated type to indicate if the matrix Q is retained

```
typedef enum {
  VSIP_QRD_NOSAVEQ = 0,        Do not save Q
  VSIP_QRD_SAVEQ = 1,          Save Q
  VSIP_QRD_SAVEQ1 = 2          Save Skinny Q
} vsip_qrd_qopt;
```

Selects between the covariance and linear least squares problem

```
typedef enum {
  VSIP_COV = 0,                Solve a covariance linear system
problem
  VSIP_LLS =1                  Solve a linear least squares problem
} vsip_qrd_prob;
```

Attribute structure for a QRD matrix object.

```
typedef struct {
  vsip_length m;               Input matrix is M by N
  vsip_length n;               Input matrix is M by N
  vsip_qrd_opt Qopt;           Matrix Q is saved/not saved
} vsip_qr_attr_f;
```

Attribute structure for a complex QRD matrix object.

```
typedef struct {
  vsip_length m;               Input matrix is M by N
  vsip_length n;               Input matrix is M by N
  vsip_qrd_opt Qopt;           Matrix Q is saved/not saved
} vsip_cqr_attr_f;
```

Attribute structure for a QR matrix decomposition object.

```
struct vsip_qrobject_f;
typedef struct vsip_qrobject_f vsip_qr_f;
```

Attribute structure for a complex QR matrix decomposition object.

```
struct vsip_cqrobject_f;
typedef struct vsip_cqrobject_f vsip_cqr_f;
```

SVD

Enumerated type to indicate

```
typedef enum {
  VSIP_SVD_UVNOS = 0,          No columns/rows of U/V are computed
  VSIP_SVD_UVFULL = 1,         All columns/rows of U/V are computed
  VSIP_SVD_UVPART = 2          First min{M, N} columns/rows of U/V are computed
} vsip_svd_uv;
```

Attribute structure for a real SVD matrix object.

```
typedef struct {
  vsip_length m;                    Input matrix is M by N
  vsip_length n;                    Input matrix is M by N
  vsip_svd_uv Usave;                Columns of U computed
  vsip_svd_uv Vsave;                Columns of V computed
} vsip_sv_attr_f;
```

Attribute structure for a complex SVD matrix object.

```
typedef struct {
  vsip_length m;                    Input matrix is M by N
  vsip_length n;                    Input matrix is M by N
  vsip_svd_uv Usave;                Columns of U computed
  vsip_svd_uv Vsave;                Columns of V computed
} vsip_csv_attr_f;
```

Attribute structure for an SVD matrix object.

```
struct vsip_svobject_f;
typedef struct vsip_svobject_f vsip_sv_f;
```

Attribute structure for a complex SVD matrix object.

```
struct vsip_csvdobject_f;
typedef struct vsip_csvdobject_f vsip_csv_f;
```

SUPPORT FUNCTIONS

Support - Introduction

This section covers the support functions needed by VSIPL. These support functions include routines to initialize and to finalize VSIPL function usage, as well as to create, destroy, and manipulate VSIPL block and view objects. The support functions are divided into five sections, describing library initialization and finalization functions, array and block object functions, vector view functions, matrix view functions, and tensor view functions. In each of the latter four sections, the functions may be divided into creation, destruction, and manipulation functions.

Library Initialization and Finalization

Before any other VSIPL functions can be called, the VSIPL library must be initialized by a call to `vsip_init`. Conversely, any program that uses VSIPL and that terminates must call `vsip_finalize` before terminating.

To support third party libraries that use VSIPL without the knowledge of the application programmer, calls to `vsip_init` and `visp_finalize` functions may be nested. In addition, sequences of `vsip_init` and `vsip_finalize` pairs may occur in a given program. The following program is legal:

```
/* Example of nesting and sequence of init/finalize */
#include "vsip.h"
int main()
{
    /* Nested vsip_init and vsip_finalize */
    vsip_init ((void *)0);
    vsip_init ((void *)0);
    vsip_finalize ((void *)0);
    vsip_finalize ((void *)0);
    /* No VSIPL calls permitted here….*/
    vsip_init ((void *)0);
    /* A second appearance of VSIPL calls */
    vsip_finalize ((void *)0);
    return 0;
}
```

If `vsip_init` and `vsip_finalize` functions are called multiple times, then the calls must be made in pairs. The intermediate `vsip_init` calls (after the first) and the intermediate `vsip_finalize` calls (before the one corresponding to the first `vsip_init` call) may have little or no effect. If the VSIPL library has not been initialized, or has been terminated, no calls to VSIPL functions other than `vsip_init` are allowed.

The user must destroy all VSIPL objects before calling `vsip_finalize`. In the case of nested calls to `vsip_init` and `vsip_finalize`, all VSIPL objects must be destroyed before the outermost call to `vsip_finalize`.

Object Creation and Destruction

Functions to create and destroy each particular type of object are included. A *block* is typically created first, followed by one or more *views* of the block. Every VSIPL object that is created must eventually be destroyed. All views on a block should be destroyed before the block is destroyed.

Convenience functions are included to create both a *block* and a *view* of the *block* with a single call. These functions return the *view*. The *view* encompasses *all* the data in the *block,* and contains the *block object* pointer as an attribute. Convenience functions are also included to destroy a *view* and a *block* together when the view in question is the only one that references the *block*. This function is the dual of the view creation function. For the vector view, these convenience functions are called `vsip_vcreate_p and vsip_valldestroy_p`.

Block Object Manipulation

In order to create a block, a `vsip_dblockcreate_p` or `vsip_dblockbind_p` function is used. The `vsip_dblockcreate_p` functions create a VSIPL block. The `vsip_dblockbind_p` functions create a user block.

Blocks do not have attributes that can be directly manipulated, but they exist in either the *released* or *admitted* state, as explained in the introduction. A released block may be admitted to VSIPL, at which time the block functions like any other VSIPL block. When the application programmer wishes to access the data directly, a *block* in an *admitted* state must first be *released* from VSIPL.

The purpose of defining an admitted state is to provide the VSIPL implementation the opportunity to operate on the VSIPL object in any manner necessary for optimum performance without making such optimizations visible to the application programmer. Potential optimizations include, but are not limited to, deferred execution, explicit management of a hierarchical memory system, and use of system specific resources.

VSIPL effectively owns the data in an admitted block. The purpose of release is to give ownership of the data back to the application programmer. When an admitted block is released, all operations on the data associated with that block must be completed before the release function returns the block to the released state.

A *user block* may be admitted and released multiple times during the application, and it is possible that the data in the associated user data array may not be required by the application during any individual admit or release operation. To provide the implementor with an opportunity for further optimization, the admit and release function each provide a boolean update flag. If this flag is false then the data need not be maintained during a particular admit or release operation.

A *VSIPL block* is one created directly by VSIPL using a VSIPL create function which allocates memory for the *block object* and the *data array*. A VSIPL block is created in the admitted state and may not be released. To access this data the application programmer must use a VSIPL access function (such as get or put), or must copy the data to a block which may be released. Only blocks bound to a user data array (user blocks) may be released.

View Object Manipulation

Vector, matrix, and tensor view objects allow the user to treat data in a block as one, two, or three-dimensional objects (respectively). All view objects have four categories of attributes: the *block* that they are bound to, an *offset* from the start of the block, and a *stride* and *length* for each dimension of the view object. The *block* attribute can be read by the user but not altered after the view is created. Functions are provided for the user to read and set the other view attributes.

VSIPL provides functions that allow a view to be created as a subset of another view. For higher-dimensional view objects (matrices and tensors), additional functions provide the ability to view part of the data set as a lower-dimensional object.

Complex Blocks and Views

As described in the introduction, a complex data array is not necessarily an array of complex scalars. For VSIPL data, the internal behavior of complex objects is hidden from the application programmer by the implementation. In the case of user data complex arrays are defined as either interleaved, which is sequential memory locations of real/imaginary pairs; or as split, which is real in sequential order in one section of memory, and imaginary in matching sequential order in another section of memory. For split complex, the memory for the real part may not necessarily be contiguous with the memory of the imaginary part. Upon admission of a user complex block to VSIPL, the layout of the data is no longer visible and is implementation dependent. Upon release of a user complex block from VSIPL the complex layout is the same as when the block was initially created.

The stride, length and offset of complex data are in terms of a complex element. The stride and offset of real or imaginary views of complex data are vendor dependent and must be probed using get attribute functions if the information is needed.

For admitted VSIPL objects the data array is controlled by the implementation. To manipulate complex data VSIPL functions provided for that purpose must be used.

Real Views of Real or Imaginary Parts of Complex Views

Functions are available which allow one to retrieve a real view of the real or imaginary portion of a complex view. The returned view acts like any other real view. It is possible to make subviews of it, query it to obtain its real block and attributes, and to use the attribute information to bind other views to the space encompassed by the real block. These views have the following special conditions:

1. The attribute information (block, offset, stride, and length) of a real view obtained from a complex view are vendor dependent.
2. The underlying data space of the real view is owned by the complex block of the complex view that the real view was derived from.
3. It is an error to destroy the block of any real view derived from a complex view. The view is destroyed in the normal manner, but the block bound to it is destroyed by the implementation when the complex block is destroyed.
4. Real blocks derived from complex views bound to user data may not be directly admitted or released. Such blocks are admitted or released when the complex block bound to the user data is admitted or released.
5. Using a block find on a real block derived from a complex block bound to user data will produce a *null* value.

Examples

Examples in the Support section are, for the most part, code fragments. For complete examples, see the chapter on Vector and Elementwise operations.

Initialization/Finalization Operations

Two functions, `vsip_init` and `vsip_finalize`, are provided to control the initialization and finalization of VSIPL. The use of these initialization and finalization functions is required for all VSIPL programs. Programs that never terminate (e.g., periodic loops) need never invoke the finalization function, but all programs that terminate must first call `vsip_finalize`. All programs must use the `vsip_init` function before calling other VSIPL functions. These functions may be nested in order to support third party and nested libraries. It is correct to initialize and finalize VSIPL an arbitrary number of times during the lifetime of a program.

`vsip_`**`init`**	Initialization Function
`vsip_`**`finalize`**	Finalization (or termination) Function

vsip_init Initialize the Library

Provides initialization, allowing the implementation to allocate and set any global state, and prepare to support the use of VSIPL functionality by the user.

Functionality

This required function informs the VSIPL library that library initialization is requested, and that other VSIPL functions will be called. Each implementation does as much or as little internally as is needed in order to support VSIPL services. Some implementations may do little or nothing at this stage, while others may do quite a bit of resource management. All programs must call this function at least once. The example illustrates a canonical form of a VSIPL program. It may be called multiple times as well, with corresponding calls to `vsip_finalize` to create nested pairs of initialization/termination. Only the final `vsip_finalize` call will actually deinitialize the library. Intermediate calls to `vsip_init` may have little or no effect, but support easy program/library development through compositional programming, where the user may not even know that a library itself invokes VSIPL. The single `void*` argument is reserved for future purposes. The NULL pointer should be passed to it for VSIPL 1.0 compliance.

Prototypes

```
int vsip_init(
                        void *);
```

Arguments

There is no argument value to be passed other than `(void *)0` at this point. This is an argument reserved for future purposes.

Return Value

Returns 0 if the initialization succeeded, and non-zero otherwise.

Restrictions

This function may be called anytime during the execution of the program.

Errors

Notes/References

All programs must use the initialization function (`vsip_init`) before calling any other VSIPL functions.

Unsuccessful initialization of the library is not an error. It is always signaled via the function's return value, and should always be checked by the application.

Several modes of usage of the initialize/terminate are supported: nested (init/init/code/finalize/finalize), sequences (init/code/finalize … init/code/finalize), and generalizations of these.

Examples

```
/* Canonical form of a VSIPL program */
#include "vsip.h"

int main()
{
   /* no VSIPL calls except: vsip_init() at this stage */
   vsip_init((void *)0);

   /* all VSIPL calls here,
   including pairs of vsip_init() and vsip_finalize()... */
```

```
    vsip_finalize((void *)0);

    /* no VSIPL calls until another vsip_init */
    return 0;
}
```

See Also

vsip_finalize

vsip_finalize Terminate the Library

Provides cleanup and releases resources used by VSIPL (if the last of a nested series of calls), allowing an implementation to guarantee that any resources allocated by `vsip_init` are no longer in use after the call is complete.

Functionality

This required function informs the VSIPL library that it is not being used anymore by a program, so that all needed global state and hardware state can be returned. Each implementation does as much or as little internally as is needed in order to support cleanup of VSIPL services. Some implementations may do little or nothing at this stage, while others may do quite a bit of resource management. All programs must call this function at least once if they terminate. If the program does terminate, the last VSIPL function called must be an outermost `vsip_finalize`. Because nested `vsip_init`'s are supported, so are nested `vsip_finalize`'s. The user must explicitly destroy all VSIPL objects before calling this function if this is an "outermost" `vsip_finalize`. When nesting initializations, there is no need to destroy all objects prior to calling this function, but the user is obliged to keep track of the nesting depth if programs are written in such a manner.

Prototypes

```
int vsip_finalize(
                        void *);
```

Arguments

There is a reserved argument, which must have the value `(void *)0` for VSIPL 1.0 compliance.

Return Value

Returns 0 if the finalization succeeded, and non-zero otherwise.

Non-outermost `vsip_finalize`'s always return "success."

Restrictions

This function may only be called if a previous `vsip_init` call has been called, with no previous corresponding `vsip_finalize`.

Errors

An outermost `vsip_finalize` function produces an error if there are any VSIPL objects not destroyed.

Notes/References

The user program is always responsible for returning resources it is no longer using by destroying VSIPL objects. An outermost finalization function (`vsip_finalize`) will return resources that it allocated previously with `vsip_init`. Non-outermost `vsip_finalize`'s always return zero (success).

Several modes of usage of the initialize/terminate are supported: nested (init/init/code/finalize/finalize), sequences (init/code/finalize … init/code/finalize), and generalizations of these.

Examples

See example for `vsip_init`.

See Also

`vsip_init`

Array and Block Object Functions

This section covers the functions needed to create, destroy, and manipulate VSIPL blocks.

A VSIPL *block* includes state information about the status of the *blocks* data (*admitted* or *released*), and the type of data arrays associated with the block (*user* and/or *VSIPL data arrays*). Blocks of type `vsip_block_p` (a real block) also must contain state information to indicate if they are a *derived block* (derived from a complex block). A block of type `vsip_cblock_p` must contain information about any real block derived from it. In addition, in *development mode,* the block includes information about the size of the data array the block references, and the number of vector, matrix or tensor objects that are bound to the block.

`vsip_dblockadmit_p`	Block Admit
`vsip_blockbind_p`	Memory Block Bind
`vsip_cblockbind_p`	Complex Memory Block Bind
`vsip_dblockcreate_p`	Memory Block Create
`vsip_dblockdestroy_p`	Memory Block Destroy
`vsip_blockrebind_p`	Block Rebind
`vsip_blockfind_p`	Memory Block Find
`vsip_cblockfind_p`	Memory Complex Block Find
`vsip_cblockrebind_p`	Complex Block Rebind
`vsip_blockrelease_p`	Block Release
`vsip_cblockrelease_p`	Complex Block Release
`vsip_complete`	Complete Deferred Execution
`vsip_cstorage_p`	Complex Storage
`vsip_cstorage`	Deprecated; see Notes to Implementors

vsip_*d*blockadmit_*p* Block Admit

Admit a VSIPL block for VSIPL operations.

Functionality

Admits a VSIPL block, `vsip_dblock_p`, for VSIPL operations on the associated views. Admission changes the ownership of the *user data array* to VSIPL, and the user should not operate on the *data array* after the *block* is admitted. It returns non-zero if the admission fails. A true *update* flag indicates that the data in the block shall be made consistent with the user-specified data array. If the *update* flag is false the data in the block is implementation dependent and the user should consider the block to contain undefined data.

Prototypes

```
int vsip_blockadmit_f(
                      vsip_block_f *block,
                      vsip_scalar_bl update);
int vsip_cblockadmit_f(
                      vsip_cblock_f *block,
                      vsip_scalar_bl update);
int vsip_blockadmit_i(
                      vsip_block_i *block,
                      vsip_scalar_bl update);
int vsip_cblockadmit_i(
                      vsip_cblock_i *block,
                      vsip_scalar_bl update);
int vsip_blockadmit_bl(
                      vsip_block_bl *block,
                      vsip_scalar_bl update);
int vsip_blockadmit_vi(
                      vsip_block_vi *block,
                      vsip_scalar_bl update);
int vsip_blockadmit_mi(
                      vsip_block_mi *block,
                      vsip_scalar_bl update);
int vsip_blockadmit_ti(
                      vsip_block_ti *block,
                      vsip_scalar_bl update);
```

Arguments

`*block`	Pointer to a block object.
`update`	Boolean flag where *true* indicates that the data array values must be maintained during the state change.

Return Value

Returns zero on success and non-zero on failure.

Restrictions

Errors

The arguments must conform to the following:

1. The block object must be valid.

Notes/References

It is **not** an error to admit a block that is already in the *admitted state*.

The intent of using a false *update* flag is that if the data in the user array is not needed, then there is no need to force consistency between the block object's data and the user-specified data array with a potential copy operation.

Examples

See example with `vsip_dblockrelease_p`.

See Also

`vsip_cblockbind_p`, `vsip_blockbind_p`, `vsip_blockadmit_p`, `vsip_cblockadmit_p`, `vsip_dblockrelease_p`, `vsip_blockfind_p`, `vsip_cblockfind_p`, and `vsip_dblockdestroy_p`.

vsip_blockbind_*p* Memory Block Bind

Create and bind a VSIPL block to user allocated (*user data array*) memory.

Functionality

Creates a real VSIPL block object, `vsip_block_p`, and binds the block object to a user-defined *user data array*. The data array should contain at least *N* `vsip_scalar_p` elements. The function returns a pointer to the block object. The block is created in the *released* state and must be *admitted* to VSIPL before calling VSIPL functions that operate on the data.

Prototypes

```
vsip_block_f *vsip_blockbind_f(
                    vsip_scalar_f* const data,
                    vsip_length N,
                    vsip_memory_hint hint);
vsip_block_i *vsip_blockbind_i(
                    vsip_scalar_i* const data,
                    vsip_length N,
                    vsip_memory_hint hint);
vsip_block_bl *vsip_blockbind_bl(
                    vsip_scalar_bl* const data,
                    vsip_length N,
                    vsip_memory_hint hint);
vsip_block_vi *vsip_blockbind_vi(
                    vsip_scalar_vi* const data,
                    vsip_length N,
                    vsip_memory_hint hint);
vsip_block_mi *vsip_blockbind_mi (
                    vsip_scalar_vi* const data,
                    vsip_length N,
                    vsip_memory_hint hint);
vsip_block_ti *vsip_blockbind_ti (
                    vsip_scalar_vi* const data,
                    vsip_length N,
                    vsip_memory_hint hint);
```

Arguments

`*data` Pointer to a data array of contiguous memory containing at least *N* `vsip_scalar_p` elements.

`N` Number of elements, of user data array, to which a *user block*, `vsip_block_p`, is bound.

`hint` Memory hint

```
typedef enum {
  VSIP_MEM_NONE = 0,                 No hint
  VSIP_MEM_RDONLY = 1,               Read Only
  VSIP_MEM_CONST = 2,                Constant
  VSIP_MEM_SHARED = 3,               Shared
  VSIP_MEM_SHARED_RDONLY = 4,        Shared, Read Only
  VSIP_MEM_SHARED_CONST = 5          Shared, Constant
} vsip_memory_hint;
```

Return Value

Returns a pointer of type `vsip_block_p`, or returns *null* if the block bind fails.

Restrictions

Errors

The arguments must conform to the following:

1. The data array size, *N*, must be greater than zero.
2. The memory *hint* must be a valid member of the `vsip_memory_hint` enumeration.

Notes/References

It is acceptable to bind a block to a *null* pointer for initialization purposes. However, it must be bound to a non-*null* pointer before it can be admitted.

Examples

To create a block and bind a *user data array* (memory) large enough to hold an *M* by *N* matrix of type `double`:

```
#include <vsip.h>
…
vsip_scalar_d A[M * N];
vsip_block_d *Ablock = vsip_blockbind_d(A, M * N, VSIP_MEM_NONE);
if (NULL == Ablock) error("Bind of A to Ablock failed");
```

See Also

`vsip_dblockcreate_p`, `vsip_blockfind_p`, `vsip_blockrebind_p`, `vsip_blockadmit_p`, `vsip_blockrelease_p`, `vsip_blockdestroy_p`, and `vsip_cblockbind_p`.

vsip_cblockbind_*p* Complex Memory Block Bind

Create and bind a VSIPL complex block to user allocated (*user data array*) memory.

Functionality

Creates a complex VSIPL block object, `vsip_cblock_p`, and binds the complex block object to either a single user-defined *user data array*, or to two user-defined data arrays. In the case of a single data array, the array must contain 2*N* `vsip_scalar_p` elements. For two data arrays, each array should contain *N* `vsip_scalar_p` elements. The block is created in the *released* state and must be *admitted* to VSIPL before calling VSIPL functions that operate on the data.

Prototypes

```
vsip_cblock_f *vsip_cblockbind_f(
                    vsip_scalar_f* const data1,
                    vsip_scalar_f* const data2,
                    vsip_length N,
                    vsip_memory_hint hint);
vsip_cblock_i *vsip_cblockbind_i(
                    vsip_scalar_i* const data1,
                    vsip_scalar_i* const data2,
                    vsip_length N,
                    vsip_memory_hint hint);
```

Arguments

`*data1`

If `data2` is *null*, then `data1` is a pointer to a data array of contiguous memory containing at least 2*N* `vsip_scalar_p` elements. The even elements of the data array contain the real part values, and the odd elements contain the imaginary part values. The data are stored in *interleaved complex* form. Note that the first element is considered to be even because index values start at zero.

If `data2` is not *null*, then `data1` is a pointer to a data array of contiguous memory containing at least *N* `vsip_scalar_p` elements. The data array contains the real part values. The data are stored in *split complex* form.

`*data2`

If `data2` is *null*, then the data are stored in *interleaved complex* form.

If `data2` is not *null*, then it is a pointer to a data array of contiguous memory containing at least *N* `vsip_scalar_p` elements. The data array contains the imaginary part values. The data are stored in *split complex* form.

`N`

Number of complex elements, of a *user data array,* to which a *user block* of type `vsip_cblock_p` is bound.

`hint` Memory hint

```
typedef enum {
  VSIP_MEM_NONE = 0,                  No hint
  VSIP_MEM_RDONLY = 1,                Read Only
  VSIP_MEM_CONST = 2,                 Constant
  VSIP_MEM_SHARED = 3,                Shared
  VSIP_MEM_SHARED_RDONLY = 4,         Shared, Read Only
  VSIP_MEM_SHARED_CONST = 5           Shared, Constant
} vsip_memory_hint;
```

Return Value

Returns a pointer of type `vsip_cblock_p`, or returns *null* if the block bind fails.

Restrictions

Errors

The arguments must conform to the following:

1. The data array size, *N*, must be greater than zero.
2. The memory *hint* must be a valid member of the `vsip_memory_hint` enumeration.
3. The `data1` pointer must be valid – non-*null* if the `data2` pointer is non-*null*.

Notes/References

It is acceptable to bind a block to a *null* pointer for initialization purposes. However, it must be bound to a non-*null* pointer before it can be admitted.

Complex data in the *released* state is treated as either *interleaved* or *split* as described above. A single *user data array* is used for storing complex data in the *interleaved* form. Two (identically sized) *user* data arrays, one for the real part and one for imaginary part, are used for storing complex data in the *split* form. The function `vsip_cstorage` will return an indicator of the desired storage format of the particular implementation. However, either storage format will work once *admitted* to VSIPL.

Examples

To create a block and bind *user data array* (memory) large enough to hold an *M* by *N* split complex matrix of type `double`:

```
#include <vsip.h>
…
vsip_scalar_d Ai[M*N], Aq[M*N];
vsip_cblock_d *Ablock = vsip_cblockbind_d(Ai, Aq, M*N, VSIP_MEM_NONE);
if (NULL == Ablock) error("Bind of A to Ablock failed");
```

See Also

`vsip_dblockcreate_p`, `vsip_cblockfind_p`, `vsip_cblockrebind_p`, `vsip_cblockadmit_p`, `vsip_cblockrelease_p`, `vsip_cblockdestroy_p`, `vsip_blockbind_p`, and `vsip_cstorage_p`.

vsip_*d*blockcreate_*p* Memory Block Create

Creates a VSIPL block and binds a (VSIPL allocated) data array (memory) to it.

Functionality

Creates an *admitted* VSIPL block object (`vsip_dblock_p`) and allocates data array memory (*VSIPL data*) for *N* elements. The size of the data array is at least `N*sizeof(vsip_scalar_p)` bytes for real data, or `2*N*sizeof(vsip_scalar_p)` bytes for complex data. The function binds the block object to the allocated data memory and returns a pointer to the block object.

Data arrays created using `vsip_dblockcreate_p` can only be accessed using VSIPL functions. Information that would allow direct manipulation, such as a pointer to the data array, is not available.

Prototypes

```
vsip_block_f *vsip_blockcreate_f(
                     vsip_length N,
                     vsip_memory_hint hint);
vsip_block_i *vsip_blockcreate_i(
                     vsip_length N,
                     vsip_memory_hint hint);
vsip_cblock_f *vsip_cblockcreate_f(
                     vsip_length N,
                     vsip_memory_hint hint);
vsip_cblock_i *vsip_cblockcreate_i(
                     vsip_length N,
                     vsip_memory_hint hint);
vsip_block_bl *vsip_blockcreate_bl(
                     vsip_length N,
                     vsip_memory_hint hint);
vsip_block_vi *vsip_blockcreate_vi(
                     vsip_length N,
                     vsip_memory_hint hint);
vsip_block_mi *vsip_blockcreate_mi(
                     vsip_length N,
                     vsip_memory_hint hint);
vsip_block_ti *vsip_blockcreate_ti(
                     vsip_length N,
                     vsip_memory_hint hint);
```

Arguments

`N` Number of elements to allocate for the data array.

`hint` Memory hint

```
typedef enum {
  VSIP_MEM_NONE = 0,                  No hint
  VSIP_MEM_RDONLY = 1,                Read Only
  VSIP_MEM_CONST = 2,                 Constant
  VSIP_MEM_SHARED = 3,                Shared
  VSIP_MEM_SHARED_RDONLY = 4,         Shared, Read Only
  VSIP_MEM_SHARED_CONST = 5           Shared, Constant
} vsip_memory_hint;
```

Return Value

Returns a pointer of type `vsip_dblock_p`, or *null* if the block create fails.

Restrictions

Errors

The arguments must conform to the following:

1. The data array size, *N*, must be greater than zero.
2. The memory *hint* must be a valid member of the `vsip_memory_hint` enumeration.

Notes/References

VSIPL data space allocated using the block create function is vendor dependent. The data are admitted and the pointer to the data space is hidden. The layout of the data in memory is vendor dependent. The data are accessed as if they were logically contiguous with an offset, stride(s), and length(s) into the data space. Offset, strides and lengths are in units of the data type.

Note to Implementors: memory allocation shall not be deferred.

Examples

To create a block large enough to hold an *M* by *N* matrix of type double:

```
vsip_block_d *Ablock = vsip_blockcreate_d(M*N, VSIP_MEM_NONE);
if (NULL == Ablock) error("Create of Ablock failed");
```

To create a block large enough to hold a complex vector of type complex double and length *N*:

```
vsip_cblock_d *Ablock = vsip_cblockcreate_d(N, VSIP_MEM_NONE);
if (NULL == Ablock) error("Create of Ablock failed");
```

See Also

`vsip_blockbind_p`, `vsip_blockadmit_p`, `vsip_blockrelease_p`, `vsip_blockfind_p`, and `vsip_blockdestroy_p`.

vsip_*d*blockdestroy_*p* Memory Block Destroy

Destroy (free) a VSIPL block object and any data array(s) allocated for it by VSIPL.

Functionality

Destroys (frees) a VSIPL block object, `vsip_block_p`, and any *VSIPL data* array.

Prototypes

```
void vsip_blockdestroy_f(
                    vsip_block_f *block);
void vsip_blockdestroy_i(
                    vsip_block_i *block);
void vsip_cblockdestroy_f(
                    vsip_cblock_f *block);
void vsip_cblockdestroy_i(
                    vsip_cblock_i *block);
void vsip_blockdestroy_bl(
                    vsip_block_bl *block);
void vsip_blockdestroy_vi(
                    vsip_block_vi *block);
void vsip_blockdestroy_mi(
                    vsip_block_mi *block);
void vsip_blockdestroy_ti(
                    vsip_block_ti *block);
```

Arguments

`*block` Pointer to a block object.

Return Value

None.

Restrictions

Errors

The arguments must conform to the following:

1. The block object must be valid. It is not a mistake to destroy a *null* pointer.
2. The block object must not be *derived* from a complex block object.

Notes/References

If necessary, the programmer can determine the pointer(s) to the user bound array(s) with a call to `vsip_dblockfind_f` before the (*released*) block is destroyed.

Destroying an admitted block bound to a user data array(s) may not update the data in the user array(s).

An argument of *null* is not an error.

Examples

Destroy a VSIPL block object.

```
{
   vsip_block_bl* xblock = vsip_blockcreate_bl(1000, VSIP_MEM_NONE);
   …
   vsip_blockdestroy_bl(xblock);
}
```

See Also

`vsip_`*d*`blockcreate_`*p*, `vsip_blockbind_`*p*, `vsip_cblockbind_`*p*

vsip_blockfind_*p* Memory Block Find

Find the pointer to the data bound to a VSIPL *released* block object.

Functionality

Returns the address of the user data array bound to a VSIPL released block. If the block is not released a *null* pointer is returned. Note that if the block was bound to `NULL` and is in the released state then a *null* pointer will be returned.

Prototypes

```
vsip_scalar_f *vsip_blockfind_f(
                    const vsip_block_f *block);
vsip_scalar_i *vsip_blockfind_i(
                    const vsip_block_i *block);
vsip_scalar_bl *vsip_blockfind_bl(
                    const vsip_block_bl *block);
vsip_scalar_vi *vsip_blockfind_vi(
                    const vsip_block_vi *block);
vsip_scalar_vi *vsip_blockfind_mi(
                    const vsip_block_mi *block);
vsip_scalar_vi *vsip_blockfind_ti(
                    const vsip_block_ti *block);
```

Arguments

`*block` Pointer to a block object.

Return Value

Returns a pointer of type `vsip_scalar_p` to the *released* block's data array, or *null* if the block object is not in the released state. Note that if the block is released and bound to a `NULL` then a *null* pointer is also returned.

Restrictions

Errors

The arguments must conform to the following:

1. The block object must be valid.

Notes/References

Although the data in a *derived* block is released when the parent block is released the derived block is never in a released state so blockfind will fail and return *null*. To find the data for a derived block the parent block must be queried.

Examples

See Also

`vsip_dblockcreate_p`, `vsip_dblockbind_p`, and `vsip_dblockdestroy_p`.

vsip_cblockfind_*p* Memory Complex Block Find

Returns the pointers to the *user data array*(s) bound to a VSIPL *released* complex block object, `vsip_cblock_p`, or *nulls* if the complex block object data are in the *admitted* state.

Functionality

Returns pointers to the *user data arrays* bound to a VSIPL *released* complex block object, `vsip_cblock_p`, or *null* if the block object data are in the *admitted* state.

Prototypes

```
void vsip_cblockfind_f(
                    const vsip_cblock_f *block,
                    vsip_scalar_f* *data1,
                    vsip_scalar_f* *data2);
void vsip_cblockfind_i(
                    const vsip_cblock_i *block,
                    vsip_scalar_i* *data1,
                    vsip_scalar_i* *data2);
```

Arguments

`*block`	Block object.
`**data1`	Output - Pointer to a pointer to the data array, or data array for the real values. Returns *null* if the complex block is in the *admitted* state.
`**data2`	Output - Pointer to a *null* pointer, or to the previous data array for the imaginary values. Returns *null* if the complex block is in the *admitted* state or if the previous binding was to data in *complex interleaved* form.

Return Value

None.

Restrictions

Errors

The arguments must conform to the following:

1. The block object must be valid.
2. The pointers to the *user data arrays* must be valid – non-*null*.

Notes/References

Examples

See Also

`vsip_dblockcreate_p`, `vsip_dblockbind_p`, and `vsip_dblockdestroy_p`

vsip_blockrebind_*p* Block Rebind

Rebind a VSIPL block to user-specified data.

Functionality

Rebinds an existing VSIPL *released* real block object, `vsip_block_p`, to a new (previously allocated) *user data array*. It must contain at least *N*, `vsip_scalar_p`, elements where *N* is the number of elements in the existing block object. An attempt to rebind either a derived block object, or a block object that is in an *admitted* state, will fail. In either case a *null* will be returned. Otherwise, it returns a pointer to the old *user data array*.

Prototypes

```
vsip_scalar_f *vsip_blockrebind_f (
                    vsip_block_f *block,
                    vsip_scalar_f* const data);
vsip_scalar_i *vsip_blockrebind_i(
                    vsip_block_i *block,
                    vsip_scalar_i* const data);
vsip_scalar_bl *vsip_blockrebind_bl(
                    vsip_block_bl *block,
                    vsip_scalar_bl* const data);
vsip_scalar_vi *vsip_blockrebind_vi(
                    vsip_block_vi *block,
                    vsip_scalar_vi* const data);
vsip_scalar_vi *vsip_blockrebind_mi(
                    vsip_block_mi *block,
                    vsip_scalar_vi* const data);
vsip_scalar_vi *vsip_blockrebind_ti(
                    vsip_block_ti *block,
                    vsip_scalar_vi* const data);
```

Arguments

`*block`	Pointer to a block object.
`*data`	Pointer to a *user data array* of contiguous memory containing at least *N* `vsip_scalar_p` elements to be bound to the block.

Return Value

Returns a pointer to the previous data array bound to the block, or *null* if the block is in an *admitted* state.

Restrictions

Rebind does not allow you to change the number of elements in a block.

Errors

The arguments must conform to the following:

1. The block object must be valid.
2. The *user data array* pointer must be valid – non-*null*.

Notes/References

Rebind does not allow you to change the number of elements in a block. However, there is no method to determine that the data pointer being bound is a valid pointer to an array of the proper size.

A derived block is not *releasable* and so may not be rebound. When the parent block is released and rebound to user data the corresponding data in the derived block is changed.

The block must be *admitted* to VSIPL before calling VSIPL functions that operate on the data.

The intent of rebind is to support efficient dynamic binding of buffers for I/O.

Examples

Ping-Pong I/O buffering.

```
#include <vsip.h>

#define BUFSIZE 1000

extern const volatile vsip_scalar_f *buf_ping, *buf_pong;
int pingpong = 0;
vsip_block_f* buf_blk = vsip_blockbind_f(buf_ping, BUFSIZE, VSIP_MEM_NONE);
vsip_vview_f* buf = vsip_vbind_f(buf_blk,0,1,N);

while{1){
  /* Wait until data ready in buf_ping (buf_pong) */
  wait_until_data_ready();
  pingpong = !pingpong;

  /* Release buf_pong  (buf_ping) */
  vsip_blockrelease_f(buf_blk,VSIP_FALSE);

  /* Start DMA of next data frame into buf_pong (buf_ping) */
  /* Rebind and admit buf_ping (buf_pong) */
  dma_write((vsip_blockrebind_f(buf_blk,
                              (pingpong ? buf_ping : buf_pong)));
  vsip_blockadmit_f(buf_blk, VSIP_TRUE);

  /* Do some processing using buf_ping */
  do_some_processing_with_buf();
}
```

See Also

`vsip_`*d*`blockcreate_`*p*, `vsip_`*d*`blockbind_`*p*, `vsip_`*d*`blockfind_`*p*, and `vsip_`*d*`blockdestroy_`*p*

vsip_cblockrebind_*p* Complex Block Rebind

Rebind a VSIPL complex block to user-specified data.

Functionality

Rebinds an existing VSIPL *released* complex block object, `vsip_cblock_p`, to either a single new (previously allocated) user-defined *user data array*, or to two new (previously allocated) user-defined data arrays. In the case of a single data array, the array must contain 2*N* `vsip_scalar_p` elements. For two data arrays, each array should contain *N* `vsip_scalar_p` elements. An attempt to rebind a block object that is in an *admitted* state will fail, and *null* will be returned. Otherwise, it returns a pointer to the old *user data array*.

Prototypes

```
void vsip_cblockrebind_f(
                    vsip_cblock_f *block,
                    vsip_scalar_f* const data1,
                    vsip_scalar_f* const data2,
                    vsip_scalar_f * *prevdata1,
                    vsip_scalar_f * *prevdata2);
void vsip_cblockrebind_i(
                    vsip_cblock_i *block,
                    vsip_scalar_i* const data1,
                    vsip_scalar_i* const data2,
                    vsip_scalar_i * *prevdata1,
                    vsip_scalar_i * *prevdata2);
```

Arguments

`*block` Pointer to complex block object

`*data1` If `data2` is *null*, then `data1` is a pointer to a *user data array* of contiguous memory containing at least 2*N* `vsip_scalar_p` elements. The even elements of the data array contain the real values, and the odd elements contain the imaginary values. The data are stored in *interleaved complex* form. Note that the first element is considered to be even because index values start at zero.

If `data2` is not *null*, then `data1` is a pointer to a *user data array* of contiguous memory containing at least *N* `vsip_scalar_p` elements, whose elements contain the real values. The data are stored in *split complex* form.

`*data2` If `data2` is *null*, then the data are stored in *interleaved complex* form.

If `data2` is not *null*, then it is a pointer to a *user data array* of contiguous memory containing at least *N* `vsip_scalar_p` elements, whose elements contain the imaginary values. The data are stored in *split complex* form.

`**prevdata1` Output - Pointer to a pointer to the previous *user data array*, or *user data* array for the real values. Returns *null* if the complex block is in the *admitted* state.

`**prevdata2` Output - Pointer to a *null* pointer, or to the previous *user data array* for the imaginary values. Returns *null* if the complex block is in the *admitted* state or if the previous binding was to data in *complex interleaved* form.

Return Value

None.

Restrictions

Complex rebind does not allow you to change the number of elements in a block.

Errors

The arguments must conform to the following:

1. The block object must be valid.
2. The pointers to the *user data arrays* must be valid – non-*null*.

Notes/References

Complex rebind does not allow you to change the number of elements in a block. However, there is no method to determine that the data pointer being bound is a valid pointer to an array of the proper size.

The block must be *admitted* to VSIPL before calling VSIPL functions that operate on the data.

The intent of rebind is to support efficient dynamic binding of buffers for I/O.

Examples

See Also

`vsip_`*d*`blockcreate_`*p*, `vsip_`*d*`blockbind_`*p*, `vsip_`*d*`blockfind_`*p*, and `vsip_`*d*`blockdestroy_`*p*

vsip_blockrelease_*p* Block Release

Release a VSIPL block for direct user access.

Functionality

Releases a VSIPL block object, `vsip_block_p`, to allow direct user access of the data array. Block objects created by `vsip_dblockcreate_p` and derived blocks cannot be released. An attempt to do so will return a *null*. A true *update* flag indicates that the data in the user-specified data array shall be updated to match the data associated with the block. If the *update* flag is false the state of the user data is implementation dependent, and the user data array should be assumed to contain undefined data.

Prototypes

```
vsip_scalar_f *vsip_blockrelease_f(
                     vsip_block_f *block,
                     vsip_scalar_bl update);

vsip_scalar_i *vsip_blockrelease_i(
                     vsip_block_i *block,
                     vsip_scalar_bl update);

vsip_scalar_bl *vsip_blockrelease_bl(
                     vsip_block_bl *block,
                     vsip_block_bl update);

vsip_scalar_vi *vsip_blockrelease_vi(
                     vsip_block_vi *block,
                     vsip_scalar_bl update);

vsip_scalar_vi *vsip_blockrelease_mi(
                     vsip_block_mi *block,
                     vsip_scalar_bl update);

vsip_scalar_vi *vsip_blockrelease_ti(
                     vsip_block_ti *block,
                     vsip_scalar_bl update);
```

Arguments

`*block`	Pointer to a block object.
`update`	Boolean flag where true indicates that the block object's data must be maintained during the state change.

Return Value

Returns *null* if the block release fails, otherwise it returns the pointer to the data array.

Restrictions

Errors

The arguments must conform to the following:

1. The block object must be valid.

Notes/References

It is **not** an error to *release* a block that is already in the released state.

Release causes any deferred execution associated with the block object, and any changes to the data array, to be completed before the function returns.

The intent of using a false *update* flag is that if the data in the block object is no longer needed, then there is no need to force consistency between the block object's data and the user-specified data array with a potential copy operation.

If the block is a *derived* block, derived from a complex block, only the complex block object can be released and admitted.

Examples

Add two vectors together.

```
#include <vsip.h>
…
int i;
vsip_scalar_d a[N], c[N];
vsip_block_d
*ablk = vsip_blockbind_d (a, N, VSIP_MEM_NONE),
*cblk = vsip_blockbind_d (c, N, VSIP_MEM_NONE);
vsip_vview_d
*va = vsip_vbind_d(ablk, 0, 1, N),
*vb = vsip_vcreate_d(N, VSIP_MEM_NONE),
*vc = vsip_vbind_d(cblk, 0, 1, N);
for (i=0; i<N; i++) a[i] = cosh(2*M_PI*i/N);
vsip_vramp_d(0.0, 1.0/N, vb);
vsip_blockadmit_d(ablk,VSIP_TRUE);
vsip_vadd_d(va,vb,vc);
vsip_blockrelease_d(cblk,VSIP_TRUE);
for (i=0; i<N; i++) printf("c[%i] = %d\n",i,c[i]);
```

See Also

`vsip_`*d*`blockcreate_`*p*, `vsip_blockadmit_`*p*, `vsip_blockrelease_`*p*, `vsip_blockfind_`*p*, and `vsip_blockdestroy_`*p*.

vsip_cblockrelease_*p* Complex Block Release

Release a complex block from VSIPL for direct user access.

Functionality

Releases a VSIPL complex block object, `vsip_cblock_p`, for direct user access to the data array(s). Block objects created by `vsip_dblockcreate_p` cannot be released. An attempt to do so will return *nulls* in both pointer values. A true *update* flag indicates that the data in the user-specified data array shall be updated to match the data associated with the block. If the *update* flag is false the state of the user data is implementation dependent, and the user data array should be assumed to contain undefined data.

Prototypes

```
void vsip_cblockrelease_f(
                    vsip_cblock_f *block,
                    vsip_scalar_bl update,
                    vsip_scalar_f* *data1,
                    vsip_scalar_f* *data2
                    );
void vsip_cblockrelease_i(
                    vsip_cblock_i *block,
                    vsip_scalar_bl update,
                    vsip_scalar_i* *data1,
                    vsip_scalar_i* *data2
                    );
```

Arguments

`*block`	Pointer to a block object.
`update`	Boolean flag where true indicates that the block object's data must be maintained during the state change.
`**data1`	Pointer to output data array - If the pointer returned in `data2` is *null*, then the pointer returned in `data1` is a pointer to a *user data array* of contiguous memory containing at least 2*N* `vsip_scalar_p` elements. The even elements of the data array contain the real part values and the odd elements contain the imaginary part values. The data are stored in *interleaved complex* form. Note that the first element is considered to be even because index values start at zero. If the pointer returned in `data2` is not *null*, then the pointer returned in `data1` is a pointer to a *user data array* of contiguous memory containing at least *N* `vsip_scalar_p` elements, whose elements contain the real part values. The data are stored in *split complex* form.
`**data2`	Pointer to output data array - If the pointer returned in `data2` is *null*, then the data are stored in *interleaved complex* form. If the pointer returned in `data2` is not *null*, then it is a pointer to a *user data array* of contiguous memory containing at least *N* `vsip_scalar_p` elements, whose elements contain the imaginary part values. The data are stored in *split complex* form.

Return Value

None

Restrictions

Errors

The arguments must conform to the following:

1. The block object must be valid.
2. The pointers to the *user data arrays* must be valid – non-*null*.

Notes/References

It is **not** an error to *release* a block that is already in the released state.

Release causes any deferred execution associated with the complex block object, and any changes to the data, to be completed before the function returns.

The intent of using a false *update* flag is that if the data in the block object is no longer needed, then there is no need to force consistency between the block object's data and the user-specified data array with a potential copy operation.

This function returns either a single pointer to the *user data array*, as the third argument (for *interleaved* complex data), or two pointers to the *user data arrays* as the third and fourth arguments (for *split* complex data). In the case of *interleaved* complex data, the fourth argument will be returned as *null*. If the block is not releasable, both pointers will be returned as *null*.

Examples

The first example below illustrates a split `cblockrelease`:

```
#include <stdio.h>
#include <stdlib.h>
#include <vsip.h>

#define  N   829

int main ( )
{
   int i;
   vsip_cscalar_f X, Y;
   vsip_scalar_f  dat1[N], dat2[N];  /* input data arrays */
   vsip_scalar_f  *addr1, *addr2;    /* returned data pointers */

   /* Input complex scalar */
   X.r = 2.0;
   X.i = 0.0;

   /* Input data sets */
   for (i = 0; i < N; i++)
   {
      dat1[i] = (vsip_scalar_f)( i);
      dat2[i] = (vsip_scalar_f)(-i);
   }
   /*  Initialize VSIPL  */
   vsip_init((void *)0);
   {
      /* Bind data to a complex block */
      vsip_cblock_f *cblock = vsip_cblockbind_f(
                dat1, dat2, N, VSIP_MEM_NONE);
```

```
      vsip_cvview_f *cdat = vsip_cvbind_f(cblock, 0, 1, N);

      /* Admit the block into VSIPL */
      vsip_cblockadmit_f(cblock, VSIP_TRUE);

      /* Multiply in-place the complex data type (dat1,dat2) by X */
      vsip_csvmul_f(X, cdat, cdat);

      /* Release the block back to the user */
      vsip_cblockrelease_f(cblock, VSIP_TRUE, &addr1, &addr2);

      /* Destroy the block and its views and data */
      vsip_cvdestroy_f(cdat);
      vsip_cblockdestroy_f(cblock);
   }
   /*  Finalize VSIPL  */
   vsip_finalize((void *)0);

   /* Print results using original pointers */
   for (i = 0; i < N; i++)
      printf("result  %d :   %.1f   %.1f\n", i, dat1[i], dat2[i]);
   printf("\n\n");

   /* Print results using returned pointers */
   for (i = 0; i < N; i++)
      printf( "result(again)  %d :   %.1f   %.1f\n",
                i, *(addr1 + i), *(addr2 + i) );
   printf("\n\n");
   return 0;
}
```

The second example below illustrates an interleaved `cblockrelease`:

```
#include <stdio.h>
#include <stdlib.h>
#include <vsip.h>

#define  N   829

int main ( )
{
   int i;
   vsip_cscalar_f X, Y;
   vsip_scalar_f  dat1[2*N];
   vsip_scalar_f  *addr1, *addr2;

   /* Initialize VSIPL */
   vsip_init( (void *)0 );

   /* Input complex scalar */
   X.r = 2.0;
   X.i = 0.0;

   /* Input data sets */
   for (i = 0; i < 2*N; i += 2){
      dat1[i] = (vsip_scalar_f)( i);
      dat1[i+1] = (vsip_scalar_f)(-i);
   }
```

```
   {
      /* Bind data to a complex block */
      vsip_cblock_f *cblock = vsip_cblockbind_f(
                dat1, NULL, N, VSIP_MEM_NONE);
      vsip_cvview_f *cdat = vsip_cvbind_f(cblock, 0, 1, N);

      /* Admit the block into VSIPL */
      vsip_cblockadmit_f(cblock, VSIP_TRUE);

      /* Multiply in-place the complex data by X */
      vsip_csvmul_f(X, cdat, cdat);

      /* Release the block back to the user */
      vsip_cblockrelease_f(cblock, VSIP_TRUE, &addr1, &addr2);

      /* Destroy the block and its views and data */
      vsip_cvdestroy_f(cdat);
      vsip_cblockdestroy_f(cblock);
   }

   /* Finalize VSIPL */
   vsip_finalize( (void *)0 );

   /*  Print results using the original pointer */
   for (i = 0; i < 2*N; i += 2)
      printf("result  %d :   %.1f   %.1f\n", i, dat1[i], dat1[i+1]);
   printf("\n\n");

   /* Note that a pointer to NULL is returned in addr2.
    * Only addr1 is useful. Then print results using
    * the returned pointer.*/
   for (i = 0; i < 2*N; i += 2)
      printf( "result(again)  %d :   %.1f   %.1f\n",
                i, *(addr1 + i), *(addr1 + i + 1) );
   printf("\n\n");
   return 0;
}
```

See Also

vsip_*d*blockcreate_*p*, vsip_cblockadmit_*p*, vsip_blockrelease_*p*, vsip_cblockfind_*p*, and vsip_cblockdestroy_*p*

vsip_complete Complete Deferred Execution

Force all deferred VSIPL execution to complete.

Functionality

Forces all deferred VSIPL execution (limited to this thread on this processor) to complete and then returns.

Prototypes

```
void vsip_complete(
                     void);
```

Arguments

None.

Return Value

None.

Restrictions

None.

Errors

None.

Notes/References

The primary purpose of `vsip_complete` is for debugging. Applications may be coordinating with other libraries that share implementation knowledge with VSIPL. User application code cannot directly observe the effects of deferred execution without using VSIPL private information.

Deferred execution is an implementation issue, and is an optional method to potentially improve performance.

Examples

See Also

vsip_cstorage_*p* Complex Storage

Returns the preferred complex storage format, *interleaved*, *split*, or *none* for a precision type for this implementation.

Functionality

Returns the preferred complex storage format. The preferred storage type can be dependent on the precision if required for best performance. For instance float precision may have a preferred user data type of split, and double precision may have a preferred user data type of interleaved.

Prototypes

```
vsip_cmplx_mem vsip_cstorage_p (
                     void);
```

Where:

```
typedef enum {
  VSIP_CMPLX_INTERLEAVED = 0,        Interleaved
  VSIP_CMPLX_SPLIT = 1,              Split, separate real and imaginary
  VSIP_CMPLX_NONE = 2,               No, preferred storage layout
  VSIP_CMPLX_DEPENDENT = 3           Not returned for this function
} vsip_cmplx_mem;
```

Arguments

None.

Return Value

Enumerated type corresponding to preferred storage format.

Restrictions

Errors

The arguments must conform to the following:

1. All view objects must be valid.

Notes/References

It is also possible to determine the preferred storage format at compile time. The include file vsip.h defines the value of `VSIP_CMPLX_MEM_P` to be one of:

```
{VSIP_CMPLX_INTERLEAVED_P |  VSIP_CMPLX_SPLIT_P | VSIP_CMPLX_NONE_P}.
e.g.
#define VSIP_CMPLX_MEM_F (VSIP_CMPLX_SPLIT_F)
#define VSIP_CMPLX_MEM_D (VSIP_CMPLX_INTERLEAVED_D)
```

We note that the *_P* character in the macro is replaced with the appropriate capital for the supported types.

We note this function replaces the `vsip_cstorage` function which has been deprecated and moved to the end of the document. The original is retained in the specification to support code portability.

This function allows for precision dependent prefered storage. For instance internal storage for float might be split and for integer might be interleaved or different precisions of float or integer may have different internal storage requirements.

Examples

See Also

The function `vsip_cstorage` is deprecated. See section *Notes to Implementors* for this function.

Vector View Object Functions

A VSIPL block holds the data in a *data array*. A block can be *viewed* as a vector or matrix. Two or more vector and/or matrix objects may reference the same block. There is no apparent difference to the application programmer for operation by VSIPL library functions on blocks bound only to *VSIPL data* arrays or (admitted) blocks associated with *user data arrays*.

A vector view object has the attributes of *offset*, *stride*, and *length* (number) of elements.

Vector views can be treated as row vectors or column vectors. When used in conjunction with matrix view objects they are normally treated as row vectors.

Vector view object functions are provided to:

- Create (constructors) vector view objects
- Destroy (destructors) vector view objects,
- Modify/manipulate vector view objects, and
- Access functions for vector view objects.

vsip_***d*valldestroy_*p***	Destroy Vector and Block
vsip_***d*vbind_*p***	Create and Bind a Vector View
vsip_***d*vcloneview_*p***	Create Vector View Clone
vsip_***d*vcreate_*p***	Create Vector
vsip_***d*vdestroy_*p***	Destroy Vector View
vsip_***d*vget_*p***	Vector Get Element
vsip_***d*vgetattrib_*p***	Vector Get View Attributes
vsip_***d*vgetblock_*p***	Vector Get Block
vsip_***d*vgetlength_*p***	Vector Get Length
vsip_***d*vgetoffset_*p***	Vector Get Offset
vsip_***d*vgetstride_*p***	Vector Get Stride
vsip_**vimagview_*p***	Create Imaginary Vector View
vsip_***d*vput_*p***	Vector Put Element
vsip_***d*vputattrib_*p***	Put Vector View Attributes
vsip_***d*vputlength_*p***	Vector Put Length
vsip_***d*vputoffset_*p***	Vector Put Offset
vsip_***d*vputstride_*p***	Vector Put Stride
vsip_**vrealview_*p***	Create Real Vector View
vsip_***d*vsubview_*p***	Create Subview Vector View

vsip_*d*valldestroy_*p* Destroy Vector and Block

Destroy (free) a vector, its associated block, and any *VSIPL* data array bound to the block.

Functionality

Destroys (frees) a vector view object, the block object to which it is bound, and any *VSIPL data* array. If **v** is a vector of type `vsip_dvview_p` then

```
vsip_dvalldestroy_p(v);
```

is equivalent to

```
vsip_dblockdestroy_p(vsip_dvdestroy_p(v);
```

This is the complementary function to `vsip_dvcreate_p` and should only be used to destroy vectors that have only one view bound to the block object.

Prototypes

```
void vsip_valldestroy_f(
                     vsip_vview_f *v);
void vsip_valldestroy_i(
                     vsip_vview_i *v);
void vsip_cvalldestroy_f(
                     vsip_cvview_f *v);
void vsip_cvalldestroy_i(
                     vsip_cvview_i *v);
void vsip_valldestroy_bl(
                     vsip_vview_bl *v);
void vsip_valldestroy_vi(
                     vsip_vview_vi *v);
void vsip_valldestroy_mi(
                     vsip_vview_mi *v);
void vsip_valldestroy_ti(
                     vsip_vview_ti *v);
```

Arguments

`*v` Vector view object.

Return Value

None.

Restrictions

Errors

The arguments must conform to the following:

1. The vector view object must be valid. An argument of *null* is not an error.
2. The specified vector view must be the only view bound to the block.
3. The vector view must not be bound to a *derived* block (derived from a complex block).

Notes/References

If the vector view is bound to a *derived* block (derived from a complex block) the complex block must be destroyed to free the block and associated data.

An argument of *null* is not an error.

Examples

Create and destroy a simple vector of 100 elements.

```
vsip_vview_f *v = vsip_vcreate_f((vsip_length)100, VSIP_MEM_NONE);
…
vsip_valldestroy_f(v);
```

See Also

`vsip_`*d*`blockcreate_`*p*, `vsip_blockbind_`*p*, `vsip_cblockbind_`*p*, `vsip_`*d*`blockadmit_`*p*, `vsip_blockrelease_`*p*, `vsip_cblockrelease_`*p*, `vsip_blockrebind_`*p*, `vsip_cblockrebind_`*p*, `vsip_blockfind_`*p*, `vsip_cblockfind_`*p*, `vsip_`*d*`blockdestroy_`*p*, `vsip_`*d*`vbind_`*p*, `vsip_`*d*`vcreate_`*p*, and `vsip_`*d*`vdestroy_`*p*

vsip_*d*vbind_*p* Create and Bind a Vector View

Create a vector view object and bind it to a block object.

Functionality

Creates a vector view object or returns *null* if it fails. If the view create is successful, it: (1) binds the vector view object to the block object, (2) sets the *offset* from the beginning of the data array to the beginning of the vector, the *stride* between scalar elements, and the *length* in elements (number of scalar elements), and (3) then returns a (pointer to the) vector view object.

Prototypes

```
vsip_vview_f *vsip_vbind_f(
                    const vsip_block_f *block,
                    vsip_offset offset,
                    vsip_stride stride,
                    vsip_length length);
vsip_vview_i *vsip_vbind_i(
                    const vsip_block_i *block,
                    vsip_offset offset,
                    vsip_stride stride,
                    vsip_length length);
vsip_cvview_f *vsip_cvbind_f(
                    const vsip_cblock_f *block,
                    vsip_offset offset,
                    vsip_stride stride,
                    vsip_length length);
vsip_cvview_i *vsip_cvbind_i(
                    const vsip_cblock_i *block,
                    vsip_offset offset,
                    vsip_stride stride,
                    vsip_length length);
vsip_vview_bl *vsip_vbind_bl(
                    const vsip_block_bl *block,
                    vsip_offset offset,
                    vsip_stride stride,
                    vsip_length length);
vsip_vview_vi *vsip_vbind_vi(
                    const vsip_block_vi *block,
                    vsip_offset offset,
                    vsip_stride stride,
                    vsip_length length);
vsip_vview_mi *vsip_vbind_mi(
                    const vsip_block_mi *block,
                    vsip_offset offset,
                    vsip_stride stride,
                    vsip_length length);
vsip_vview_ti *vsip_vbind_ti(
                    const vsip_block_ti *block,
                    vsip_offset offset,
                    vsip_stride stride,
                    vsip_length length);
```

Arguments

`*block`	Pointer to block object.
`offset`	Vector view *offset* in elements relative to the base of block object.
`stride`	Vector view *stride* between scalar elements.
`length`	Vector view *length* in elements.

Return Value

The function returns a pointer to the created vector view object, or *null* if the memory allocation for new object fails.

Restrictions

Errors

The arguments must conform to the following:

1. The block object must be valid.
2. The *offset* must be less than the length of the block's data array.
3. The *stride, length, and offset* arguments must not specify a vector view that exceeds the bounds of the data array of the associated block.

Notes/References

It is important for the application to check the function's return value for a memory allocation failure.

Note to Implementors: In *development mode*, this function updates the number of bindings (reference count) recorded in the block object.

Examples

See Also

`vsip_`*d*`blockcreate_`*p*, `vsip_blockbind_`*p*, `vsip_cblockbind_`*p*, `vsip_`*d*`blockadmit_`*p*, `vsip_blockrelease_`*p*, `vsip_cblockrelease_`*p*, `vsip_blockrebind_`*p*, `vsip_cblockrebind_`*p*, `vsip_blockfind_`*p*, `vsip_cblockfind_`*p*, `vsip_`*d*`blockdestroy_`*p*, `vsip_`*d*`vcreate_`*p*, `vsip_`*d*`vdestroy_`*p*, and `vsip_`*d*`valldestroy_`*p*

vsip_*d*vcloneview_*p* Create Vector View Clone

Create a clone of a vector view.

Functionality

Creates a new vector view object, copies all of the attributes of the source vector view object to the new vector view object, and then binds the new vector view object to the block object of the source vector view object. This function returns *null* on a memory allocation (creation) failure; otherwise, it returns a pointer to the new vector view object.

Prototypes

```
vsip_vview_f *vsip_vcloneview_f(
                    const vsip_vview_f *v);

vsip_vview_i *vsip_vcloneview_i(
                    const vsip_vview_i *v);

vsip_cvview_f *vsip_cvcloneview_f(
                    const vsip_cvview_f *v);

vsip_cvview_i *vsip_cvcloneview_i(
                    const vsip_cvview_i *v);

vsip_vview_bl *vsip_vcloneview_bl(
                    const vsip_vview_bl *v);

vsip_vview_vi *vsip_vcloneview_vi(
                    const vsip_vview_vi *v);

vsip_vview_mi *vsip_vcloneview_mi(
                    const vsip_vview_mi *v);

vsip_vview_ti *vsip_vcloneview_ti(
                    const vsip_vview_ti *v);
```

Arguments

`*v` Source vector view object.

Return Value

Returns a pointer to the created vector view object clone, or *null* if the memory allocation for new object fails.

Restrictions

Errors

The arguments must conform to the following:

1. The vector view object must be valid.

Notes/References

It is important for the application to check the return value for *null* in case of a memory allocation failure.

Note to Implementors: In *development mode,* `vsip_dvcloneview_p` increments the number of bindings (reference count) recorded in the block object.

Examples

See Also

`vsip_`*d*`vbindview_`*p*, `vsip_`*d*`vcloneview_`*p*, `vsip_`*d*`vcreateview_`*p*, `vsip_`*d*`vsubview_`*p*
`vsip_`*d*`vimagview_`*p*, `vsip_`*d*`vrealview_`*p*

vsip_*d*vcreate_*p* Create Vector

Creates a block object and a vector view (object) of the block.

Functionality

Creates a block object with an *N* element *VSIPL data* array, it creates a unit stride vector view object and then binds the block object to it.

The function

```
vsip_vview_p *vsip_vcreate_p(N, hint);
```

returns the same result as

```
vsip_dvbind_p(vsip_dblockcreate_p(N, hint),
              (vsip_offset)0, (vsip_stride)1, N);
```

except that `vsip_vcreate_p` returns *null* if `vsip_dblockcreate_p(N, hint)` returns *null*.

Prototypes

```
vsip_vview_f *vsip_vcreate_f(
                     vsip_length N,
                     vsip_memory_hint hint);

vsip_vview_i *vsip_vcreate_i(
                     vsip_length N,
                     vsip_memory_hint hint);

vsip_cvview_f *vsip_cvcreate_f(
                     vsip_length N,
                     vsip_memory_hint hint);

vsip_cvview_i *vsip_cvcreate_i(
                     vsip_length N,
                     vsip_memory_hint hint);

vsip_vview_bl *vsip_vcreate_bl(
                     vsip_length length,
                     vsip_memory_hint hint);

vsip_vview_vi *vsip_vcreate_vi(
                     vsip_length N,
                     vsip_memory_hint hint);

vsip_vview_mi *vsip_vcreate_mi(
                     vsip_length N,
                     vsip_memory_hint hint);

vsip_vview_ti *vsip_vcreate_ti(
                     vsip_length N,
                     vsip_memory_hint hint);
```

Arguments

`N`	Number of elements of vector.
`hint`	Memory hint

```
typedef enum {
  VSIP_MEM_NONE = 0,                  No hint
  VSIP_MEM_RDONLY = 1,                Read Only
  VSIP_MEM_CONST = 2,                 Constant
  VSIP_MEM_SHARED = 3,                Shared
  VSIP_MEM_SHARED_RDONLY = 4,         Shared, Read Only
  VSIP_MEM_SHARED_CONST = 5           Shared, Constant
} vsip_memory_hint;
```

Return Value

Returns a pointer to the created vector view object, or *null* if the memory allocation for new object fails.

Restrictions

Errors

The arguments must conform to the following:

1. The vector length, *N*, must be greater than zero.
2. The memory *hint* must be a valid member of the `vsip_memory_hint` enumeration.

Notes/References

Note to Implementors: In *development mode,* `vsip_`*d*`vcreate_`*p* sets the initial number of bindings in the block object on which the returned vector is bound.

Examples

Make a block of type double, and length 32, and attach to it a vector view of type double, unit stride, and of length 32.

```
vsip_vview_d *vector = vsip_vcreate_d((vsip_length) 32, VSIP_MEM_NONE);
```

See Also

`vsip_`*d*`blockcreate_`*p*, `vsip_blockbind_`*p*, `vsip_cblockbind_`*p*, `vsip_`*d*`blockadmit_`*p*, `vsip_blockrelease_`*p*, `vsip_cblockrelease_`*p*, `vsip_blockrebind_`*p*, `vsip_cblockrebind_`*p*, `vsip_blockfind_`*p*, `vsip_cblockfind_`*p*, `vsip_`*d*`blockdestroy_`*p*, `vsip_`*d*`vbind_`*p*, `vsip_`*d*`vdestroy_`*p*, and `vsip_`*d*`valldestroy_`*p*

vsip_*d*vdestroy_*p* Destroy Vector View

Destroy (free) a vector view object and return a pointer to the associated block object.

Functionality

Frees a vector view object from the block object that it was bound to, destroys the vector view object, and then returns a pointer to the block object. If the vector view argument is *null*, it returns *null*.

Prototypes

```
vsip_block_f *vsip_vdestroy_f(
                     vsip_vview_f *v);

vsip_block_i *vsip_vdestroy_i(
                     vsip_vview_i *v);

vsip_cblock_f *vsip_cvdestroy_f(
                     vsip_cvview_f *v);

vsip_cblock_i *vsip_cvdestroy_i(
                     vsip_cvview_i *v);

vsip_block_bl *vsip_vdestroy_bl(
                     vsip_vview_bl *v);

vsip_block_vi *vsip_vdestroy_vi(
                     vsip_vview_vi *v);

vsip_block_mi *vsip_vdestroy_mi(
                     vsip_vview_mi *v);

vsip_block_ti *vsip_vdestroy_ti(
                     vsip_vview_ti *v);
```

Arguments

`*v` Vector view object.

Return Value

Returns a pointer to the block object to which the vector view was bound, or *null* if the calling argument was *null*.

Restrictions

Errors

The arguments must conform to the following:

1. The vector view object must be valid. It is not an error to destroy a *null* pointer.

Notes/References

An argument of *null* is not an error.

Note to Implementors: In *development mode,* the function updates the bindings (reference count) recorded in the block object.

Examples

See Also

`vsip_`*d*`blockcreate_`*p*, `vsip_blockbind_`*p*, `vsip_cblockbind_`*p*, `vsip_`*d*`blockadmit_`*p*, `vsip_blockrelease_`*p*, `vsip_cblockrelease_`*p*, `vsip_blockrebind_`*p*, `vsip_cblockrebind_`*p*, `vsip_blockfind_`*p*, `vsip_cblockfind_`*p*, `vsip_`*d*`blockdestroy_`*p*, `vsip_`*d*`vbind_`*p*, `vsip_`*d*`vcreate_`*p*, and `vsip_`*d*`valldestroy_`*p*

vsip_*d*vget_*p* Vector Get Element

Get the value of a specified element of a vector view object.

Functionality

Returns the value of the specified element of a vector view object, x_j.

Prototypes

```
vsip_scalar_f vsip_vget_f(
                    const vsip_vview_f *x,
                    vsip_index j);
vsip_scalar_i vsip_vget_i(
                    const vsip_vview_i *x,
                    vsip_index j);
vsip_cscalar_f vsip_cvget_f(
                    const vsip_cvview_f *x,
                    vsip_index j);
vsip_cscalar_i vsip_cvget_i(
                    const vsip_cvview_i *x,
                    vsip_index j);
vsip_scalar_bl vsip_vget_bl(
                    const vsip_vview_bl *x,
                    vsip_index j);
vsip_scalar_vi vsip_vget_vi(
                    const vsip_vview_vi *x,
                    vsip_index j);
vsip_scalar_mi vsip_vget_mi(
                    const vsip_vview_mi *x,
                    vsip_index j);
vsip_scalar_ti vsip_vget_ti(
                    const vsip_vview_ti *x,
                    vsip_index j);
```

Arguments

`*x`	Vector view object
`j`	Index of vector element.

Return Value

Returns the value of the specified element of a vector view object.

Restrictions

Errors

The arguments must conform to the following:

1. The vector view object must be valid.
2. The index must be a valid index of the vector view.

Notes/References

Examples

See Also

`vsip_dsput_p`

vsip_*d*vgetattrib_*p* Vector Get View Attributes

Get the attributes of a vector view object.

Functionality

Returns the attributes of a vector view object: (pointer to) bound block object, *offset*, *stride*, and *length*.

Prototypes

```
void vsip_vgetattrib_f(
                      const vsip_vview_f *v,
                      vsip_vattr_f *attrib);

void vsip_vgetattrib_i(
                      const vsip_vview_i *v,
                      vsip_vattr_i *attrib);

void vsip_cvgetattrib_f(
                      const vsip_cvview_f *v,
                      vsip_cvattr_f *attrib);

void vsip_cvgetattrib_i(
                      const vsip_cvview_i *v,
                      vsip_cvattr_i *attrib);

void vsip_vgetattrib_bl(
                      const vsip_vview_bl *v,
                      vsip_vattr_bl *attrib);

void vsip_vgetattrib_vi(
                      const vsip_vview_vi *v,
                      vsip_vattr_vi *attrib);

void vsip_vgetattrib_mi(
                      const vsip_vview_mi *v,
                      vsip_vattr_mi *attrib);

void vsip_vgetattrib_ti(
                      const vsip_vview_ti *v,
                      vsip_vattr_ti *attrib);

typedef struct {
   vsip_offset offset;
   vsip_stride stride;
   vsip_length length;
   vsip_block_p *block;      Get only, ignored on Put
} vsip_vattr_p;

typedef struct {
   vsip_offset offset;
   vsip_stride stride;
   vsip_length length;
   vsip_cblock_p *block;     Get only, ignored on Put
} vsip_cvattr_p;
```

Arguments

`*v`	Vector view object.
`*attrib`	Pointer to output vector attribute structure.

Return Value

None.

Restrictions

Errors

The arguments must conform to the following:

1. The vector view object must be valid.
2. The pointer to the vector attribute structure must be valid – non-*null*.

Notes/References

The block attribute can be read (get), but cannot be set.

Examples

See Also

vsip_*d*vgetblock_*p* Vector Get Block

Get the *block* attribute of a vector view object.

Functionality

Returns a pointer to the *VSIPL block* object to which the vector view object is bound.

Prototypes

```
vsip_block_f *vsip_vgetblock_f(
                      const vsip_vview_f *v);

vsip_block_i *vsip_vgetblock_i(
                      const vsip_vview_i *v);

vsip_cblock_f *vsip_cvgetblock_f(
                      const vsip_cvview_f *v);

vsip_cblock_i *vsip_cvgetblock_i(
                      const vsip_cvview_i *v);

vsip_block_bl *vsip_vgetblock_bl(
                      const vsip_vview_bl *v);

vsip_block_vi *vsip_vgetblock_vi(
                      const vsip_vview_vi *v);

vsip_block_mi *vsip_vgetblock_mi(
                      const vsip_vview_mi *v);

vsip_block_ti *vsip_vgetblock_ti(
                      const vsip_vview_ti *v);
```

Arguments

`*v` Vector view object.

Return Value

Returns a pointer to the block object to which the vector view object is bound.

Restrictions

Errors

The arguments must conform to the following:

1. The vector view object must be valid.

Notes/References

The functions `vsip_dvgetattrib_p` and `vsip_dvputattrib_p` are not symmetric since you can *get* the block object but you cannot *put* the block object.

Examples

See Also

vsip_*d*vgetlength_*p* Vector Get Length

Get the *length* attribute of a vector view object

Functionality

Returns the value of the *length* (number of elements) attribute of a vector view object.

Prototypes

```
vsip_length vsip_vgetlength_f(
                      const vsip_vview_f *v);

vsip_length vsip_vgetlength_i(
                      const vsip_vview_i *v);

vsip_length vsip_cvgetlength_f(
                      const vsip_cvview_f *v);

vsip_length vsip_cvgetlength_i(
                      const vsip_cvview_i *v);

vsip_length vsip_vgetlength_bl(
                      const vsip_vview_bl *v);

vsip_length vsip_vgetlength_vi(
                      const vsip_vview_vi *v);

vsip_length vsip_vgetlength_mi(
                      const vsip_vview_mi *v);

vsip_length vsip_vgetlength_ti(
                      const vsip_vview_ti *v);
```

Arguments

`*v` Vector view object.

Return Value

Returns the value of the *length* attribute of a vector view object.

Restrictions

Errors

The arguments must conform to the following:

1. The vector view object must be valid.

Notes/References

Examples

See Also

vsip_*d*vgetoffset_*p* Vector Get Offset

Get the *offset* attribute of a vector view object.

Functionality

Returns the *offset* (in elements) to the first scalar element of a vector view from the start of the block object to which it is bound.

Prototypes

```
vsip_offset vsip_vgetoffset_f(
                    const vsip_vview_f *v);

vsip_offset vsip_vgetoffset_i(
                    const vsip_vview_i *v);

vsip_offset vsip_cvgetoffset_f(
                    const vsip_cvview_f *v);

vsip_offset vsip_cvgetoffset_i(
                    const vsip_cvview_i *v);

vsip_offset vsip_vgetoffset_bl(
                    const vsip_vview_bl *v);

vsip_offset vsip_vgetoffset_vi(
                    const vsip_vview_vi *v);

vsip_offset vsip_vgetoffset_mi(
                    const vsip_vview_mi *v);

vsip_offset vsip_vgetoffset_ti(
                    const vsip_vview_ti *v);
```

Arguments

`*v` Vector view object.

Return Value

Returns the value of the *offset* attribute of the vector view object.

Restrictions

Errors

The arguments must conform to the following:

1. The vector view object must be valid.

Notes/References

Examples

See Also

vsip_*d*vgetstride_*p* Vector Get Stride

Get the *stride* attribute of a vector view object.

Functionality

Returns the stride (in elements of the bound block) between successive scalar elements in a vector view.

Prototypes

```
vsip_stride vsip_vgetstride_f(
                    const vsip_vview_f *v);

vsip_stride vsip_vgetstride_i(
                    const vsip_vview_i *v);

vsip_stride vsip_cvgetstride_f(
                    const vsip_cvview_f *v);

vsip_stride vsip_cvgetstride_i(
                    const vsip_cvview_i *v);

vsip_stride vsip_vgetstride_bl(
                    const vsip_vview_bl *v);

vsip_stride vsip_vgetstride_vi(
                    const vsip_vview_vi *v);

vsip_stride vsip_vgetstride_mi(
                    const vsip_vview_mi *v);

vsip_stride vsip_vgetstride_ti(
                    const vsip_vview_ti *v);
```

Arguments

`*v` Vector view object.

Return Value

Returns the value of the *stride* attribute of the vector view object.

Restrictions

Errors

The arguments must conform to the following:

1. The vector view object must be valid.

Notes/References

Examples

See Also

vsip_vimagview_*p* Create Imaginary Vector View

Create a vector view object of the imaginary part of a complex vector from a complex vector view object.

Functionality

Creates a real vector view object from the "imaginary part of a complex" vector view object, or returns *null* if it fails.

On success, the function creates a *derived* block object (derived from the complex block object). The derived block object is bound to the imaginary data part of the original complex block and then binds a real vector view object to the block. The new vector encompasses the imaginary part of the input complex vector.

Prototypes

```
vsip_vview_f *vsip_vimagview_f(
                    const vsip_cvview_f *v);
vsip_vview_i *vsip_vimagview_i(
                    const vsip_cvview_i *v);
```

Arguments

`*v`	Source vector view object.

Return Value

Returns a pointer to the created "imaginary" vector view object, or *null* if the memory allocation for new object fails.

Restrictions

The *derived* block object cannot be destroyed or released. The parent complex block object may be released (if it is bound to *user data*). Destroying the complex block is the only way to free the memory associated with the *derived* block object.

Errors

The arguments must conform to the following:

1. The complex vector view object must be valid.

Notes/References

It is important for the application to check the return value for a memory allocation failure.

This function should not be confused with the function `vsip_simag_p()` which is a copy operator (copies the imaginary data).

There are no requirements on *offset* or *stride* of a real view on its *derived* block. Using `vsip_vgetattrib_p` information about the layout of the view on the block may be obtained.

CAUTION. Using attribute information, and the block bound to the vector, to bind new vectors outside the data space of the original vector produced by `vsip_simagview_p` will produce non-portable code. Portable code may be produced by: (1) remaining inside the data space of the vector, (2) by not assuming a set relationship of strides and offsets, and (3) by using the get

attributes functions to obtain necessary information within the application code to understand the layout for each implementation.

Note to Implementors:

- The resulting *derived* block must have a property which prevents it from being released or destroyed.
- In *development mode,* `vsip_dvimagview_p` updates the binding count in the parent complex block object.

Examples

Calculate a complex vector of length *N*, whose imaginary part is the cosine from zero to 2π and whose real part is the sine from zero to 2π, using real view and imaginary view functions to find the cosine and sine vectors.

```
#include <vsip.h>

vsip_cvview_d *CV = vsip_cvcreate_d((vsip_length)N, VSIP_MEM_NONE);
vsip_vview_d  *V = vsip_vcreate_d(vsip_length) N, VSIP_MEM_NONE),
              *RV, *IV;

/* make a ramp from zero to two pi where pi is M_PI*/
vsip_vramp_d(0, (2.0 * M_PI)/(double)(N - 1),V);

/* Fill the complex vector */
vsip_veuler_d(V,CV);

/* get the view of the real (cos) and imaginary (sin) parts */
vsip_vrealview_d(CV,RV);
vsip_vimagview_d(CV,IV);
```

See Also

`vsip_dvcloneview_p`, `vsip_dvcreateview_p`, `vsip_dvrealview_p`, `vsip_dvsubview_p`

vsip_*d*vput_*p* Vector Put Element

Put (Set) the value of a specified element of a vector view object

Functionality

Puts (sets) the value of the specified element of a vector view object.

$y_j \leftarrow x$

Prototypes

```
void vsip_vput_f(
                    const vsip_vview_f *y,
                    vsip_index j,
                    vsip_scalar_f x);
void vsip_vput_i(
                    const vsip_vview_i *y,
                    vsip_index j,
                    vsip_scalar_i x);
void vsip_cvput_f(
                    const vsip_cvview_f *y,
                    vsip_index j,
                    vsip_cscalar_f x);
void vsip_cvput_i(
                    const vsip_cvview_i *y,
                    vsip_index j,
                    vsip_cscalar_i x);
void vsip_vput_bl(
                    const vsip_vview_bl *y,
                    vsip_index j,
                    vsip_scalar_bl x);
void vsip_vput_vi(
                    const vsip_vview_vi *y,
                    vsip_index j,
                    vsip_scalar_vi x);
void vsip_vput_mi(
                    const vsip_vview_mi *y,
                    vsip_index j,
                    vsip_scalar_mi x);
void vsip_vput_ti(
                    const vsip_vview_ti *y,
                    vsip_index j,
                    vsip_scalar_ti x);
```

Arguments

`*y`	Vector view object of destination
`j`	Vector index *j* of vector element.
`x`	Scalar value to put

Return Value

None.

Restrictions

Errors

The arguments must conform to the following:

1. The vector view object must be valid.
2. The index must be a valid index of the vector view.

Notes/References

Examples

See Also

`vsip_`***ds***`get_`***p***

vsip_*d*vputattrib_*p* Put Vector View Attributes

Put (Set) the attributes of a vector view object.

Functionality

Sets the vector view attributes of *offset*, *stride*, and *length*, and as a programmer convenience, returns a pointer to the vector view object.

Prototypes

```
vsip_vview_f *vsip_vputattrib_f(
                    vsip_vview_f *v,
                    const vsip_vattr_f *attrib);

vsip_vview_i *vsip_vputattrib_i(
                    vsip_vview_i *v,
                    const vsip_vattr_i *attrib);

vsip_cvview_f *vsip_cvputattrib_f(
                    vsip_cvview_f *v,
                    const vsip_cvattr_f *attrib);

vsip_cvview_i *vsip_cvputattrib_i(
                    vsip_cvview_i *v,
                    const vsip_cvattr_i *attrib);

vsip_vview_bl *vsip_vputattrib_bl(
                    vsip_vview_bl *v,
                    const vsip_vattr_bl *attrib);

vsip_vview_vi *vsip_vputattrib_vi(
                    vsip_vview_vi *v,
                    const vsip_vattr_vi *attrib);

vsip_vview_mi *vsip_vputattrib_mi(
                    vsip_vview_mi *v,
                    const vsip_vattr_mi *attrib);

vsip_vview_ti *vsip_vputattrib_ti(
                    vsip_vview_ti *v,
                    const vsip_vattr_ti *attrib);

typedef struct {
   vsip_offset offset;
   vsip_stride stride;
   vsip_length length;
   vsip_block_p *block;      Get only, ignored on Put
} vsip_vattr_p;

typedef struct {
   vsip_offset offset;
   vsip_stride stride;
   vsip_length length;
   vsip_cblock_p *block;     Get only, ignored on Put
} vsip_cvattr_p;
```

Arguments

`*v`	Vector view object.
`*attrib`	Pointer to a vector attribute structure.

Return Value

Returns a pointer to the source vector view object as a programming convenience.

Restrictions

Errors

The arguments must conform to the following:

1. The vector view object must be valid.
2. The pointer to the vector attribute structure must be valid – non-*null*.
3. The *stride, length, and offset* arguments must not specify a vector view that exceeds the bounds of the data array of the associated block.

Notes/References

The functions `vsip_`*d*`vgetattrib_`*p* and `vsip_`*d*`vputattrib_`*p* are not symmetric since you can *get* the block object but you cannot *put* the block object.

Examples

```
#include <vsip.h>
…
vsip_vview_f *vect;
vsip_scalar_f sum;
…
{ /* Modify vector view to view "odd" elements and sum the odd values */
  vsip_vattr_f attrib;
  vsip_vgetattrib_f(vect, &attrib);
  attrib.stride *= 2;
  attrib.offset += 1;
  attrib.length /=2;
  sum = vsip_vsumval_f(vsip_vputattrib_f(vect, &attrib));
}
```

See Also

vsip_*d*vputlength_*p* Vector Put Length

Put (Set) the *length* attribute of a vector view object.

Functionality

Puts (sets) the *length* (number of elements) of a vector view.

Prototypes

```
vsip_vview_f *vsip_vputlength_f(
                    vsip_vview_f *v,
                    vsip_length length);
vsip_vview_i *vsip_vputlength_i(
                    vsip_vview_i *v,
                    vsip_length length);
vsip_vview_f *vsip_cvputlength_f(
                    vsip_cvview_f *v,
                    vsip_length length);
vsip_vview_i *vsip_cvputlength_i(
                    vsip_cvview_i *v,
                    vsip_length length);
vsip_vview_bl *vsip_vputlength_bl(
                    vsip_vview_bl *v,
                    vsip_length length);
vsip_vview_vi *vsip_vputlength_vi(
                    vsip_vview_vi *v,
                    vsip_length length);
vsip_vview_mi *vsip_vputlength_mi(
                    vsip_vview_mi *v,
                    vsip_length length);
vsip_vview_ti *vsip_vputlength_ti(
                    vsip_vview_ti *v,
                    vsip_length length);
```

Arguments

`*v`	Vector view object.
`length`	*Length* in elements.

Return Value

Returns a pointer to the source vector view object as a programming convenience.

Restrictions

Errors

The arguments must conform to the following:

1. The vector view object must be valid.
2. The *length* must be greater than zero.
3. The *length* argument must not specify a vector view that exceeds the bounds of the data array of the associated block.

Notes/References

Examples

See Also

vsip_*d*vputoffset_*p* Vector Put Offset

Put (Set) the *offset* attribute of a vector view object.

Functionality

Puts (sets) the *offset* (in elements) to the first scalar element of a vector view, from the start of the block object's data array, to which it is bound.

Prototypes

```
vsip_vview_f *vsip_vputoffset_f(
                    vsip_vview_f *v,
                    vsip_offset offset);
vsip_vview_i *vsip_vputoffset_i(
                    vsip_vview_i *v,
                    vsip_offset offset);
vsip_vview_f *vsip_cvputoffset_f(
                    vsip_cvview_f *v,
                    vsip_offset offset);
vsip_vview_i *vsip_cvputoffset_i(
                    vsip_cvview_i *v,
                    vsip_offset offset);
vsip_vview_bl *vsip_vputoffset_bl(
                    vsip_vview_bl *v,
                    vsip_offset offset);
vsip_vview_vi *vsip_vputoffset_vi(
                    vsip_vview_vi *v,
                    vsip_offset offset);
vsip_vview_mi *vsip_vputoffset_mi(
                    vsip_vview_mi *v,
                    vsip_offset offset);
vsip_vview_ti *vsip_vputoffset_ti(
                    vsip_vview_ti *v,
                    vsip_offset offset);
```

Arguments

`*v`	Vector view object.
`offset`	Offset in elements relative to the start of the block object.

Return Value

Returns a pointer to the source vector view object as a programming convenience.

Restrictions

Errors

The arguments must conform to the following:

1. The vector view object must be valid.
2. The *offset* argument must not specify a vector view that exceeds the bounds of the data array of the associated block.

Notes/References

Examples

See Also

vsip_*d*vputstride_*p* Vector Put Stride

Put (Set) the *stride* attribute of a vector view object.

Functionality

Puts (sets) the *stride* attribute of a vector view object. Stride is the distance in elements of the block between successive elements of the vector view.

Prototypes

```
vsip_vview_f *vsip_vputstride_f(
                    vsip_vview_f *v,
                    vsip_stride stride);
vsip_vview_i *vsip_vputstride_i(
                    vsip_vview_i *v,
                    vsip_stride stride);
vsip_cvview_f *vsip_cvputstride_f(
                    vsip_cvview_f *v,
                    vsip_stride stride);
vsip_cvview_i *vsip_cvputstride_i(
                    vsip_cvview_i *v,
                    vsip_stride stride);
vsip_vview_bl *vsip_vputstride_bl(
                    vsip_vview_bl *v,
                    vsip_stride stride);
vsip_vview_vi *vsip_vputstride_vi(
                    vsip_vview_vi *v,
                    vsip_stride stride);
vsip_vview_mi *vsip_vputstride_mi(
                    vsip_vview_mi *v,
                    vsip_stride stride);
vsip_vview_ti *vsip_vputstride_ti(
                    vsip_vview_ti *v,
                    vsip_stride stride);
```

Arguments

`*v`	Vector view object.
`stride`	Stride in elements.

Return Value

Returns a pointer to the source vector view object as a programming convenience.

Restrictions

Errors

The arguments must conform to the following:

1. The vector view object must be valid.
2. The *stride* argument must not specify a vector view that exceeds the bounds of the data array of the associated block.

Notes/References

Examples

See Also

vsip_vrealview_*p* Create Real Vector View

Create a vector view object of the real part of a complex vector from a complex vector view object.

Functionality

Creates a real vector view object from the "real part of a complex" vector view object, or returns *null* if it fails.

On success, the function creates a *derived* block object (derived from the complex block object). The derived block object is bound to the real data part of the original complex block and then binds a real vector view object to the block. The new vector encompasses the real part of the input complex vector.

Prototypes

```
vsip_vview_f *vsip_vrealview_f(
                    const vsip_cvview_f *v);
vsip_vview_i *vsip_vrealview_i(
                    const vsip_cvview_i *v);
```

Arguments

`*v` Source vector view object.

Return Value

Returns a pointer to the created "real" vector view object, or *null* if the memory allocation for new object fails.

Restrictions

The *derived* block object cannot be destroyed or released. The parent complex block object may be released (if it is bound to *user data*). Destroying the complex block is the only way to free the memory associated with the *derived* block object.

Errors

The arguments must conform to the following:

1. The complex vector view object must be valid.

Notes/References

It is important for the application to check the return value for a memory allocation failure.

This function should not be confused with the function `vsip_sreal_p()`, which is a copy operator (copies the real data).

There are no requirements on *offset* or *stride* of a real view on its *derived* block. By using `vsip_vgetattrib_p`, information about the layout of the view on the block may be obtained.

CAUTION. Using attribute information, and the block bound to the vector, to bind new vectors outside the data space of the original vector produced by `vsip_srealview_p` will produce non-portable code. Portable code may be produced by: (1) remaining inside the data space of the vector, (2) by not assuming a set relationship of strides and offsets, and (3) by using the get

attributes functions to obtain necessary information within the application code to understand the layout for each implementation.

Note to Implementors:

- The resulting *derived* block must have a property which prevents it from being released or destroyed.
- In *development mode,* `vsip_dvrealview_p` updates the binding count in the parent complex block object.

Examples

See example with `vsip_dvimagview_p`.

See Also

`vsip_dvcloneview_p`, `vsip_dvcreateview_p`, `vsip_dvimagview_p`, `vsip_dvsubview_p`

vsip_*d*vsubview_*p* Create Subview Vector View

Create a vector view object that is a subview of a vector view object (*offset*, and *length* are relative to the source view object, not the bound block object).

Functionality

Creates a subview vector view object from a source vector view object, and binds it to the same block object, or returns *null* if it fails. The zero index element of the new subview corresponds to the *index* element of the source vector view.

(The subview is relative to the source view, and *stride* is inherited from the source view).

Prototypes

```
vsip_vview_f *vsip_vsubview_f(
                      const vsip_vview_f *v,
                      vsip_index index,
                      vsip_length length);
vsip_vview_i *vsip_vsubview_i(
                      const vsip_vview_i *v,
                      vsip_index index,
                      vsip_length length);
vsip_cvview_f *vsip_cvsubview_f(
                      const vsip_cvview_f *v,
                      vsip_index index,
                      vsip_length length);
vsip_cvview_i *vsip_cvsubview_i(
                      const vsip_cvview_i *v,
                      vsip_index index,
                      vsip_length length);
vsip_vview_bl *vsip_vsubview_bl(
                      const vsip_vview_bl *v,
                      vsip_index index,
                      vsip_length length);
vsip_vview_vi *vsip_vsubview_vi(
                      const vsip_vview_vi *v,
                      vsip_index index,
                      vsip_length length);
vsip_vview_mi *vsip_vsubview_mi(
                      const vsip_vview_mi *v,
                      vsip_index index,
                      vsip_length length);
vsip_vview_ti *vsip_vsubview_ti(
                      const vsip_vview_ti *v,
                      vsip_index index,
                      vsip_length length);
```

Arguments

`*v`	Source vector view.
`index`	The subview vectors first element (index 0) is at vector index "*index*" of the source vector.
`length`	Length in elements of new vector view.

Return Value

Returns a pointer to the created subview vector view object, or *null* if the memory allocation for new object fails.

Restrictions

Errors

The arguments must conform to the following:

1. The vector view object must be valid.
2. The *length* must be greater than zero.
3. The subview must not extend beyond the bounds of the source view.

Notes/References

It is important for the application to check the return value for a memory allocation failure.

Note to Implementors: In *development mode*, the function updates the binding count (reference count) recorded in the block object.

Examples

See Also

`vsip_`*d*`vbindview_`*p*, `vsip_`*d*`vcloneview_`*p*, `vsip_`*d*`vcreateview_`*p*, `vsip_`*d*`vimagview_`*p*, `vsip_`*d*`vrealview_`*p*

Matrix View Object Functions

VSIPL *blocks* hold the data in a *data array*. A block can be *viewed* as a vector or a matrix. Two or more vector and/or matrix objects may reference the same *block*. VSIPL library functions make no distinction that is apparent to the application programmer between block objects which reference only *VSIPL data arrays* and block objects which reference *user data arrays* associated with an admitted block.

Matrix view objects have the attributes of *offset*, *col_stride*, *row_stride*, *col_length* and *row_length*.

VSIPL makes no distinction between row major or column major. The major direction is decided on matrix view creation by the selection of the strides, *row_stride* and *col_stride*. See `vsip_dmcreate_p` for more details.

Note: A matrix view object can be transposed by using a function to exchange the values of *row_stride*, *row_length*, *col_stride*, and *col_length*. This is not the same thing as transposing the underlying data in the block. The applications programmer needs to be aware that memory access in an inner loop (or implied in a library function) that access successive matrix elements with strides that are longer than cache lines are likely to experience a very significant performance degradation. There does not appear to be a satisfactory abstraction to hide this aspect of portability from the applications programmer for any computing system that has non-homogeneous average latency and bandwidth for accessing memory.

Matrix view object functions are provided to:

- Create (constructors) matrix view objects
- Create (constructors) vector view objects from matrix view objects
- Destroy (destructors) matrix view objects,
- Modify/manipulate matrix view objects, and
- Access functions for matrix view objects.

`vsip_dmalldestroy_p`	Destroy Matrix and Block
`vsip_dmbind_p`	Create and Bind a Matrix View
`vsip_dmcloneview_p`	Create Matrix View Clone
`vsip_dmcolview_p`	Create Column-View Matrix View
`vsip_dmcreate_p`	Create Matrix
`vsip_dmdestroy_p`	Destroy Matrix View
`vsip_dmdiagview_p`	Create Matrix Diagonal View
`vsip_dmget_p`	Matrix Get Element
`vsip_dmgetblock_p`	Matrix Get Block
`vsip_dmgetattrib_p`	Matrix Get View Attributes
`vsip_dmgetcollength_p`	Matrix Get Column Length
`vsip_dmgetcolstride_p`	Matrix Get Column Stride
`vsip_dmgetoffset_p`	Matrix Get Offset
`vsip_dmgetrowlength_p`	Matrix Get Row Length

vsip_*d*mgetrowstride_*p*	Matrix Get Row Stride
vsip_mimagview_*p*	Create Imaginary Matrix View
vsip_*d*mput_*p*	Matrix Put Element
vsip_*d*mputattrib_*p*	Matrix Put View Attributes
vsip_*d*mputcollength_*p*	Matrix Put Column Length
vsip_*d*mputcolstride_*p*	Matrix Put Column Stride
vsip_*d*mputoffset_*p*	Matrix Put Offset
vsip_*d*mputrowlength_*p*	Matrix Put Row Length
vsip_*d*mputrowstride_*p*	Matrix Put Row Stride
vsip_mrealview_*p*	Create Real Matrix View
vsip_*d*mrowview_*p*	Create Matrix Row View
vsip_*d*msubview_*p*	Create Sub-View Matrix View
vsip_*d*mtransview_*p*	Create Matrix Transposed View

vsip_*d*malldestroy_*p* Destroy Matrix and Block

Destroy (free) a matrix, its associated block, and any *VSIPL data* array bound to the block.

Functionality

Destroys (frees) a `matrix` view object, the block object to which it is bound, and any *VSIPL data* array. If **X** is a matrix of type `vsip_dmview_p` then

```
vsip_dmalldestroy_p(X);
```

is equivalent to

```
vsip_dblockdestroy_p(vsip_dmdestroy_p(X);
```

This is the complementary function to `vsip_dmcreate_p` and should only be used to destroy matrices that have only one view bound to the block object.

Prototypes

```
void vsip_malldestroy_f(
                     vsip_mview_f *X);

void vsip_malldestroy_i(
                     vsip_mview_i *X);

void vsip_cmalldestroy_f(
                     vsip_cmview_f *X);

void vsip_cmalldestroy_i(
                     vsip_cmview_i *X);

void vsip_malldestroy_bl(
                     vsip_mview_bl *X);
```

Arguments

`*X`	Matrix view object.

Return Value

None.

Restrictions

Errors

The arguments must conform to the following:

1. The matrix view object must be valid. An argument of *null* is not an error.
2. The specified matrix view must be the only view bound.
3. The matrix view must not be bound to a *derived* block (derived from a complex block).

Notes/References

If the matrix view is bound to a *derived* block, derived from a complex block, the complex block must be destroyed to free the block and associated data.

An argument of *null* is not an error.

Examples

Create and destroy a simple matrix of 10 by 15 elements.

```
vsip_mview_f *X =
vsip_mcreate_f((vsip_length)10,(vsip_length)15,VSIP_MEM_NONE);
...
vsip_malldestroy_f(X);
```

See Also

`vsip_blockdestroy_`*p*, `vsip_`*d*`mcreate_`*p*, `vsip_`*d*`mdestroy_`*p*

vsip_*d*mbind_*p* Create and Bind a Matrix View

Create a matrix view object and bind it to a block object.

Functionality

Creates a matrix object or returns *null* if it fails. If the view create is successful, it: (1) binds the matrix view object to the block object; (2) sets the ***offset*** from the beginning of the data array to the beginning of the matrix, the stride, *col_stride,* between scalar elements in a column, the number *col_length* of scalar elements in a column, the stride, *row_stride*, between scalar elements in a row, the number *row_length* of scalar elements in a row; and (3) then returns a pointer to the created matrix view object.

Prototypes

```
vsip_mview_f *vsip_mbind_f(
                      const vsip_block_f *block,
                      vsip_offset offset,
                      vsip_stride col_stride,
                      vsip_length col_length,
                      vsip_stride row_stride,
                      vsip_length row_length);
vsip_mview_i *vsip_mbind_i(
                      const vsip_block_i *block,
                      vsip_offset offset,
                      vsip_stride col_stride,
                      vsip_length col_length,
                      vsip_stride row_stride,
                      vsip_length row_length);
vsip_cmview_f *vsip_cmbind_f(
                      const vsip_cblock_f *block,
                      vsip_offset offset,
                      vsip_stride col_stride,
                      vsip_length col_length,
                      vsip_stride row_stride,
                      vsip_length row_length);
vsip_cmview_i *vsip_cmbind_i(
                      const vsip_cblock_i *block,
                      vsip_offset offset,
                      vsip_stride col_stride,
                      vsip_length col_length,
                      vsip_stride row_stride,
                      vsip_length row_length);
vsip_mview_bl *vsip_mbind_bl(
                      const vsip_block_bl *block,
                      vsip_offset offset,
                      vsip_stride col_stride,
                      vsip_length col_length,
                      vsip_stride row_stride,
                      vsip_length row_length);
```

Arguments

`*block`	Pointer to block object.
`offset`	Matrix view *offset* in elements relative to the base of block object.
`col_stride`	Matrix view *stride* between elements in a column.
`col_length`	Matrix view *length* in elements of a column.
`row_stride`	Matrix view *stride* between scalar elements in a row.
`row_length`	Matrix view *length* in elements of a row.

Return Value

Returns a pointer to the created matrix view object, or *null* if the memory allocation for new object fails.

Restrictions

Errors

The arguments must conform to the following:

1. The block object must be valid.
2. The *offset* must be less than the length of the block's data array.
3. The *row stride, row length, column stride, column length,* and *offset* arguments must not specify a matrix view that exceeds the bounds of the data array of the associated block.

Notes/References

It is important for the application to check the return value for a memory allocation failure.

Note to Implementors: In *development mode*, the function updates the bindings (reference count) recorded in the block object.

Examples

See Also

`vsip_`*d*`mcloneview_`*p*, `vsip_`*d*`mcolview_`*p*, `vsip_`*d*`mdiagview_`*p*, `vsip_`*d*`mimagview_`*p*, `vsip_`*d*`mrealview_`*p*, `vsip_`*d*`mrowview_`*p*, `vsip_`*d*`msubview_`*p*, `vsip_`*d*`mtransview_`*p*

vsip_*d*mcloneview_*p* Create Matrix View Clone

Create a clone of a matrix view.

Functionality

Creates a new `matrix` view object, copies all of the attributes of the source `matrix` view object to the new `matrix` view object, and then binds the new `matrix` view object to the block object of the source `matrix` view object. This function returns *null* on a memory allocation (creation) failure; otherwise, it returns a pointer to the new `matrix` view object.

Prototypes

```
vsip_mview_f *vsip_mcloneview_f(
                    const vsip_mview_f *X);
vsip_mview_i *vsip_mcloneview_i(
                    const vsip_mview_i *X);
vsip_cmview_f *vsip_cmcloneview_f(
                    const vsip_cmview_f *X);
vsip_cmview_i *vsip_cmcloneview_i(
                    const vsip_cmview_i *X);
vsip_mview_bl *vsip_mcloneview_bl(
                    const vsip_mview_bl *X);
```

Arguments

`*X` Source matrix view object.

Return Value

Returns a pointer to the created matrix view object clone, or *null* if the memory allocation for new object fails.

Restrictions

Errors

The arguments must conform to the following:

1. The matrix view object must be valid.

Notes/References

It is important for the application to check the return value for a memory allocation failure.

Note to Implementors: In *development mode*, the function updates the number of bindings (reference count) recorded in the block object.

Examples

See Also

`vsip_dmbindview_p`, `vsip_dmcolview_p`, `vsip_dmdiagview_p`, `vsip_dmimagview_p`, `vsip_dmrealview_p`, `vsip_dmrowview_p`, `vsip_dmsubview_p`, `vsip_dmtransview_p`

vsip_*d*mcolview_*p* Create Column-View Matrix View

Create a vector view object of a specified column of the source matrix view object.

Functionality

Creates a vector view object from a specified column of a matrix view object, or returns *null* if it fails. Otherwise, it binds the new vector view object to the same block object as the source matrix view object and sets its attributes to view just the specified column of the source matrix object.

Prototypes

```
vsip_vview_f *vsip_mcolview_f(
                    const vsip_mview_f *X,
                    vsip_index col_index);
vsip_vview_i *vsip_mcolview_i(
                    const vsip_mview_i *X,
                    vsip_index col_index);
vsip_cvview_f *vsip_cmcolview_f(
                    const vsip_cmview_f *X,
                    vsip_index col_index);
vsip_cvview_i *vsip_cmcolview_i(
                    const vsip_cmview_i *X,
                    vsip_index col_index);
vsip_vview_bl *vsip_mcolview_bl(
                    const vsip_mview_bl *X,
                    vsip_index col_index);
```

Arguments

`*X`	Source matrix view object.
`col_index`	*Column index* of source matrix view object.

Return Value

Returns a pointer to the created column vector view object, or *null* if the memory allocation for new object fails.

Restrictions

Errors

Notes/References

The arguments must conform to the following:

1. The matrix view object must be valid.
2. The requested column index must be a valid column index of the source matrix view.

Examples

See Also

`vsip_dmbindview_p`, `vsip_dmcloneview_p`, `vsip_dmdiagview_p`, `vsip_dmimagview_p`, `vsip_dmrealview_p`, `vsip_dmrowview_p`, `vsip_dmsubview_p`, and `vsip_dmtransview_p`

vsip_*d*mcreate_*p* Create Matrix

Creates a block object and matrix view (object) of the block.

Functionality

Creates a block object with an $M \cdot N$ element *VSIPL data* array, it creates an M by N dense matrix view object and then binds the block object to it.

The function

```
vsip_mview_p *vsip_mcreate_p(M, N, VSIP_ROW, hint);
```

returns the same result as

```
vsip_mbind_p(vsip_blockcreate_p(M*N, hint),(vsip_offset)0,
   (vsip_stride)N, (vsip_length)M,     column stride, column length
   (vsip_stride)1, (vsip_length)N);    row stride, row length
```

or

```
vsip_mview_p *vsip_mcreate_p(M, N, VSIP_COL, hint);
```

returns the same result as

```
vsip_mbind_p(vsip_blockcreate_p(M*N, hint),(vsip_offset)0,
   (vsip_stride)1, (vsip_length)M,     column stride, column length
   (vsip_stride)M, (vsip_length)N);    row stride, row length
```

except that `vsip_mcreate_p` returns a *null* if `vsip_blockcreate_p(M*N, hint)` returns a *null*.

Prototypes

```
vsip_mview_f *vsip_mcreate_f(
                     vsip_length M,
                     vsip_length N,
                     vsip_major major,
                     vsip_memory_hint hint);
vsip_mview_i *vsip_mcreate_i(
                     vsip_length M,
                     vsip_length N,
                     vsip_major major,
                     vsip_memory_hint hint);
vsip_cmview_f *vsip_cmcreate_f(
                     vsip_length M,
                     vsip_length N,
                     vsip_major major,
                     vsip_memory_hint hint);
vsip_cmview_i *vsip_cmcreate_i(
                     vsip_length M,
                     vsip_length N,
                     vsip_major major,
                     vsip_memory_hint hint);
vsip_mview_bl *vsip_mcreate_bl(
                     vsip_length M,
                     vsip_length N,
                     vsip_major major,
                     vsip_memory_hint hint);
```

Arguments

`M` Number of rows of the matrix view (*column length*).

`N` Number of columns of the matrix view (*row length*).

`major` Row or Column major

```
typedef enum {
  VSIP_ROW = 0,                    Row
  VSIP_COL = 1                     Column
} vsip_major;
```

`hint` Memory hint

```
typedef enum {
  VSIP_MEM_NONE = 0,               No hint
  VSIP_MEM_RDONLY = 1,             Read Only
  VSIP_MEM_CONST = 2,              Constant
  VSIP_MEM_SHARED = 3,             Shared
  VSIP_MEM_SHARED_RDONLY = 4,      Shared, Read Only
  VSIP_MEM_SHARED_CONST = 5        Shared, Constant
} vsip_memory_hint;
```

Return Value

Returns a pointer to the created matrix view object, or *null* if it fails.

Restrictions

Errors

The arguments must conform to the following:

1. The lengths, *N* and *M*, must be greater than zero.
2. The *major* memory direction must be a valid member of the `vsip_major` enumeration.
3. The memory *hint* must be a valid member of the `vsip_memory_hint` enumeration.

Notes/References

Note to Implementors: In *development mode,* it should also update the bindings (reference count) recorded in the block object.

Examples

See Also

vsip_*d*mdestroy_*p* Destroy Matrix View

Destroy (free) a matrix view object and returns a pointer to the associated block object.

Functionality

Frees a matrix view object from the block object that it was bound to, destroys the matrix view object, and then returns a pointer to the block object. If the matrix view argument is *null*, it returns *null*.

Prototypes

```
vsip_block_f *vsip_mdestroy_f(
                    vsip_mview_f *X);
vsip_block_i *vsip_mdestroy_i(
                    vsip_mview_i *X);
vsip_cblock_f *vsip_cmdestroy_f(
                    vsip_cmview_f *X);
vsip_cblock_i *vsip_cmdestroy_i(
                    vsip_cmview_i *X);
vsip_block_bl *vsip_mdestroy_bl(
                    vsip_mview_bl *X);
```

Arguments

`*X` Matrix view object.

Return Value

Returns a pointer to the block object to which the matrix view was bound, or *null* if the calling argument was *null*.

Restrictions

Errors

The arguments must conform to the following:

1. The matrix view object must be valid. An argument of *null* is not an error.

Notes/References

An argument of *null* is not an error.

Note to Implementors: In *development mode,* the function updates the bindings (reference count) recorded in the block object.

Examples

See Also

vsip_*d*mdiagview_*p* Create Matrix Diagonal View

Create a vector view object of a matrix diagonal of a matrix view object

Functionality

Creates a vector view object of a specified diagonal of a matrix view object, or returns *null* if it fails. On success, it binds the new vector view object to the same block object as the source matrix view object and sets its attributes to view just the specified diagonal of the source matrix object. An *index* of '0' specifies the main diagonal, positive indices are above the main diagonal, and negative indices are below the main diagonal.

Prototypes

```
vsip_vview_f *vsip_mdiagview_f(
                    const vsip_mview_f *X,
                    vsip_stride index);
vsip_vview_i *vsip_mdiagview_i(
                    const vsip_mview_i *X,
                    vsip_stride index);
vsip_cvview_f *vsip_cmdiagview_f(
                    const vsip_cmview_f *X,
                    vsip_stride index);
vsip_cvview_i *vsip_cmdiagview_i(
                    const vsip_cmview_i *X,
                    vsip_stride index);
vsip_vview_bl *vsip_mdiagview_bl(
                    const vsip_mview_bl *X,
                    vsip_stride index);
```

Arguments

`*X`	Source matrix view object.
`index`	*Index* of diagonal: 0 main, + above, - below.

Return Value

Returns a pointer to the created diagonal vector view object, or *null* if the memory allocation for new object fails.

Restrictions

Errors

The arguments must conform to the following:

1. The matrix view object must be valid.
2. The *index* must specify a valid diagonal. For positive indices, *index* must be less than the number of column, for negative indices, the |*index*| must be less than the number of rows.

Notes/References

It is important for the application to check the return value for a memory allocation failure.

The *index* is of type `vsip_stride` since this is a portable signed integer of sufficient size to index any supported matrix view.

Examples

See Also

`vsip_`*d*`mbind_`*p*, `vsip_`*d*`mcloneview_`*p*, `vsip_`*d*`mcolview_`*p*, `vsip_`*d*`mimagview_`*p*, `vsip_`*d*`mrealview_`*p*, `vsip_`*d*`mrowview_`*p*, `vsip_`*d*`msubview_`*p*, and `vsip_`*d*`mtransview_`*p*

vsip_*d*mget_*p* Matrix Get Element

Get the value of a specified element of a matrix view object.

Functionality

Returns the value of the specified element of a matrix view object.

Returns $x_{i,j}$

Prototypes

```
vsip_scalar_f vsip_mget_f(
                      const vsip_mview_f *X,
                      vsip_index i,
                      vsip_index j);
vsip_scalar_i vsip_mget_i(
                      const vsip_mview_i *X,
                      vsip_index i,
                      vsip_index j);
vsip_cscalar_f vsip_cmget_f(
                      const vsip_cmview_f *X,
                      vsip_index i,
                      vsip_index j);
vsip_cscalar_i vsip_cmget_i(
                      const vsip_cmview_i *X,
                      vsip_index i,
                      vsip_index j);
vsip_scalar_bl vsip_mget_bl(
                      const vsip_mview_bl *X,
                      vsip_index i,
                      vsip_index j);
```

Arguments

`*X`	Vector/matrix/tensor view object
`i`	Matrix index *i* of (*i, j*), the row index.
`j`	Matrix index *j* of (*i, j*), the column index

Return Value

Returns the value of the specified element of a matrix view object.

Restrictions

Errors

The arguments must conform to the following:

1. The matrix view object must be valid.
2. The index must be a valid index of the matrix view.

Notes/References

Examples

See Also

vsip_*ds*put_*p*

vsip_*d*mgetattrib_*p* Matrix Get View Attributes

Get the attributes of a matrix view object.

Functionality

Retrieves the attributes of matrix view object: (pointer to) bound block object, *offset*, *col_stride*, *col_length*, *row_stride*, and *row_length*.

Prototypes

```
void vsip_mgetattrib_f(
                      vsip_mview_f *X,
                      vsip_mattr_f *attrib);
void vsip_mgetattrib_i(
                      vsip_mview_i *X,
                      vsip_mattr_i *attrib);
void vsip_cmgetattrib_f(
                      vsip_cmview_f *X,
                      vsip_cmattr_f *attrib);
void vsip_cmgetattrib_i(
                      vsip_cmview_i *X,
                      vsip_cmattr_i *attrib);
void vsip_mgetattrib_bl(
                      vsip_mview_bl *X,
                      vsip_mattr_bl *attrib);
typedef struct {
   vsip_offset offset;
   vsip_stride row_stride;
   vsip_length row_length;
   vsip_stride col_stride;
   vsip_length col_length;
   vsip_block_p *block;      Get only, ignored on Put
} vsip_mattr_p;
typedef struct {
   vsip_offset offset;
   vsip_stride row_stride;
   vsip_length row_length;
   vsip_stride col_stride;
   vsip_length col_length;
   vsip_cblock_p *block;     Get only, ignored on Put
} vsip_cmattr_p;
```

Arguments

`*X`	Matrix view object.
`*attrib`	Pointer to output matrix attribute structure.

Return Value

None.

Restrictions

Errors

The arguments must conform to the following:

1. The matrix view object must be valid.
2. The pointer to the matrix attribute structure must be valid – non-*null*.

Notes/References

The block attribute can be read (get), but cannot be set.

Examples

See Also

vsip_*d*mgetblock_*p* Matrix Get Block

Get the block attribute of a matrix view object

Functionality

Returns a pointer to the *VSIPL block* object to which the `matrix` view object is bound.

Prototypes

```
vsip_block_f *vsip_mgetblock_f(
                    const vsip_mview_f *X);
vsip_block_i *vsip_mgetblock_i(
                    const vsip_mview_i *X);
vsip_cblock_f *vsip_cmgetblock_f(
                    const vsip_cmview_f *X);
vsip_cblock_i *vsip_cmgetblock_i(
                    const vsip_cmview_i *X);
vsip_block_bl *vsip_mgetblock_bl(
                    const vsip_mview_bl *X);
```

Arguments

`*X` Matrix view object.

Return Value

Returns a pointer to the block object to which the matrix view object is bound.

Restrictions

Errors

The arguments must conform to the following:

1. The matrix view object must be valid.

Notes/References

The functions `vsip_dvgetattrib_p` and `vsip_dvputattrib_p` are not symmetric since you can *get* the block object but you cannot *put* the block object.

Examples

See Also

vsip_*d*mgetcollength_*p* Matrix Get Column Length

Get the *column length* attribute of a matrix view object

Functionality

Returns the *length* of (number of elements along) a *column* of a matrix view.

Prototypes

```
vsip_length vsip_mgetcollength_f(
                      const vsip_mview_f *X);
vsip_length vsip_mgetcollength_i(
                      const vsip_mview_i *X);
vsip_length vsip_cmgetcollength_f(
                      const vsip_cmview_f *X);
vsip_length vsip_cmgetcollength_i(
                      const vsip_cmview_i *X);
vsip_length vsip_mgetcollength_bl(
                      const vsip_mview_bl *X);
```

Arguments

`*X` Matrix view object.

Return Value

Returns the value of the *col_length* attribute of a matrix view object.

Restrictions

Errors

The following cause a VSIPL runtime error in *development mode;* in *production mode* the results will be implementation dependent.

1. The matrix view is invalid.

Notes/References

Examples

See Also

vsip_*d*mgetcolstride_*p* Matrix Get Column Stride

Get the *column stride* attribute of a matrix view object

Functionality

Returns the *stride* (in elements of the bound block) between successive scalar elements along a *column* of a matrix view.

Prototypes

```
vsip_stride vsip_mgetcolstride_f(
                    const vsip_mview_f *X);
vsip_stride vsip_mgetcolstride_i(
                    const vsip_mview_i *X);
vsip_stride vsip_cmgetcolstride_f(
                    const vsip_cmview_f *X);
vsip_stride vsip_cmgetcolstride_i(
                    const vsip_cmview_i *X);
vsip_stride vsip_mgetcolstride_bl(
                    const vsip_mview_bl *X);
```

Arguments

`*X`	Matrix view object.

Return Value

Returns the value of the *column stride* attribute of the matrix view object.

Restrictions

Errors

The arguments must conform to the following:

1. The matrix view object must be valid.

Notes/References

Examples

See Also

vsip_*d*mgetoffset_*p* Matrix Get Offset

Get the *offset* attribute of a matrix view object.

Functionality

Returns the *offset* (in elements) to the first scalar element of a matrix view from the start of the block object to which it is bound.

Prototypes

```
vsip_offset vsip_mgetoffset_f(
                    const vsip_mview_f *X);
vsip_offset vsip_mgetoffset_i(
                    const vsip_mview_i *X);
vsip_offset vsip_cmgetoffset_f(
                    const vsip_cmview_f *X);
vsip_offset vsip_cmgetoffset_i(
                    const vsip_cmview_i *X);
vsip_offset vsip_mgetoffset_bl(
                    const vsip_mview_bl *X);
```

Arguments

`*X` Matrix view object.

Return Value

Returns the value of the *offset* attribute of the matrix view object.

Restrictions

Errors

The arguments must conform to the following:

1. The matrix view object must be valid.

Notes/References

Examples

See Also

vsip_*d*mgetrowlength_*p* Matrix Get Row Length

Get the *row length* attribute of a matrix view object

Functionality

Returns the *length* of (number of elements along) a *row* of a matrix view.

Prototypes

```
vsip_length vsip_mgetrowlength_f(
                      const vsip_mview_f *X);
vsip_length vsip_mgetrowlength_i(
                      const vsip_mview_i *X);
vsip_length vsip_cmgetrowlength_f(
                      const vsip_cmview_f *X);
vsip_length vsip_cmgetrowlength_i(
                      const vsip_cmview_i *X);
vsip_length vsip_mgetrowlength_bl(
                      const vsip_mview_bl *X);
```

Arguments

`*X` Matrix view object.

Return Value

Returns the value of the *row_length* attribute of a matrix view object.

Restrictions

Errors

The following cause a VSIPL runtime error in *development mode;* in *production mode* the results will be implementation dependent.

1. The matrix view object is invalid.

Notes/References

Examples

See Also

vsip_*d*mgetrowstride_*p* Matrix Get Row Stride

Get the *row stride* attribute of a matrix view object

Functionality

Returns the *stride* (in elements of the bound block) between successive scalar elements along a *row* of a matrix view.

Prototypes

```
vsip_stride vsip_mgetrowstride_f(
                      const vsip_mview_f *X);
vsip_stride vsip_mgetrowstride_i(
                      const vsip_mview_i *X);
vsip_stride vsip_cmgetrowstride_f(
                      const vsip_cmview_f *X);
vsip_stride vsip_cmgetrowstride_i(
                      const vsip_cmview_i *X);
vsip_stride vsip_mgetrowstride_bl(
                      const vsip_mview_bl *X);
```

Arguments

`*X` Matrix view object.

Return Value

Returns the value of the *row stride* attribute of the matrix view object.

Restrictions

Errors

The arguments must conform to the following:

1. The matrix view object must be valid.

Notes/References

Examples

See Also

vsip_mimagview_*p* Create Imaginary Matrix View

Create a matrix view object of the imaginary part of complex matrix from a complex matrix view object.

Functionality

Creates a real `matrix` view object from the "`imaginary` part of a complex" `matrix` view object, or returns *null* if it fails.

On success the function creates a *derived* block object, derived from the complex block object, that is bound to the `imaginary` data part of the original complex block and then binds a real `matrix` view object to the block. The new `matrix` encompasses the `imaginary` part of the source complex `matrix`.

Prototypes

```
vsip_mview_f *vsip_mimagview_f(
                      const vsip_cmview_f *X);
vsip_mview_i *vsip_mimagview_i(
                      const vsip_cmview_i *X);
```

Arguments

`*X` Source matrix view object.

Return Value

Returns a pointer to the created "imaginary" part matrix view object, or *null* if the memory allocation for new object fails.

Restrictions

The *derived* block object, derived from the complex block object cannot be destroyed or released. The parent complex block object may be released (if it is bound to user data), and destroying the complex block is the only way to free the memory associated with the *derived* block object.

Errors

The arguments must conform to the following:

1. The complex matrix view object must be valid.

Notes/References

It is important for the application to check the return value for a memory allocation failure.

This function should not be confused with the function `vsip_simag_p()` which is a copy operator (copies the imaginary data).

There are no requirements on *offset* or *stride* of a real view on its *derived* block. By using `vsip_mgetattrib_p,` information about the layout of the view on the block may be obtained.

CAUTION. Using attribute information, and the block bound to the matrix, to bind new matrixes outside the data space of the original matrix produced by `vsip_simagview_p` will produce non-portable code. Portable code may be produced by: (1) remaining inside the data space of the matrix, (2) by not assuming a set relationship of strides and offsets, and (3) by using the get attributes functions to obtain necessary information within the application code to understand the layout for each implementation.

Note to Implementors:

- The resulting *derived* block must have a property which prevents it from being released or destroyed.
- In *development mode,* block binding count (reference count) recorded in the block object is incremented.

Examples

See Also

`vsip_dmbindview_p`, `vsip_dmcloneview_p`, `vsip_dmcolview_p`, `vsip_dmdiagview_p`, `vsip_dmrealview_p`, `vsip_dmrowview_p`, `vsip_dmsubview_p`, `vsip_dmtransview_p`, and `vsip_simag_p`

vsip_*d*mput_*p* Matrix Put Element

Put (Set) the value of a specified element of a matrix view object

Functionality

Puts (sets) the value of the specified element of a matrix view object.

$y_{i,j} \leftarrow x$

Prototypes

```
void vsip_mput_f(
                     const vsip_mview_f *y,
                     vsip_index i,
                     vsip_index j,
                     vsip_scalar_f x);
void vsip_mput_i(
                     const vsip_mview_i *y,
                     vsip_index i,
                     vsip_index j,
                     vsip_scalar_i x);
void vsip_cmput_f(
                     const vsip_cmview_f *y,
                     vsip_index i,
                     vsip_index j,
                     vsip_cscalar_f x);
void vsip_cmput_i(
                     const vsip_cmview_i *y,
                     vsip_index i,
                     vsip_index j,
                     vsip_cscalar_i x);
void vsip_mput_bl(
                     const vsip_mview_bl *y,
                     vsip_index i,
                     vsip_index j,
                     vsip_scalar_bl x);
```

Arguments

`*y`	Matrix view object of destination.
`i`	Matrix index *i* of (*i, j*), the row index.
`j`	Matrix index *j* of (*i, j*), the column index.
`x`	Scalar value to put in matrix.

Return Value

None.

Restrictions

Errors

The arguments must conform to the following:

1. The matrix view object must be valid.
2. The index must be a valid index of the matrix view.

Notes/References

Examples

See Also

`vsip_`*`ds`*`get_`*`p`*

vsip_*d*mputattrib_*p* Matrix Put View Attributes

Put (Set) the attributes of a matrix view object.

Functionality

Stores the matrix view attributes of: *offset*, *column stride*, *column length*, *row stride*, and *row length*, of a matrix view object, and as a programmer convenience, returns a pointer to the matrix view object.

Prototypes

```
vsip_mview_f *vsip_mputattrib_f(
                    vsip_mview_f *X,
                    const vsip_mattr_f *attrib);
vsip_mview_i *vsip_mputattrib_i(
                    vsip_mview_i *X,
                    const vsip_mattr_i *attrib);
vsip_cmview_f *vsip_cmputattrib_f(
                    vsip_cmview_f *X,
                    const vsip_cmattr_f *attrib);
vsip_cmview_i *vsip_cmputattrib_i
                    vsip_cmview_i *X,
                    const vsip_cmattr_i *attrib);
vsip_mview_bl *vsip_mputattrib_bl(
                    vsip_mview_bl *X,
                    const vsip_mattr_bl *attrib);
typedef struct {
   vsip_offset offset;
   vsip_stride row_stride;
   vsip_length row_length;
   vsip_stride col_stride;
   vsip_length col_length;
   vsip_block_p *block;      Get only, ignored on Put
} vsip_mattr_p;
typedef struct {
   vsip_offset offset;
   vsip_stride row_stride;
   vsip_length row_length;
   vsip_stride col_stride;
   vsip_length col_length;
   vsip_cblock_p *block;     Get only, ignored on Put
} vsip_cmattr_p;
```

Arguments

`*X`	Matrix view object.
`*attrib`	Pointer to a matrix attribute structure.

Return Value

Returns a pointer to the source matrix view object as a programming convenience.

Restrictions

Errors

The arguments must conform to the following:

1. The matrix view object must be valid.
2. The pointer to the matrix attribute structure must be valid – non-*null*.
3. The *offset*, *column stride*, *column length*, *row stride*, and *row length* arguments must not specify a matrix view that exceeds the bounds of the data array of the associated block.

Notes/References

The functions `vsip_mgetattrib_p` and `vsip_mputattrib_p` are not symmetric since you can "get" the block object but you cannot "put" the block object.

Examples

See Also

vsip_*d*mputcollength_*p* Matrix Put Column Length

Put (Set) the *column length* attribute of a matrix view object.

Functionality

Puts (sets) the *length* of (number of elements along) a *column* of a matrix view.

Prototypes

```
vsip_mview_f *vsip_mputcollength_f(
                    const vsip_mview_f *X,
                    vsip_length length);
vsip_mview_i *vsip_mputcollength_i(
                    const vsip_mview_i *X,
                    vsip_length length);
vsip_mview_f *vsip_cmputcollength_f(
                    const vsip_cmview_f *X,
                    vsip_length length);
vsip_mview_i *vsip_cmputcollength_i(
                    const vsip_cmview_i *X,
                    vsip_length length);
vsip_mview_bl *vsip_mputcollength_bl(
                    const vsip_mview_bl *X,
                    vsip_length length);
```

Arguments

`*X`	Matrix view object.
`length`	Column *length* in elements.

Return Value

Returns a pointer to the source matrix view object as a programming convenience.

Restrictions

Errors

The arguments must conform to the following:

1. The matrix view object must be valid.
2. The *length* must be greater than zero.
3. The *length* argument must not specify a matrix view that exceeds the bounds of the data array of the associated block.

Notes/References

Examples

See Also

vsip_*d*mputcolstride_*p* Matrix Put Column Stride

Put (Set) the *column stride* attribute of a matrix view object.

Functionality

Puts (sets) the *stride* (in elements of the bound block) between successive scalar elements along a *column* of a matrix view.

Prototypes

```
vsip_mview_f *vsip_mputcolstride_f(
                    const vsip_mview_f *X,
                    vsip_stride stride);
vsip_mview_i *vsip_mputcolstride_i(
                    const vsip_mview_i *X,
                    vsip_stride stride);
vsip_mview_f *vsip_cmputcolstride_f(
                    const vsip_cmview_f *X,
                    vsip_stride stride);
vsip_mview_i *vsip_cmputcolstride_i(
                    const vsip_cmview_i *X,
                    vsip_stride stride);
vsip_mview_bl *vsip_mputcolstride_bl(
                    const vsip_mview_bl *X,
                    vsip_stride stride);
```

Arguments

`*X`	Matrix view object.
`stride`	Column *stride* in elements.

Return Value

Returns a pointer to the source matrix view object as a programming convenience.

Restrictions

Errors

The arguments must conform to the following:

1. The matrix view object must be valid.
2. The *stride* argument must not specify a matrix view that exceeds the bounds of the data array of the associated block.

Notes/References

A column *stride* of zero may be used to define a matrix view where each column is filled with a constant.

Examples

See Also

vsip_*d*mputoffset_*p* Matrix Put Offset

Put (Set) the *offset* attribute of a matrix view object.

Functionality

Puts (sets) the *offset* (in elements) to the first scalar element of a matrix view, from the start of the block, to which it is bound.

Prototypes

```
vsip_mview_f *vsip_mputoffset_f(
                    const vsip_mview_f *X,
                    vsip_offset offset);
vsip_mview_i *vsip_mputoffset_i(
                    const vsip_mview_i *X,
                    vsip_offset offset);
vsip_mview_f *vsip_cmputoffset_f(
                    const vsip_cmview_f *X,
                    vsip_offset offset);
vsip_mview_i *vsip_cmputoffset_i(
                    const vsip_cmview_i *X,
                    vsip_offset offset);
vsip_mview_bl *vsip_mputoffset_bl(
                    const vsip_mview_bl *X,
                    vsip_offset offset);
```

Arguments

`*X`	Matrix view object.
`offset`	Offset in elements relative to the start of the block object.

Return Value

Returns a pointer to the source matrix view object as a programming convenience.

Restrictions

Errors

The arguments must conform to the following:

1. The matrix view object must be valid.
2. The *offset* argument must not specify a matrix view that exceeds the bounds of the data array of the associated block.

Notes/References

Examples

See Also

vsip_*d*mputrowlength_*p* Matrix Put Row Length

Put (Set) the *row length* attribute of a matrix view object.

Functionality

Puts (sets) the *length* of (number of elements along) a *row* of a matrix view.

Prototypes

```
vsip_mview_f *vsip_mputrowlength_f(
                    const vsip_mview_f *X,
                    vsip_length length);
vsip_mview_i *vsip_mputrowlength_i(
                    const vsip_mview_i *X,
                    vsip_length length);
vsip_mview_f *vsip_cmputrowlength_f(
                    const vsip_cmview_f *X,
                    vsip_length length);
vsip_mview_i *vsip_cmputrowlength_i(
                    const vsip_cmview_i *X,
                    vsip_length length);
vsip_mview_bl *vsip_mputrowlength_bl(
                    const vsip_mview_bl *X,
                    vsip_length length);
```

Arguments

`*X`	Matrix view object.
`length`	Row *length* in elements.

Return Value

Returns a pointer to the source matrix view object as a programming convenience.

Restrictions

Errors

The arguments must conform to the following:

1. The matrix view object must be valid.
2. The *length* must be greater than zero.
3. The *length* argument must not specify a matrix view that exceeds the bounds of the data array of the associated block.

Notes/References

Examples

See Also

vsip_*d*mputrowstride_*p* Matrix Put Row Stride

Put (Set) the *row stride* attribute of a matrix view object.

Functionality

Puts (sets) the *stride* (in elements of the bound block) between successive scalar elements along a *row* of a matrix view.

Prototypes

```
vsip_mview_f *vsip_mputrowstride_f(
                    const vsip_mview_f *X,
                    vsip_stride stride);
vsip_mview_i *vsip_mputrowstride_i(
                    const vsip_mview_i *X,
                    vsip_stride stride);
vsip_mview_f *vsip_cmputrowstride_f(
                    const vsip_cmview_f *X,
                    vsip_stride stride);
vsip_mview_i *vsip_cmputrowstride_i(
                    const vsip_cmview_i *X,
                    vsip_stride stride);
vsip_mview_bl *vsip_mputrowstride_bl(
                    const vsip_mview_bl *X,
                    vsip_stride stride);
```

Arguments

`*X`	Matrix view object.
`stride`	Row *stride* in elements.

Return Value

Returns a pointer to the source matrix view object as a programming convenience.

Restrictions

Errors

The arguments must conform to the following:

1. The matrix view object must be valid.
2. The *stride* argument must not specify a matrix view that exceeds the bounds of the data array of the associated block.

Notes/References

A *row stride* of zero may be used to define a matrix view where each row is filled with a constant.

Examples

See Also

vsip_mrealview_*p* Create Real Matrix View

Create a matrix view object of the real part of complex matrix from a complex matrix view object.

Functionality

Creates a real `matrix` view object from the "real part of a complex" `matrix` view object, or returns *null* if it fails.

On success, the function creates a *derived* block object (derived from the complex block object). The derived block object is bound to the real data part of the original complex block and then binds a real `matrix` view object to the block. The new `matrix` encompasses the real part of the input complex `matrix`.

Prototypes

```
vsip_mview_f *vsip_mrealview_f(
                    const vsip_cmview_f *X);
vsip_mview_i *vsip_mrealview_i(
                    const vsip_cmview_i *X);
```

Arguments

`*X` Source matrix view object.

Return Value

Returns a pointer to the created "real" part matrix view object, or *null* if the memory allocation for new object fails.

Restrictions

The *derived* block object, derived from the complex block object cannot be destroyed or released. The parent complex block object may be released (if it is bound to user data), and destroying the complex block is the only way to free the memory associated with the *derived* block object.

Errors

The arguments must conform to the following:

1. The complex matrix view object must be valid.

Notes/References

It is important for the application to check the return value for a memory allocation failure.

This function should not be confused with the function `vsip_sreal_p()` which is a copy operator (copies the real data).

There are no requirements on *offset* or *stride* of a real view on its *derived* block. By using `vsip_mgetattrib_p` information about the layout of the view on the block may be obtained.

CAUTION. Using attribute information, and the block bound to the matrix, to bind new matrixes outside the data space of the original matrix produced by `vsip_srealview_p` will produce non-portable code. Portable code may be produced by: (1) remaining inside the data space of the matrix, (2) by not assuming a set relationship of strides and offsets, and (3) by using the get attributes functions to obtain necessary information within the application code to understand the layout for each implementation.

Note to Implementors:

- The resulting *derived* block must have a property which prevents it from being released or destroyed.
- In *development mode,* block binding count (reference count) recorded in the block object is incremented.

Examples

See Also

`vsip_dmbindview_p`, `vsip_dmcloneview_p`, `vsip_dmcolview_p`, `vsip_dmdiagview_p`, `vsip_dmimagview_p`, `vsip_dmrowview_p`, `vsip_dmsubview_p`, `vsip_dmtransview_p`, and `vsip_sreal_p`

vsip_*d*mrowview_*p* Create Matrix Row View

Create a vector view object of a specified row of the source matrix view object.

Functionality

Creates a vector view object from a specified row of a matrix view object, or returns *null* if it fails. On success, it binds the new vector view object to the same block object as the source matrix view object and sets its attributes to view just the specified row of the source matrix object.

Prototypes

```
vsip_vview_f *vsip_mrowview_f(
                    const vsip_mview_f *X,
                    vsip_index row_index);
vsip_vview_i *vsip_mrowview_i(
                    const vsip_mview_i *X,
                    vsip_index row_index);
vsip_cvview_f *vsip_cmrowview_f(
                    const vsip_cmview_f *X,
                    vsip_index row_index);
vsip_cvview_i *vsip_cmrowview_i(
                    const vsip_cmview_i *X,
                    vsip_index row_index);
vsip_vview_bl *vsip_mrowview_bl(
                    const vsip_mview_bl *X,
                    vsip_index row_index);
```

Arguments

`*X`	Source matrix view object.
`row_index`	*Row index* of source matrix view object.

Return Value

Returns a pointer to the created row vector view object, or *null* if the memory allocation for new object fails.

Restrictions

Errors

The arguments must conform to the following:

1. The matrix view object must be valid.
2. The requested row index must be a valid row index of the source matrix view.

Notes/References

It is important for the application to check the return value for a memory allocation failure.

Examples

See Also

`vsip_dmbind_p`, `vsip_dmcloneview_p`, `vsip_dmcolview_p`, `vsip_dmdiagview_p`, `vsip_dmimagview_p`, `vsip_dmrealview_p`, `vsip_dmsubview_p`, and `vsip_dmtransview_p`

vsip_*d*msubview_*p* Create Subview Matrix View

Create a matrix view object that is a subview of matrix view object.

Functionality

Creates a matrix view object from a subview of a matrix view, or returns *null* if it fails. The subview is an *M* by *N* matrix view whose (0,0) element corresponds with the (*row index*, *column index*) element of the source matrix view

(The subview is relative to the source view, *row stride* and *column stride* are inherited from the source view).

Prototypes

```
vsip_mview_f *vsip_msubview_f(
                      const vsip_mview_f *X,
                      vsip_index row_index,
                      vsip_index col_index,
                      vsip_length M,
                      vsip_length N);

vsip_mview_i *vsip_msubview_i(
                      const vsip_mview_i *X,
                      vsip_index row_index,
                      vsip_index col_index,
                      vsip_length M,
                      vsip_length N);

vsip_cmview_f *vsip_cmsubview_f(
                      const vsip_cmview_f *X,
                      vsip_index row_index,
                      vsip_index col_index,
                      vsip_length M,
                      vsip_length N);

vsip_cmview_i *vsip_cmsubview_i(
                      const vsip_cmview_i *X,
                      vsip_index row_index,
                      vsip_index col_index,
                      vsip_length M,
                      vsip_length N);
```

```
vsip_mview_bl *vsip_msubview_bl(
                    const vsip_mview_bl *X,
                    vsip_index row_index,
                    vsip_index col_index,
                    vsip_length M,
                    vsip_length N);
```

Arguments

`*X`	Source matrix view object.
`row_index`	The index (*row index*, *column index*) of the source matrix view object is mapped to the index (0,0) of the submatrix view object.
`col_index`	The index (*row index*, *column index*) of the source matrix view object is mapped to the index (0,0) of the submatrix view object.
`M`	Number of rows of the matrix view (*column length*).
`N`	Number of columns of the matrix view (*row length*).

Return Value

Returns a pointer to the created subview matrix view object, or *null* if the memory allocation for new object fails.

Restrictions

Errors

The arguments must conform to the following:

1. The matrix view object must be valid.
2. The matrix index (*row index*, *column index*) must be a valid index of the matrix view.
3. The subview must not extend beyond the bounds of the source matrix view.

Notes/References

It is important for the application to check the return value for a memory allocation failure.

Implementor Note: In *development mode,* it should also increment the number of bindings (reference count) recorded in the block object.

Examples

See Also

`vsip_`*d*`mbind_`*p*, `vsip_`*d*`mcloneview_`*p*, `vsip_`*d*`mcolview_`*p*, `vsip_`*d*`mdiagview_`*p*, `vsip_`*d*`mimagview_`*p*, `vsip_`*d*`mrealview_`*p*, `vsip_`*d*`mrowview_`*p*, `vsip_`*d*`mtransview_`*p*

vsip_*d*mtransview_*p* Create Matrix Transposed View

Create a matrix view object that is the transpose of a matrix view object.

Functionality

Creates a matrix view object that provides a transposed view of a specified a matrix view, or returns *null* if it fails. On success, it binds the new matrix view object to the same block object as the source matrix view object and sets its attributes to view the transpose of the source matrix object.

Prototypes

```
vsip_mview_f *vsip_mtransview_f(
                    const vsip_mview_f *X);
vsip_mview_i *vsip_mtransview_i(
                    const vsip_mview_i *X);
vsip_cmview_f *vsip_cmtransview_f(
                    const vsip_cmview_f *X);
vsip_cmview_i *vsip_cmtransview_i(
                    const vsip_cmview_i *X);
vsip_mview_bl *vsip_mtransview_bl(
                    const vsip_mview_bl *X);
```

Arguments

`*X` Source matrix view object.

Return Value

Returns a pointer to the created transposed matrix view object, or *null* if the memory allocation for new object fails.

Restrictions

Errors

The arguments must conform to the following:

1. The matrix view object must be valid.

Notes/References

This should not be confused with the function `vsip_dtranspose_p()` which transposes the underlying data of the matrix block object. Inner loop memory accesses are more efficient on most processors if they are accessing memory with small (unit) strides between memory elements. Use this information to guide the selection of transpose method.

It is important for the application to check the return value for a memory allocation failure.

Examples

See Also

`vsip_dmbind_p`, `vsip_dmcloneview_p`, `vsip_dmcolview_p`, `vsip_dmdiagview_p`, `vsip_dmimagview_p`, `vsip_dmrealview_p`, `vsip_dmrowview_p`, and `vsip_dmtransview_p`
`vsip_dtranspose_p`

Tensor Views

vsip_**dtalldestroy**_*p*	Destroy Tensor And Block
vsip_**dtbind**_*p*	Create and Bind a Tensor View
vsip_**dtcloneview**_*p*	Create Tensor View Clone
vsip_**dtcreate**_*p*	Create Tensor
vsip_**dtdestroy**_*p*	Destroy Tensor View
vsip_**dtget**_*p*	Tensor Get Element
vsip_**dtgetattrib**_*p*	Tensor Get View Attributes
vsip_**dtgetblock**_*p*	Tensor Get Block
vsip_**dtgetoffset**_*p*	Tensor Get Offset
vsip_**dtgetxlength**_*p*	Tensor Get X Length
vsip_**dtgetxstride**_*p*	Tensor Get X Stride
vsip_**dtgetylength**_*p*	Tensor Get Y Length
vsip_**dtgetystride**_*p*	Tensor Get Y Stride
vsip_**dtgetzlength**_*p*	Tensor Get Z Length
vsip_**dtgetzstride**_*p*	Tensor Get Z Stride
vsip_**timagview**_*p*	Create Imaginary Tensor View
vsip_**dtmatrixview**_*p*	Create Tensor Plane View
vsip_**dtput**_*p*	Tensor Put Element
vsip_**dtputattrib**_*p*	Tensor Put View Attributes
vsip_**dtputoffset**_*p*	Tensor Put Offset
vsip_**dtputxlength**_*p*	Tensor Put X Length
vsip_**dtputxstride**_*p*	Tensor Put X Stride

vsip_*d*tputylength_*p*	Tensor Put Y Length
vsip_*d*tputystride_*p*	Tensor Put Y Stride
vsip_*d*tputzlength_*p*	Tensor Put Z Length
vsip_*d*tputzstride_*p*	Tensor Put Z Stride
vsip_trealview_*p*	Create Real Tensor View
vsip_*d*tsubview_*p*	Create Sub-View Tensor View
vsip_*d*ttransview_p	Create Tensor Transposed View
vsip_*d*tvectview_*p*	Create Tensor Vector View

vsip_*d*talldestroy_*p* Destroy Tensor And Block

Destroy (free) a tensor, its associated block, and any *VSIPL data* array bound to the block.

Functionality

Destroys (frees) a `tensor` view object, the block object to which it is bound, and any *VSIPL data* array. If **T** is a tensor of type `vsip_dtview_p`, then

```
vsip_dtalldestroy_p(T);
```

is equivalent to

```
vsip_dblockdestroy_p(vsip_dtdestroy_p(T);
```

This is the complementary function to `vsip_dtcreate_p` and should only be used to destroy tensors that have only one view bound to the block object.

Prototypes

```
void vsip_talldestroy_f (
                     vsip_tview_f *T);
void vsip_ctalldestroy_f (
                     vsip_ctview_f *T);
void vsip_talldestroy_i (
                     vsip_tview_i *T);
void vsip_ctalldestroy_i (
                     vsip_ctview_i *T);
void vsip_talldestroy_bl (
                     vsip_tview_bl *T);
```

Arguments

`*T`	Tensor view object.

Return Value

None.

Restrictions

Errors

The arguments must conform to the following:

1. The tensor view object must be valid. An argument of *null* is not an error.
2. The specified tensor view must be the only view bound.
3. The tensor view must not be bound to a *derived* block (derived from a complex block).

Notes/References

An argument of *null* is not an error.

If the tensor view is bound to a *derived* block (derived from a complex block) the complex block must be destroyed to free the block and associated data.

Examples

See Also

`vsip_dblockcreate_p`, `vsip_blockbind_p`, `vsip_cblockbind_p`, `vsip_dblockadmit_p`, `vsip_blockrelease_p`, `vsip_cblockrelease_p`, `vsip_blockrebind_p`, `vsip_cblockrebind_p`, `vsip_blockfind_p`, `vsip_cblockfind_p`, `vsip_dblockdestroy_p`, `vsip_dtbind_p`, `vsip_dtcreate_p`, and `vsip_dtdestroy_p`

vsip_*d*tbind_*p* Create and Bind a Tensor View

Bind a tensor view to a block.

Functionality

Creates a tensor object or returns *null* if it fails. If the view create is successful, it: (1) binds the tensor view object to the block object; (2) sets the *offset* from the beginning of the data array to the beginning of the tensor, the stride between successive scalar elements along the *Z, Y,* and *X* axes, the number of scalar elements along the *Z, Y,* and *X* axes; (3) then returns a pointer to the created tensor view object.

Prototypes

```
vsip_tview_f *vsip_tbind_f(
                    const vsip_block_f *block,
                    vsip_offset offset,
                    vsip_stride z_stride,
                    vsip_length z_length,
                    vsip_stride y_stride,
                    vsip_length y_length,
                    vsip_stride x_stride,
                    vsip_length x_length);
vsip_tview_i *vsip_tbind_i(
                    const vsip_block_i *block,
                    vsip_offset offset,
                    vsip_stride z_stride,
                    vsip_length z_length,
                    vsip_stride y_stride,
                    vsip_length y_length,
                    vsip_stride x_stride,
                    vsip_length x_length);
vsip_ctview_f *vsip_ctbind_f(
                    const vsip_cblock_f *block,
                    vsip_offset offset,
                    vsip_stride z_stride,
                    vsip_length z_length,
                    vsip_stride y_stride,
                    vsip_length y_length,
                    vsip_stride x_stride,
                    vsip_length x_length);
vsip_ctview_i *vsip_ctbind_i(
                    const vsip_cblock_i *block,
                    vsip_offset offset,
                    vsip_stride z_stride,
                    vsip_length z_length,
                    vsip_stride y_stride,
                    vsip_length y_length,
                    vsip_stride x_stride,
                    vsip_length x_length);
```

```
vsip_tview_bl *vsip_tbind_bl (
                    const vsip_block_bl *block,
                    vsip_offset offset,
                    vsip_stride z_stride,
                    vsip_length z_length,
                    vsip_stride y_stride,
                    vsip_length y_length,
                    vsip_stride x_stride,
                    vsip_length x_length);
```

Arguments

`*block`	Pointer to a block object.
`offset`	Tensor view *offset* in elements relative to the base of block object.
`z_stride`	*Stride* between successive elements along the *Z* axis.
`z_length`	*Length* in elements along the *Z* axis.
`y_stride`	*Stride* between successive elements along the *Y* axis.
`y_length`	*Length* in elements along the *Y* axis.
`x_stride`	*Stride* between successive elements along the *X* axis.
`x_length`	*Length* in elements along the *X* axis.

Return Value

Returns a pointer to the created tensor view object, or *null* if the memory allocation for new object fails.

Restrictions

Errors

The arguments must conform to the following:

1. The block object must be valid.
2. The *offset* must be less than the length of the block's data array.
3. The *z length, z stride, y length, y stride, x length, x stride,* and *offset* arguments must not specify a tensor view that exceeds the bounds of the data array of the associated block.

Notes/References

It is important for the application to check the return value for a memory allocation failure.

Note to Implementors: In *development mode,* the function updates the bindings (reference count) recorded in the block object.

Examples

See Also

`vsip_`***d***`blockcreate_`***p***, `vsip_blockbind_`***p***, `vsip_cblockbind_`***p***, `vsip_`***d***`blockadmit_`***p***, `vsip_blockrelease_`***p***, `vsip_cblockrelease_`***p***, `vsip_blockrebind_`***p***, `vsip_cblockrebind_`***p***, `vsip_blockfind_`***p***, `vsip_cblockfind_`***p***, `vsip_`***d***`blockdestroy_`***p***, `vsip_`***d***`tcreate_`***p***, `vsip_`***d***`tdestroy_`***p***, and `vsip_`***d***`talldestroy_`***p***

vsip_*d*tcloneview_*p* Create Tensor View Clone

Create a clone of a tensor view.

Functionality

Creates a new tensor view object, copies all of the attributes of the source `tensor` view object to the new tensor view object, and then bind the new tensor view object to the block object of the source tensor view object. This function returns *null* on a memory allocation (creation) failure; otherwise, it returns a pointer to the new tensor view object.

Prototypes

```
vsip_tview_f *vsip_tcloneview_f (
                    const vsip_tview_f *T);
vsip_ctview_f *vsip_ctcloneview_f (
                    const vsip_ctview_f *T);
vsip_tview_i *vsip_tcloneview_i (
                    const vsip_tview_i *T);
vsip_ctview_i *vsip_ctcloneview_i (
                    const vsip_ctview_i *T);
vsip_tview_bl *vsip_tcloneview_bl (
                    const vsip_tview_bl *T);
```

Arguments

`*T` Tensor view object.

Return Value

Returns a pointer to the created tensor view object clone, or *null* if the memory allocation for new object fails.

Restrictions

Errors

The arguments must conform to the following:

1. The tensor view object must be valid.

Notes/References

Note to Implementors: In *development mode,* it should also increment the number of bindings recorded in the `vsip_dblock_p` object.

Examples

See Also

vsip_*d*tcreate_*p* Create Tensor

Creates a block object and tensor view (object) of the block.

Functionality

Creates a block object with an $P \cdot M \cdot N$ element *VSIPL data* array, it creates an P by M by N dense tensor view object and then binds the block object to it.

The function

```
vsip_tview_p *vsip_tcreate_p(P, M, N, VSIP_TRAILING, hint);
```

returns the same result as

```
vsip_tbind_p(vsip_blockcreate_p(P*M*N, hint),(vsip_offset)0,
    (vsip_stride)M*N, (vsip_length)P,      Z stride, length
    (vsip_stride)N, (vsip_length)M,        Y stride, length
    (vsip_stride)1, (vsip_length)N);       X stride, length
```

or

```
vsip_tview_p *vsip_tcreate_p(P, M, N, VSIP_LEADING, hint);
```

returns the same result as

```
vsip_tbind_p(vsip_blockcreate_p(P*M*N, hint),(vsip_offset)0,
    (vsip_stride)1, (vsip_length)P,        Z stride, length
    (vsip_stride)P, (vsip_length)M,        Y stride, length
    (vsip_stride)P*M, (vsip_length)N);     X stride, length
```

except that `vsip_tcreate_p` returns a *null* if `vsip_blockcreate_p(P*M*N, hint)` returns a *null*.

Prototypes

```
vsip_tview_f *vsip_tcreate_f(
                    vsip_length P,
                    vsip_length M,
                    vsip_length N,
                    vsip_tmajor major,
                    vsip_memory_hint hint);
vsip_tview_i *vsip_tcreate_i(
                    vsip_length P,
                    vsip_length M,
                    vsip_length N,
                    vsip_tmajor major,
                    vsip_memory_hint hint);
vsip_ctview_f *vsip_ctcreate_f(
                    vsip_length P,
                    vsip_length M,
                    vsip_length N,
                    vsip_tmajor major,
                    vsip_memory_hint hint);
vsip_ctview_i *vsip_ctcreate_i(
                    vsip_length P,
                    vsip_length M,
                    vsip_length N,
                    vsip_tmajor major,
                    vsip_memory_hint hint);
```

```
vsip_tview_bl *vsip_tcreate_bl(
                    vsip_length P,
                    vsip_length M,
                    vsip_length N,
                    vsip_tmajor major,
                    vsip_memory_hint hint);
```

Arguments

`P`	Number of elements (*Z length*) along *Z* axis of a tensor.
`M`	Number of elements (*Y length*) along *Y* axis of a tensor.
`N`	Number of elements (*X length*) along *X* axis of a tensor.
`major`	Trailing or leading index is the unit stride direction.

```
typedef enum {
    VSIP_TRAILING = 0,   Trailing dimension, C style
    VSIP_LEADING = 1     Leading Dimension, FORTRAN style
} vsip_tmajor;
```

`hint` Memory hint

```
typedef enum {
  VSIP_MEM_NONE = 0,                 No hint
  VSIP_MEM_RDONLY = 1,               Read Only
  VSIP_MEM_CONST = 2,                Constant
  VSIP_MEM_SHARED = 3,               Shared
  VSIP_MEM_SHARED_RDONLY = 4,        Shared, Read Only
  VSIP_MEM_SHARED_CONST = 5          Shared, Constant
} vsip_memory_hint;
```

Return Value

Returns a pointer to the created tensor view object, or *null* if it fails.

Restrictions

Errors

The arguments must conform to the following:

1. The lengths, *P*, *M* and *N*, must be greater than zero.
2. The *major* memory direction must be a valid member of the `vsip_tmajor` enumeration.
3. The memory *hint* must be a valid member of the `vsip_memory_hint` enumeration.

Notes/References

Note to Implementors: In *development mode,* it should also update the bindings (reference count) recorded in the block object.

Examples

See Also

`vsip_`*d*`blockcreate_`*p*, `vsip_blockbind_`*p*, `vsip_cblockbind_`*p*, `vsip_`*d*`blockadmit_`*p*, `vsip_blockrelease_`*p*, `vsip_cblockrelease_`*p*, `vsip_blockrebind_`*p*, `vsip_cblockrebind_`*p*, `vsip_blockfind_`*p*, `vsip_cblockfind_`*p*, `vsip_`*d*`blockdestroy_`*p*, `vsip_`*d*`tbind_`*p*, `vsip_`*d*`tdestroy_`*p*, and `vsip_`*d*`talldestroy_`*p*

vsip_*d*tdestroy_*p* Destroy Tensor View

Destroy (free) a tensor view object and return a pointer to the associated block object.

Functionality

Frees a tensor view object from the block object that it was bound to, destroys the tensor view object, and then returns a pointer to the block object. If the tensor view argument is *null*, it returns *null*.

Prototypes

```
vsip_block_f *vsip_tdestroy_f (
                    vsip_tview_f *T);
vsip_cblock_f *vsip_ctdestroy_f (
                    vsip_ctview_f *T);
vsip_block_i *vsip_tdestroy_i (
                    vsip_tview_i *T);
vsip_cblock_i *vsip_ctdestroy_i (
                    vsip_ctview_i *T);
vsip_block_bl *vsip_tdestroy_bl (
                    vsip_tview_bl *T);
```

Arguments

`*T` Tensor view object.

Return Value

Returns a pointer to the block object to which the tensor view was bound, or *null* if the calling argument was *null*.

Restrictions

Errors

The arguments must conform to the following:

1. The matrix view object must be valid. An argument of *null* is not an error.

Notes/References

An argument of *null* is not an error.

Note to Implementors: In *development mode,* the function updates bindings (reference count) recorded in the block object.

Examples

See Also

`vsip_dblockcreate_p`, `vsip_blockbind_p`, `vsip_cblockbind_p`, `vsip_dblockadmit_p`, `vsip_blockrelease_p`, `vsip_cblockrelease_p`, `vsip_blockrebind_p`, `vsip_cblockrebind_p`, `vsip_blockfind_p`, `vsip_cblockfind_p`, `vsip_dblockdestroy_p`, `vsip_dtbind_p`, `vsip_dtcreate_p`, and `vsip_dtalldestroy_p`

vsip_*d*tget_*p* Tensor Get Element

Get the value of a specified element of a tensor view object.

Functionality

Returns the value of the specified element of a tensor view object.

Returns $x_{h,i,j}$

Prototypes

```
vsip_scalar_f vsip_tget_f (
                    const vsip_tview_f *x,
                    vsip_index h,
                    vsip_index i,
                    vsip_index j);
vsip_cscalar_f vsip_ctget_f (
                    const vsip_ctview_f *x,
                    vsip_index h,
                    vsip_index i,
                    vsip_index j);
vsip_scalar_i vsip_tget_i (
                    const vsip_tview_i *x,
                    vsip_index h,
                    vsip_index i,
                    vsip_index j);
vsip_cscalar_i vsip_ctget_i (
                    const vsip_ctview_i *x,
                    vsip_index h,
                    vsip_index i,
                    vsip_index j);
vsip_scalar_bl vsip_tget_bl (
                    const vsip_tview_bl *x,
                    vsip_index h,
                    vsip_index i,
                    vsip_index j);
```

Arguments

`*x`	Tensor view object
`h`	Tensor index *h* of (*h, i, j*)
`i`	Tensor index *i* of (*h, i, j*)
`j`	Tensor index *j* (*h, i, j*)

Return Value

Returns the value of the specified element of a Tensor view object.

Restrictions

Errors

The arguments must conform to the following:

1. The tensor view object must be valid.
2. The index must be a valid index of the tensor view.

Notes/References

Examples

See Also

`vsip_`*`ds`*`put_`*`p`*

vsip_*d*tgetattrib_*p* Tensor Get View Attributes

Get the attributes of a tensor view object.

Functionality

Retrieves the attributes: *offset*, *Z length*, *Z stride*, *Y length*, *Y stride*, *X length*, *X stride*, and (pointer to) bound block object of a tensor view object.

Prototypes

```
void vsip_tgetattrib_f (
                      const vsip_tview_f *T,
                      vsip_tattr_f *attrib);
void vsip_tgetattrib_i (
                      const vsip_tview_i *T,
                      vsip_tattr_i *attrib);
void vsip_tgetattrib_bl (
                      const vsip_tview_bl *T,
                      vsip_tattr_bl *attrib);
void vsip_ctgetattrib_f (
                      const vsip_ctview_f *T,
                      vsip_ctattr_f *attrib);
void vsip_ctgetattrib_i (
                      const vsip_ctview_i *T,
                      vsip_ctattr_i *attrib);
typedef struct {
   vsip_offset offset;
   vsip_stride z_length;
   vsip_length z_stride;
   vsip_stride y_length;
   vsip_length y_stride;
   vsip_stride x_length;
   vsip_length x_stride;
   vsip_block_p *block;      Get only, ignored on Put
} vsip_tattr_p;
typedef struct {
   vsip_offset offset;
   vsip_stride z_length;
   vsip_length z_stride;
   vsip_stride y_length;
   vsip_length y_stride;
   vsip_stride x_length;
   vsip_length x_stride;
   vsip_cblock_p *block;     Get only, ignored on Put
} vsip_ctattr_p;
```

Arguments

`*T`	Tensor view object.
`*attrib`	Pointer to output tensor attribute structure.

Return Value

None.

Restrictions

Errors

The arguments must conform to the following:

1. The tensor view object must be valid.
2. The pointer to the tensor attribute structure must be valid – non-*null*.

Notes/References

The block attribute can be read (get), but cannot be set.

Examples

See Also

vsip_*d*tgetblock_*p* Tensor Get Block

Get the block attribute of a tensor view object.

Functionality

Returns a pointer to the *VSIPL block* object to which the tensor view object is bound.

Prototypes

```
vsip_block_f *vsip_tgetblock_f (
                    const vsip_tview_f *T);
vsip_cblock_f *vsip_ctgetblock_f (
                    const vsip_ctview_f *T);
vsip_block_i *vsip_tgetblock_i (
                    const vsip_tview_i *T);
vsip_cblock_i *vsip_ctgetblock_i (
                    const vsip_ctview_i *T);
vsip_block_bl *vsip_tgetblock_bl (
                    const vsip_tview_bl *T);
```

Arguments

`*T` Tensor view object.

Return Value

Returns a pointer to the block object to which the tensor view object is bound.

Restrictions

Errors

The arguments must conform to the following:

1. The tensor view object must be valid.

Notes/References

The functions `vsip_dvgetattrib_p` and `vsip_dvputattrib_p` are not symmetric since you can *get* the block object but you cannot *put* the block object.

Examples

See Also

vsip_*d*tgetoffset_*p* Tensor Get Offset

Get the *offset* attribute of a tensor view object.

Functionality

Returns the *offset* (in elements) to the first scalar element of a tensor view from the start of the block object to which it is bound.

Prototypes

```
vsip_offset vsip_tgetoffset_f (
                    const vsip_tview_f *T);
vsip_offset vsip_ctgetoffset_f (
                    const vsip_ctview_f *T);
vsip_offset vsip_tgetoffset_i (
                    const vsip_tview_i *T);
vsip_offset vsip_ctgetoffset_i (
                    const vsip_ctview_i *T);
vsip_offset vsip_tgetoffset_bl (
                    const vsip_tview_bl *T);
```

Arguments

`*T` Tensor view object.

Return Value

Returns the value of the *offset* attribute of the tensor view object.

Restrictions

Errors

The arguments must conform to the following:

1. The tensor view object must be valid.

Notes/References

Examples

See Also

vsip_*d*tgetxlength_*p* Tensor Get X Length

Get the *X length* attribute of a tensor view object.

Functionality

Returns the *length* of (number of elements along) the *X* axis of a tensor view.

Prototypes

```
vsip_length vsip_tgetxlength_f (
                      const vsip_tview_f *T);
vsip_length vsip_ctgetxlength_f (
                      const vsip_ctview_f *T);
vsip_length vsip_tgetxlength_i (
                      const vsip_tview_i *T);
vsip_length vsip_ctgetxlength_i (
                      const vsip_ctview_i *T);
vsip_length vsip_tgetxlength_bl (
                      const vsip_tview_bl *T);
```

Arguments

`*T` Tensor view object.

Return Value

Returns the value of the *X length* attribute of the tensor view object.

Restrictions

Errors

The arguments must conform to the following:

1. The tensor view object must be valid.

Notes/References

Examples

See Also

vsip_*d*tgetxstride_*p* Tensor Get X Stride

Get the *X stride* attribute of a tensor view object.

Functionality

Returns the *stride* (in elements of the bound block) between successive elements along the *X* axis of a tensor view.

Prototypes

```
vsip_stride vsip_tgetxstride_f (
                      const vsip_tview_f *T);
vsip_stride vsip_ctgetxstride_f (
                      const vsip_ctview_f *T);
vsip_stride vsip_tgetxstride_i (
                      const vsip_tview_i *T);
vsip_stride vsip_ctgetxstride_i (
                      const vsip_ctview_i *T);
vsip_stride vsip_tgetxstride_bl (
                      const vsip_tview_bl *T);
```

Arguments

`*T` Tensor view object.

Return Value

Returns the value of the *X stride* attribute of the tensor view object.

Errors

The arguments must conform to the following:

1. The tensor view object must be valid.

Notes/References

Examples

See Also

vsip_*d*tgetylength_*p* Tensor Get Y Length

Get the *Y length* attribute of a tensor view object.

Functionality

Returns the *length* of (number of elements along) the *Y* axis of a tensor view.

Prototypes

```
vsip_length vsip_tgetylength_f (
                      const vsip_tview_f *T);
vsip_length vsip_ctgetylength_f (
                      const vsip_ctview_f *T);
vsip_length vsip_tgetylength_i (
                      const vsip_tview_i *T);
vsip_length vsip_ctgetylength_i (
                      const vsip_ctview_i *T);
vsip_length vsip_tgetylength_bl (
                      const vsip_tview_bl *T);
```

Arguments

`*T` Tensor view object.

Return Value

Returns the value of the *Y length* attribute of the tensor view object.

Restrictions

Errors

The arguments must conform to the following:

1. The tensor view object must be valid.

Notes/References

Examples

See Also

vsip_*d*tgetystride_*p* Tensor Get Y Stride

Get the *Y stride* attribute of a tensor view object.

Functionality

Returns the *stride* (in elements of the bound block) between successive elements along the *Y* axis of a tensor view.

Prototypes

```
vsip_stride vsip_tgetystride_f (
                    const vsip_tview_f *T);
vsip_stride vsip_ctgetystride_f (
                    const vsip_ctview_f *T);
vsip_stride vsip_tgetystride_i (
                    const vsip_tview_i *T);
vsip_stride vsip_ctgetystride_i (
                    const vsip_ctview_i *T);
vsip_stride vsip_tgetystride_bl (
                    const vsip_tview_bl *T);
```

Arguments

`*T` Tensor view object.

Return Value

Returns the value of the *Y stride* attribute of the tensor view object.

Errors

The arguments must conform to the following:

1. The tensor view object must be valid.

Notes/References

Examples

See Also

vsip_*d*tgetzlength_*p* Tensor Get Z Length

Get the *Z length* attribute of a tensor view object.

Functionality

Returns the *length* of (number of elements along) the *Z* axis of a tensor view.

Prototypes

```
vsip_length vsip_tgetzlength_f (
                      const vsip_tview_f *T);
vsip_length vsip_ctgetzlength_f (
                      const vsip_ctview_f *T);
vsip_length vsip_tgetzlength_i (
                      const vsip_tview_i *T);
vsip_length vsip_ctgetzlength_i (
                      const vsip_ctview_i *T);
vsip_length vsip_tgetzlength_bl (
                      const vsip_tview_bl *T);
```

Arguments

`*T` Tensor view object.

Return Value

Returns the value of the *Z length* attribute of the tensor view object.

Restrictions

Errors

Notes/References

Examples

See Also

vsip_*d*tgetzstride_*p* Tensor Get Z Stride

Get the *Z stride* attribute of a tensor view object.

Functionality

Returns the *stride* (in elements of the bound block) between successive elements along the *Z* axis of a tensor view.

Prototypes

```
vsip_stride vsip_tgetzstride_f (
                    const vsip_tview_f *T);
vsip_stride vsip_ctgetzstride_f (
                    const vsip_ctview_f *T);
vsip_stride vsip_tgetzstride_i (
                    const vsip_tview_i *T);
vsip_stride vsip_ctgetzstride_i (
                    const vsip_ctview_i *T);
vsip_stride vsip_tgetzstride_bl (
                    const vsip_tview_bl *T);
```

Arguments

`*T` Tensor view object.

Return Value

Returns the value of the *Z stride* attribute of the tensor view object.

Errors

The arguments must conform to the following:

1. The tensor view object must be valid.

Notes/References

Examples

See Also

vsip_timagview_*p* Create Imaginary Tensor View

Create a *tensor* view object of the imaginary part of a complex tensor from a complex tensor view object.

Functionality

Creates a real tensor view object from the "imaginary part of a complex" tensor view object, or returns *null* if it fails.

On success, the function creates a *derived* block object, derived from the complex block object, which is bound to the *imaginary* data part of the original complex block and then binds a real tensor view object to the block. The new tensor encompasses the *imaginary* part of the source complex tensor.

Prototypes

```
vsip_tview_f *vsip_timagview_f (
                    const vsip_ctview_f *T);
vsip_tview_i *vsip_timagview_i (
                    const vsip_ctview_i *T);
```

Arguments

`*T` Tensor view object.

Return Value

Returns a pointer to the created "imaginary" part tensor view object, or *null* if the memory allocation for new object fails.

Restrictions

The *derived* block object, derived from the complex block object cannot be destroyed or released. The parent complex block object may be released (if it is bound to user data), and destroying the complex block is the only way to free the memory associated with the *derived* block object.

Errors

The arguments must conform to the following:

1. The complex tensor view object must be valid.

Notes/References

It is important for the application to check the return value for a memory allocation failure.

This function should not be confused with the function `vsip_simag_p()` which is a copy operator (copies the imaginary data).

There are no requirements on *offset* or *stride* of a real view on its *derived* block. By using `vsip_tgetattrib_p,` information about the layout of the view on the block may be obtained.

CAUTION. Using attribute information, and the block bound to the tensor, to bind new tensors outside the data space of the original tensor produced by `vsip_simagview_p` will produce non-portable code. Portable code may be produced by: (1) remaining inside the data space of the

tensor, (2) by not assuming a set relationship of strides and offsets, and (3) by using the get attributes functions to obtain necessary information within the application code to understand the layout for each implementation.

Note to Implementors:

- The resulting *derived* block must have a property which prevents it from being released or destroyed.
- In *development mode,* block binding count (reference count) recorded in the block object is incremented.

Examples

See Also

vsip_*d*tmatrixview_*p* Create Tensor Matrix View

Create a *matrix* view of a 2-D slice of the tensor view.

Functionality

Creates a matrix view object of a 2-D slice of the tensor view. The 2-D *slice*, or plane, is specified to be one of *Y-X, Z-X,* or *Z-Y* planes at a specified index along the remaining axis.

$M_{*,*} \leftarrow T_{i,*,*}$ *Y-X* Submatrix

$M_{*,*} \leftarrow T_{*,i,*}$ *Z-X* Submatrix

$M_{*,*} \leftarrow T_{*,*,i}$ *Z-Y* Submatrix

Where "*" denotes the set of all valid indices.

Prototypes

```
vsip_mview_f *vsip_tmatrixview_f(
                    const vsip_tview_f *T,
                    vsip_tmslice slice,
                    vsip_index i);
vsip_mview_i *vsip_tmatrixview_i(
                    const vsip_tview_i *T,
                    vsip_tmslice slice,
                    vsip_index i);
vsip_cmview_f *vsip_ctmatrixview_f (
                    const vsip_ctview_f *T,
                    vsip_tmslice slice,
                    vsip_index i);
vsip_cmview_i *vsip_ctmatrixview_i (
                    const vsip_ctview_i *T,
                    vsip_tmslice slice,
                    vsip_index i);
vsip_mview_bl *vsip_tmatrixview_bl (
                    const vsip_tview_bl *T,
                    vsip_tmslice slice,
                    vsip_index i);
typedef enum {
   VSIP_TMYX = 0,     Y-X Submatrix
   VSIP_TMZX = 1,     Z-X Submatrix
   VSIP_TMZY = 2      Z-Y Submatrix
}vsip_tmslice;
```

Arguments

`*T`	Tensor view object.
`slice`	slice of the tensor.
`i`	Index of the axis normal to the slice.

Return Value

Returns a pointer to the created matrix view object, or *null* if the memory allocation for new object fails.

Restrictions

Errors

The arguments must conform to the following:

1. The tensor view object must be valid.
2. The *slice* must be valid member of the `vsip_tmslice` enumeration.
3. The index, *i*, must be a valid index of the appropriate axis of the tensor view object.

Notes/References

It is important for the application to check the return value for a memory allocation failure.

Note to Implementors: In *development mode,* the block binding count (reference count) recorded in the block object is incremented.

Examples

See Also

vsip_*d*tput_*p* Tensor Put Element

Put (Set) the value of a specified element of a tensor view object

Functionality

Puts (sets) the value of the specified element of a tensor view object.

$y_{h,i,j} \leftarrow x$

Prototypes

```
void vsip_tput_f (
                      const vsip_tview_f *y,
                      vsip_index h,
                      vsip_index i,
                      vsip_index j,
                      vsip_scalar_f x);
void vsip_ctput_f (
                      const vsip_ctview_f *y,
                      vsip_index h,
                      vsip_index i,
                      vsip_index j,
                      vsip_cscalar_f x);
void vsip_tput_i (
                      const vsip_tview_i *y,
                      vsip_index h,
                      vsip_index i,
                      vsip_index j,
                      vsip_scalar_i x);
void vsip_ctput_i (
                      const vsip_ctview_i *y,
                      vsip_index h,
                      vsip_index i,
                      vsip_index j,
                      vsip_cscalar_i x);
void vsip_tput_bl (
                      const vsip_tview_bl *y,
                      vsip_index h,
                      vsip_index i,
                      vsip_index j,
                      vsip_scalar_bl x);
```

Arguments

`*y`	Vector/matrix/tensor view object of destination
`h`	Tensor index *h* of (*h, i, j*)
`i`	Tensor index *i* of (*h, i, j*)
`j`	Tensor index *j* of (*h, i, j*)
`x`	Scalar value to put in tensor.

Return Value

None.

Restrictions

Errors

The arguments must conform to the following:

1. The tensor view object must be valid.
2. The index must be a valid index of the tensor view.

Notes/References

Examples

See Also

`vsip_`*`ds`*`get_`*`p`*

vsip_*d*tputattrib_*p* Tensor Put View Attributes

Put (Set) the attributes of a tensor view object.

Functionality

Sets the attributes of *offset*, *Z length*, *Z stride*, *Y length*, *Y stride*, *X length*, and *X stride* of a tensor view object. As a programmer convenience, it returns a pointer to the tensor view object.

Prototypes

```
vsip_tview_f *vsip_tputattrib_f (
                     vsip_tview_f *T,
                     const vsip_tattr_f *attrib);
vsip_tview_i *vsip_tputattrib_i (
                     vsip_tview_i *T,
                     const vsip_tattr_i *attrib);
vsip_tview_bl *vsip_tputattrib_bl (
                     vsip_tview_bl *T,
                     const vsip_tattr_bl *attrib);
vsip_ctview_f *vsip_ctputattrib_f (
                     vsip_ctview_f *T,
                     const vsip_ctattr_f *attrib);
vsip_ctview_i *vsip_ctputattrib_i (
                     vsip_ctview_i *T,
                     const vsip_ctattr_i *attrib);
vsip_ctview_bl *vsip_ctputattrib_bl (
                     vsip_ctview_bl *T,
                     const vsip_tattr_bl *attrib);
typedef struct {
   vsip_offset offset;
   vsip_stride z_length;
   vsip_length z_stride;
   vsip_stride y_length;
   vsip_length y_stride;
   vsip_stride x_length;
   vsip_length x_stride;
   vsip_block_p *block;      Get only, ignored on Put
} vsip_tattr_p;
typedef struct {
   vsip_offset offset;
   vsip_stride z_length;
   vsip_length z_stride;
   vsip_stride y_length;
   vsip_length y_stride;
   vsip_stride x_length;
   vsip_length x_stride;
   vsip_cblock_p *block;     Get only, ignored on Put
} vsip_ctattr_p;
```

Arguments

`*T`	Tensor view object.
`*attrib`	Pointer to a tensor attribute structure

Return Value

Returns a pointer to the source tensor view object as a programming convenience.

Restrictions

Errors

The arguments must conform to the following:

1. The tensor view object must be valid.
2. The pointer to the tensor attribute structure must be valid – non-*null*.
3. The *z length, z stride, y length, y stride, x length, x stride,* and *offset* arguments must not specify a tensor view that exceeds the bounds of the data array of the associated block.

Notes/References

The functions `vsip_tgetattrib_p` and `vsip_tputattrib_p` are not symmetric since you can "get" the block object but you cannot "put" the block object.

Examples

See Also

vsip_*d*tputoffset_*p* Tensor Put Offset

Put (Set) the *offset* attribute of a tensor view object.

Functionality

Puts (sets) the *offset* (in elements) to the first scalar element of a tensor view, from the start of the block object's data array, to which it is bound.

Prototypes

```
vsip_tview_f *vsip_tputoffset_f (
                     vsip_tview_f *T,
                     vsip_offset offset);
vsip_ctview_f *vsip_ctputoffset_f (
                     vsip_ctview_f *T,
                     vsip_offset offset);
vsip_tview_i *vsip_tputoffset_i (
                     vsip_tview_i *T,
                     vsip_offset offset);
vsip_ctview_i *vsip_ctputoffset_i (
                     vsip_ctview_i *T,
                     vsip_offset offset);
vsip_tview_bl *vsip_tputoffset_bl (
                     vsip_tview_bl *T,
                     vsip_offset offset);
```

Arguments

`*T`	Tensor view object.
`offset`	Offset in elements relative to the start of the block object.

Return Value

Returns a pointer to the source tensor view object as a programming convenience.

Restrictions

Errors

The arguments must conform to the following:

1. The tensor view object must be valid.
2. The *offset* argument must not specify a tensor view that exceeds the bounds of the data array of the associated block.

Notes/References

Examples

See Also

vsip_*d*tputxlength_*p* Tensor Put X Length

Put (Set) the *X length* attribute of a tensor view object.

Functionality

Puts (sets) the *length* of (number of elements along) the *X* axis of a tensor view.

Prototypes

```
vsip_tview_p *vsip_tputxlength_p (
                     vsip_tview_p *T,
                     vsip_length length);
vsip_ctview_p *vsip_ctputxlength_p (
                     vsip_ctview_p *T,
                     vsip_length length);
```

Arguments

`*T`	Tensor view object.
`length`	*Length* of the *X* axis

Return Value

Returns a pointer to the source tensor view object as a programming convenience.

Restrictions

Errors

The arguments must conform to the following:

1. The tensor view object must be valid.
2. The *length* must be greater than zero.
3. The *length* argument must not specify a tensor view that exceeds the bounds of the data array of the associated block.

Notes/References

Examples

See Also

vsip_*d*tputxstride_*p* Tensor Put X Stride

Put (Set) the *X stride* attribute of a tensor view object.

Functionality

Puts (sets) the *stride* (in elements of the bound block) between successive elements along the *X* axis of a tensor view.

Prototypes

```
vsip_tview_f *vsip_tputxstride_f (
                    vsip_tview_f *T,
                    vsip_stride stride);
vsip_ctview_f *vsip_ctputxstride_f (
                    vsip_ctview_f *T,
                    vsip_stride stride);
vsip_tview_i *vsip_tputxstride_i (
                    vsip_tview_i *T,
                    vsip_stride stride);
vsip_ctview_i *vsip_ctputxstride_i (
                    vsip_ctview_i *T,
                    vsip_stride stride);
vsip_tview_bl *vsip_tputxstride_bl (
                    vsip_tview_bl *T,
                    vsip_stride stride);
```

Arguments

`*T`	Tensor view object.
`stride`	*Stride* in elements between successive elements along the *X* axis.

Return Value

Returns a pointer to the source tensor view object as a programming convenience.

Errors

The arguments must conform to the following:

1. The tensor view object must be valid.
2. The *stride* argument must not specify a tensor view that exceeds the bounds of the data array of the associated block.

Notes/References

An *X stride* of zero may be used to define a tensor view where each *Z-Y* slice is filled with a constant.

Examples

See Also

vsip_*d*tputylength_*p* Tensor Put Y Length

Put (Set) the *Y length* attribute of a tensor view object.

Functionality

Puts (sets) the *length* of (number of elements along) the *Y* axis of a tensor view.

Prototypes

```
vsip_tview_f *vsip_tputylength_f (
                     vsip_tview_f *T,
                     vsip_length length);
vsip_ctview_f *vsip_ctputylength_f (
                     vsip_ctview_f *T,
                     vsip_length length);
vsip_tview_i *vsip_tputylength_i (
                     vsip_tview_i *T,
                     vsip_length length);
vsip_ctview_i *vsip_ctputylength_i (
                     vsip_ctview_i *T,
                     vsip_length length);
vsip_tview_bl *vsip_tputylength_bl (
                     vsip_tview_bl *T,
                     vsip_length length);
```

Arguments

`*T`	Tensor view object.
`length`	*Length* of the *Y* axis.

Return Value

Returns a pointer to the source tensor view object as a programming convenience.

Restrictions

Errors

The arguments must conform to the following:

1. The tensor view object must be valid.
2. The *length* must be greater than zero.
3. The *length* argument must not specify a tensor view that exceeds the bounds of the data array of the associated block.

Notes/References

Examples

See Also

vsip_*d*tputystride_*p* Tensor Put Row Stride

Put (Set) the *Y stride* attribute of a tensor view object.

Functionality

Puts (sets) the *stride* (in elements of the bound block) between successive elements along the *Y* axis of a tensor view.

Prototypes

```
vsip_tview_f *vsip_tputystride_f (
                     vsip_tview_f *T,
                     vsip_stride stride);
vsip_ctview_f *vsip_ctputystride_f (
                     vsip_ctview_f *T,
                     vsip_stride stride);
vsip_tview_i *vsip_tputystride_i (
                     vsip_tview_i *T,
                     vsip_stride stride);
vsip_ctview_i *vsip_ctputystride_i (
                     vsip_ctview_i *T,
                     vsip_stride stride);
vsip_tview_bl *vsip_tputystride_bl (
                     vsip_tview_bl *T,
                     vsip_stride stride);
```

Arguments

`*T`	Tensor view object.
`stride`	*Stride* in elements between successive elements along the *Y* axis.

Return Value

Returns a pointer to the source tensor view object as a programming convenience.

Errors

The arguments must conform to the following:

1. The tensor view object must be valid.
2. The *stride* argument must not specify a tensor view that exceeds the bounds of the data array of the associated block.

Notes/References

A *Y stride* of zero may be used to define a tensor view where each *Z-X* slice is filled with a constant.

Examples

See Also

vsip_*d*tputzlength_*p* Tensor Put Z Length

Put (Set) the *Z length* attribute of a tensor view object.

Functionality

Puts (sets) the *length* of (number of elements along) the *Z* axis of a tensor view.

Prototypes

```
vsip_tview_f *vsip_tputylength_f (
                     vsip_tview_f *T,
                     vsip_length length);
vsip_ctview_f *vsip_ctputylength_f (
                     vsip_ctview_f *T,
                     vsip_length length);
vsip_tview_i *vsip_tputylength_i (
                     vsip_tview_i *T,
                     vsip_length length);
vsip_ctview_i *vsip_ctputylength_i (
                     vsip_ctview_i *T,
                     vsip_length length);
vsip_tview_bl *vsip_tputylength_bl (
                     vsip_tview_bl *T,
                     vsip_length length);
```

Arguments

`*T`	Tensor view object.
`length`	*Length* of the *Z* axis.

Return Value

Returns a pointer to the source tensor view object as a programming convenience.

Restrictions

Errors

The arguments must conform to the following:

1. The tensor view object must be valid.
2. The *length* must be greater than zero.
3. The *length* argument must not specify a tensor view that exceeds the bounds of the data array of the associated block.

Notes/References

Examples

See Also

vsip_*d*tputzstride_*p* Tensor Put Z Stride

Put (Set) the *Z stride* attribute of a tensor view object.

Functionality

Puts (sets) the *stride* (in elements of the bound block) between successive elements along the *Z* axis of a tensor view.

Prototypes

```
vsip_tview_f *vsip_tputzstride_f (
                    vsip_tview_f *T,
                    vsip_stride stride);
vsip_ctview_f *vsip_ctputzstride_f (
                    vsip_ctview_f *T,
                    vsip_stride stride);
vsip_tview_i *vsip_tputzstride_i (
                    vsip_tview_i *T,
                    vsip_stride stride);
vsip_ctview_i *vsip_ctputzstride_i (
                    vsip_ctview_i *T,
                    vsip_stride stride);
vsip_tview_bl *vsip_tputzstride_bl (
                    vsip_tview_bl *T,
                    vsip_stride stride);
```

Arguments

`*T`	Tensor view object.
`stride`	*Stride* in elements between successive elements along the *Z* axis.

Return Value

Returns a pointer to the source tensor view object as a programming convenience.

Errors

The arguments must conform to the following:

1. The tensor view object must be valid.
2. The *stride* argument must not specify a tensor view that exceeds the bounds of the data array of the associated block.

Notes/References

A *Z stride* of zero may be used to define a tensor view where each *Y-X* slice is filled with a constant.

Examples

See Also

vsip_trealview_*p* Create Real Tensor View

Create a tensor view object of the real part of a complex tensor from a complex tensor view object.

Functionality

Creates a real tensor view object from the "real part of a complex" tensor view object, or returns *null* if it fails.

On success, the function creates a *derived* block object (derived from the complex block object). The derived block object is bound to the real data part of the original complex block and then binds a real `tensor` view object to the block. The new `tensor` encompasses the real part of the input complex `tensor`.

Prototypes

```
vsip_tview_f *vsip_trealview_f (
                    const vsip_ctview_f *T);
vsip_tview_i *vsip_trealview_i (
                    const vsip_ctview_i *T);
```

Arguments

`*T`	Tensor view object.

Return Value

Returns a pointer to the created "real" part tensor view object, or *null* if the memory allocation for new object fails.

Restrictions

The *derived* block object, derived from the complex block object cannot be destroyed or released. The parent complex block object may be released (if it is bound to user data), and destroying the complex block is the only way to free the memory associated with the *derived* block object.

Errors

The arguments must conform to the following:

1. The complex tensor view object must be valid.

Notes/References

It is important for the application to check the return value for a memory allocation failure.

This function should not be confused with the function `vsip_sreal_p()` which is a copy operator (copies the real data).

There are no requirements on *offset* or *stride* of a real view on its *derived* block. By using `vsip_tgetattrib_p,` information about the layout of the view on the block may be obtained.

CAUTION. Using attribute information, and the block bound to the tensor, to bind new tensors outside the data space of the original tensor produced by `vsip_srealview_p` will produce non-portable code. Portable code may be produced by: (1) remaining inside the data space of the

tensor, (2) by not assuming a set relationship of strides and offsets, and (3) by using the get attributes functions to obtain necessary information within the application code to understand the layout for each implementation.

Note to Implementors:

- The resulting *derived* block must have a property which prevents it from being released or destroyed.
- In *development mode,* block binding count (reference count) recorded in the block object is incremented.

Examples

See Also

vsip_*d*tsubview_*p* Create Subview Tensor View

Create a tensor view object that is a subview of tensor view object.

Functionality

Creates a tensor view object from a subview of a tensor view, or returns *null* if it fails. The subview is a *P* by *M* by *N* tensor view whose (0,0,0) element corresponds with the (*Z index*, *Y index*, *X index*) element of the source tensor view.

(The subview is relative to the source view, *Z stride, Y stride,* and *X strides* are inherited from the source view).

Prototypes

```
vsip_tview_f *vsip_tsubview_f (
                      const vsip_tview_f *T,
                      vsip_index z_index,
                      vsip_index y_index,
                      vsip_index x_index,
                      vsip_length P,
                      vsip_length M,
                      vsip_length N);
vsip_ctview_f *vsip_ctsubview_f (
                      const vsip_ctview_f *T,
                      vsip_index z_index,
                      vsip_index y_index,
                      vsip_index x_index,
                      vsip_length P,
                      vsip_length M,
                      vsip_length N);
vsip_tview_i *vsip_tsubview_i (
                      const vsip_tview_i *T,
                      vsip_index z_index,
                      vsip_index y_index,
                      vsip_index x_index,
                      vsip_length P,
                      vsip_length M,
                      vsip_length N);
vsip_ctview_i *vsip_ctsubview_i (
                      const vsip_ctview_i *T,
                      vsip_index z_index,
                      vsip_index y_index,
                      vsip_index x_index,
                      vsip_length P,
                      vsip_length M,
                      vsip_length N);
vsip_tview_bl *vsip_tsubview_bl (
                      const vsip_tview_bl *T,
                      vsip_index z_index,
                      vsip_index y_index,
                      vsip_index x_index,
                      vsip_length P,
                      vsip_length M,
                      vsip_length N);
```

Arguments

`*T`	Tensor view object.
`z_index`	The index (*Z index*, *Y index*, *X index*) of the source matrix view object is mapped to the index (0,0,0) of the subview tensor object.
`y_index`	The index (*Z index*, *Y index*, *X index*) of the source matrix view object is mapped to the index (0,0,0) of the subview tensor object.
`x_index`	The index (*Z index*, *Y index*, *X index*) of the source matrix view object is mapped to the index (0,0,0) of the subview tensor object.
`P`	Number of elements (*Z length*) along *Z* axis of tensor subview.
`M`	Number of elements (*Y length*) along *Y* axis of tensor subview.
`N`	Number of elements (*X length*) along *X* axis of tensor subview.

Return Value

Returns a pointer to the created subview tensor view object, or *null* if the memory allocation for new object fails.

Restrictions

Errors

The arguments must conform to the following:

1. The matrix view object must be valid.
2. The matrix index (*Z index*, *Y index*, *X index*) must be a valid index of the tensor view.
3. The subview must not extend beyond the bounds of the source tensor view.

Notes/References

It is important for the application to check the return value for a memory allocation failure.

Note to Implementors: In *development mode,* it should also increment the number of bindings (reference count) recorded in the block object.

Examples

See Also

vsip_*d*ttransview_*p* Create Tensor Transposed View

Create a transposed tensor view.

Functionality

Creates a tensor view object that provides a transposed view of a specified a tensor view, or returns *null* if it fails. On success, it binds the new tensor view object to the same block object as the source tensor view object and sets its attributes to view the transpose of the source tensor object.

$T_{h,i,j} \leftarrow T_{h,i,j}$	No-transpose	`VSIP_TTRANS_NOP`
$T_{h,i,j} \leftarrow T_{h,j,i}$	Y – X transpose	`VSIP_TTRANS_YX`
$T_{h,i,j} \leftarrow T_{i,h,j}$	Z – Y transpose	`VSIP_TTRANS_ZY`
$T_{h,i,j} \leftarrow T_{j,i,h}$	Z – X transpose	`VSIP_TTRANS_ZX`
$T_{h,i,j} \leftarrow T_{i,j,h}$	Y – X & Z – Y transpose	`VSIP_TTRANS_YXZY`
$T_{h,i,j} \leftarrow T_{j,h,i}$	Y – X & Z – X transpose	`VSIP_TTRANS_YXZX`

Where "*" denotes the set of all valid indices.

Prototypes

```
vsip_tview_f *vsip_ttransview_f(
                    const vsip_tview_f *T,
                    vsip_ttrans trans);
vsip_tview_i *vsip_ttransview_i(
                    const vsip_tview_i *T,
                    vsip_ttrans trans);
vsip_ctview_f *vsip_cttransview_f (
                    const vsip_ctview_f *T,
                    vsip_ttrans trans);
vsip_ctview_i *vsip_cttransview_i (
                    const vsip_ctview_i *T,
                    vsip_ttrans trans);
vsip_tview_bl *vsip_ttransview_bl (
                    const vsip_tview_bl *T,
                    vsip_ttrans trans);
typedef enum {
  VSIP_TTRANS_NOP = 0,          No transpose
  VSIP_TTRANS_YX = 1,           Y – X transpose
  VSIP_TTRANS_ZY = 2,           Z – Y transpose
  VSIP_TTRANS_ZX = 3,           Z – X transpose
  VSIP_TTRANS_YXZY = 4,         Y – X & Z – Y transpose
  VSIP_TTRANS_YXZX = 5          Y – X & Z – X transpose
}vsip_ttrans;
```

Arguments

`*T`	Tensor view object.
`trans`	Specifies *transpose* type.

Return Value

Returns a pointer to the created tensor view object, or *null* if the create fails.

Restrictions

Errors

The arguments must conform to the following:

1. The tensor view object must be valid.
2. The *transpose* type must be a valid member of the `vsip_ttrans` enumeration.

Notes/References

Examples

See Also

vsip_*d*tvectview_*p* Create Tensor Vector View

Create a vector view of a 1-D slice of the tensor view.

Functionality

Creates a vector view of a 1-D slice of the tensor view, or returns *null* if it fails. The 1-D slice is specified to be one of *X, Y,* or *Z*, at a specified index along the other axes.

$v_* \leftarrow T_{i,j,*}$ *X* Subvector

$v_* \leftarrow T_{i,*,j}$ *Y* Subvector

$v_* \leftarrow T_{*,i,j}$ *Z* Subvector

Where "*" denotes the set of all valid indices.

Prototypes

```
vsip_vview_f *vsip_tvectview_f(
                    const vsip_tview_f *T,
                    vsip_tvslice slice,
                    vsip_index i,
                    vsip_index j);

vsip_vview_i *vsip_tvectview_i(
                    const vsip_tview_i *T,
                    vsip_tvslice slice,
                    vsip_index i,
                    vsip_index j);

vsip_cvview_f *vsip_ctvectview_f (
                    const vsip_ctview_f *T,
                    vsip_tvslice slice,
                    vsip_index i,
                    vsip_index j);

vsip_cvview_i *vsip_ctvectview_i (
                    const vsip_ctview_i *T,
                    vsip_tvslice slice,
                    vsip_index i,
                    vsip_index j);

vsip_vview_bl *vsip_tvectview_bl(
                    const vsip_tview_bl *T,
                    vsip_tvslice slice,
                    vsip_index i,
                    vsip_index j);
typedef enum {
  VSIP_TVX = 0,                    X Subvector
  VSIP_TVY = 1,                    Y Subvector
  VSIP_TVZ = 2                     Z Subvector
}vsip_tvslice;
```

Arguments

`*T`	Tensor view object.
`slice`	1-D slice is along the *slice* axis.
`i`	First fixed tensor index.
`j`	Second fixed tensor index.

Return Value

Returns a pointer to the created vector view object, or *null* if the memory allocation for new object fails.

Restrictions

Errors

The arguments must conform to the following:

1. The tensor view object must be valid.
2. The *slice* must be valid member of the `vsip_tvslice` enumeration.
3. The indices, *i* and *j*, must be valid indices of their appropriate axes of the tensor view object.

Notes/References

It is important for the application to check the return value for a memory allocation failure.

Note to Implementors: In *development mode,* the block binding count (reference count) recorded in the block object is incremented.

Examples

See Also

SCALAR FUNCTIONS

Introduction To Scalar Functions

Domain And Range Errors

VSIPL does not specify the behavior for domain and range errors for scalar functions. The result is implementation dependent. As such, domain and range error reporting via the ANSI C `errno` mechanism is implementation dependent,

Notes To Implementors

Where the prototype specification of a functions states:
`vsip_`function_name`_`*`f`*
A compliant implementation is required to implement at least one floating point data type (`float`, `double`, and `long double`). It is up to the implementation which floating point types it supports.

Similarly,
`vsip_`function_name`_`*`i`*
requires at least one integer data type (`short int`, `int`, `long int`, `unsigned short int`, `unsigned int`, `unsigned long int`. [`long long int`, `unsigned long long int`][2]).

The following tables indicate how the VSIPL scalar functions correspond with the current ANSI C standard, as well as the proposed ANSI C9x draft. While draft standards are fickle things, subject to many changes, most of the current C compilers already support all of the C9x real scalar functions that are cited in the table.

The error handling requirements for VSIPL scalar functions are non specified. Thus it would be acceptable to implement scalar functions with *defines*.

```
#define vsip_acos_d acos
#ifdef HAVE_C9X_MATH_FLOAT
#define vsip_acos_f acosf
#else
#define vsip_acos_f (float)acos
#endif
```

Better still, if your system has an ANSI C compiler that supports an *inline* extension:

```
inline double vsip_acos_d(double x){return acos(x);};
inline double vsip_acos_f(float x);
{
   #ifdef HAVE_MATH_FLOAT
      return acosf(x);
   #else
      return (float)acos((double)x);
   #endif
}
```

This eliminates confusion with debugging caused by the function name substitution.

[2] `long long int` is not an ANSI type, but is acceptable as a VSIPL precision suffix of `_ie64`, `_if64`, `_il64` (or `_ie128`, `_if128`, `_il128` or whatever is appropriate).

Real Scalar Functions

VSIPL	ANSI C double	ANSI C9x float	ANSI C9x double	ANSI C9x long double
vsip_**acos**_*p*	acos	acosf	acos	acosl
vsip_**asin**_*p*	asin	asinf	asin	asinl
vsip_**atan**_*p*	atan	atanf	atan	atanl
vsip_**atan2**_*p*	atan2	atan2f	atan2	atan2l
vsip_**cos**_*p*	cos	cosf	cos	cosl
vsip_**sin**_*p*	sin	sinf	sin	sinl
vsip_**tan**_*p*	tan	tanf	tan	tanl
vsip_**cosh**_*p*	cosh	coshf	cosh	coshl
vsip_**sinh**_*p*	sinh	sinhf	sinh	sinhl
vsip_**tanh**_*p*	tanh	tanhf	tanh	tanhl
vsip_**exp**_*p*	exp	expf	exp	expl
vsip_**exp10**_*p*				
vsip_**log**_*p*	log	logf	log	logl
vsip_**log10**_*p*	log10	log10f	log10	log10l
vsip_**pow**_*p*	pow	powf	pow	powl
vsip_**rsqrt**_*p*				
vsip_**sqrt**_*p*	sqrt	sqrtf	sqrt	sqrtl
vsip_**hypot**_*p*		hypotf	hypot	hypotl
vsip_**mag**_*p*	fabs	fabsf	fabs	fabsl
vsip_**ceil**_*p*	ceil	ceilf	ceil	ceill
vsip_**floor**_*p*	floor	floorf	floor	floorl
vsip_**fmod**_*p*	modf	modff	modf	modfl
vsip_**max**_*p*		fmaxf	fmax	fmaxl
vsip_**min**_*p*		fminf	fmin	fminl

ANSI C9x plans to extend C with a complex type. While it is to early to be certain, we expect `vsip_cscalar_d` (`vsip_cscalar_f`) to be compatible with `complex double` (`complex float`).

Complex Scalar Functions

VSIPL	ANSI C9x `float`	ANSI C9x `double`	ANSI C9x `long double`
vsip_**arg**_*p*	cargf()	carg()	cargl()
vsip_**cadd**_*p*	()+()	()+()	()+()
vsip_**conj**_*p*	conjf()	conj()	conjl()
vsip_**cdiv**_*p*	()/()	()/()	()/()
vsip_**cexp**_*p*	cexpf()	cexp()	cexpl()
vsip_**cjmul**_*p*	()*conjf()	()*conj()	()*conjl()
vsip_**clog**_*p*	clogf()	clog()	clogl()
vsip_**cmag**_*p*	cabsf()	cabs()	cabsl()
vsip_**cmagsq**_*p*			
vsip_**cmplx**_*p*			
vsip_**cmul**_*p*	()*()	()*()	()*()
vsip_**cneg**_*p*	-()	-()	-()
vsip_**crecip**_*p*	1.0/()	1.0/()	1.0/()
vsip_**csub**_*p*	()-()	()-()	()-()
vsip_**csqrt**_*p*	csqrtf()	csqrt()	csqrtl()
vsip_**imag**_*p*	cimagf()	cimag()	cimagl()
vsip_**polar**_*p*			
vsip_**real**_*p*	crealf()	creal()	creall()
vsip_**rect**_*p*			

Real Scalar Functions

The VSIPL real scalar functions are provided for two purposes.

1. VSIPL requires extensive error checking in development mode. However, the requirements for error detection and handling for scalar functions has not been defined at this time. The VSIPL real scalar functions provide a mechanism to implement such error detection and handling requirements, if and when they are specified.
2. ANSI C only defines the functions in the standard C Math library for `double`. VSIPL defines the real scalar functions for all the floating types the implementation supports (`_f` reserves the name space for `float`, `double`, and `long double`).

`vsip_acos_p`	Scalar Arccosine
`vsip_asin_p`	Scalar Arcsine
`vsip_atan_p`	Scalar Arctangent
`vsip_atan2_p`	Scalar Arctangent of Two Arguments
`vsip_ceil_p`	Ceiling
`vsip_cos_p`	Scalar Cosine
`vsip_cosh_p`	Scalar Hyperbolic Cosine
`vsip_exp_p`	Scalar Exponential
`vsip_exp10_p`	Scalar Exponential Base 10
`vsip_floor_p`	Floor
`vsip_fmod_p`	Modulo
`vsip_hypot_p`	Scalar Hypotenuse
`vsip_log_p`	Scalar Log
`vsip_log10_p`	Scalar Log Base Ten
`vsip_mag_p`	Scalar Magnitude (Abs)
`vsip_max_p`	Scalar Maximum
`vsip_min_p`	Scalar Minimum
`vsip_pow_p`	Scalar Power
`vsip_rsqrt_p`	Scalar Reciprocal Square Root
`vsip_sin_p`	Scalar Sine
`vsip_sinh_p`	Scalar Hyperbolic Sine
`vsip_sqrt_p`	Scalar Square Root
`vsip_tan_p`	Scalar Tangent
`vsip_tanh_p`	Scalar Hyperbolic Tangent

vsip_acos_*p* Scalar Arccosine

Computes the principal radian value [0, π] of the arc cosine of a scalar.

Functionality

$r \leftarrow \cos^{-1} \theta$

Prototypes

```
vsip_scalar_f vsip_acos_f(
                    vsip_scalar_f theta);
```

Arguments

`theta`	Argument

Return Value

The arc-cosine.

Restrictions

Input outside the range [-1, 1] is a domain error. Results of inputs outside this range are implementation dependent.

Errors

Notes/References

Examples

See Also

`vsip_asin_p`, `vsip_atan_p`, `vsip_atan2_p`, `vsip_cos_p`, `vsip_sin_p`, and `vsip_tan_p`

vsip_asin_*p* Scalar Arcsine

Computes the principal radian value [-π/2, π/2] of the arc sine of a scalar.

Functionality

$r \leftarrow \sin^{-1}\theta$

Prototypes

```
vsip_scalar_f vsip_asin_f(
                    vsip_scalar_f theta);
```

Arguments

`theta`	Argument

Return Value

The arc-sine.

Restrictions

Input outside the range [-1, 1] is a domain error. Results of inputs outside this range are implementation dependent.

Errors

Notes/References

Examples

See Also

`vsip_acos_p`, `vsip_atan_p`, `vsip_atan2_p`, `vsip_cos_p`, `vsip_sin_p`, and `vsip_tan_p`

vsip_atan_*p* Scalar Arctangent

Computes the principal radian value [-π/2, π/2] of the Arctangent of a scalar.

Functionality

$r \leftarrow \tan^{-1} \theta$

Prototypes

```
vsip_scalar_f vsip_atan_f(
                    vsip_scalar_f a);
```

Arguments

a	Argument

Return Value

The arc-tangent.

Restrictions

Errors

Notes/References

Examples

See Also

vsip_acos_*p*, vsip_asin_*p*, vsip_atan2_*p*, vsip_cos_*p*, vsip_sin_*p*, and vsip_tan_*p*

vsip_atan2_*p* Scalar Arctangent of Two Arguments

Computes the four quadrant radian value [-π, π] of the arc tangent of the ratio of two scalars.

Functionality

$r \leftarrow \tan^{-1}(a/b)$

The rules for calculating `vsip_atan2_p` are the same as for the ANSI C math function `atan2`. The following table may be used to calculate `atan2`, although other methods may also be used.

If	Then
$a > 0;\quad b \neq 0$	$\tan^{-1}\frac{a}{b} \equiv \cos^{-1}\left(b \Big/ \sqrt{a^2+b^2}\right)$
$a < 0;\quad b < 0$	$\tan^{-1}\frac{a}{b} \equiv \cos^{-1}\left(-b \Big/ \sqrt{a^2+b^2}\right) - \pi$
$a < 0;\quad b > 0$	$\tan^{-1}\frac{a}{b} \equiv -\cos^{-1}\left(b \Big/ \sqrt{a^2+b^2}\right)$
$a > 0;\quad b = 0$	$\tan^{-1}\frac{a}{0} \equiv \pi/2$
$a < 0;\quad b = 0$	$\tan^{-1}\frac{a}{0} \equiv -\pi/2$
$a = 0;\quad b > 0$	$\tan^{-1}\frac{0}{b} \equiv 0$
$a = 0;\quad b < 0$	$\tan^{-1}\frac{0}{b} \equiv \pi$
$a = 0;\quad b = 0$	$\tan^{-1}\frac{0}{0} \equiv$ Undefined or NaN

Prototypes

```
vsip_scalar_f vsip_atan2_f(
                    vsip_scalar_f a,
                    vsip_scalar_f b);
```

Arguments

`a`	Numerator
`b`	Denominator

Return Value

The arc tangent.

Restrictions

The domain of `vsip_atan2_p(x, y)` is not valid for both *x* and *y* zero, and the result is implementation dependent.

Errors

Notes/References

Examples

See Also

`vsip_acos_p`, `vsip_asin_p`, `vsip_atan_p`, `vsip_atan2_p`, `vsip_cos_p`, `vsip_sin_p`, `vsip_tan_p`, `vsip_hypot_p`, and `vsip_arg_p`

The function `vsip_hypot_p` is related to `vsip_atan2_p` in the same way as rectangular to polar conversion is related through the magnitude and the argument. Another related function is `vsip_arg_p`.

vsip_ceil_*p* Ceiling

Computes the ceiling of a scalar.

Functionality

$r \leftarrow \lceil x \rceil$

Returns the smallest integral value greater than or equal to the argument.

Prototypes

```
vsip_scalar_f vsip_ceil_f(
                    vsip_scalar_f x);
```

Arguments

`x`	Argument

Return Value

Returns the ceiling.

Restrictions

Errors

Notes/References

Examples

See Also

`vsip_floor_p`, and `vsip_fmod_p`

vsip_cos_*p* Scalar Cosine

Computes the cosine of a scalar angle in radians.

Functionality

$r \leftarrow \cos \theta$

Prototypes

```
vsip_scalar_f vsip_cos_f(
                    vsip_scalar_f theta);
```

Arguments

`theta`	Angle in radians

Return Value

Returns the cosine.

Restrictions

Errors

Notes/References

Input arguments are expressed in radians.

Examples

See Also

`vsip_acos_p`, `vsip_asin_p`, `vsip_atan_p`, `vsip_atan2_p`, `vsip_sin_p`, and `vsip_tan_p`

vsip_cosh_*p* Scalar Hyperbolic Cosine

Computes the hyperbolic cosine of a scalar.

Functionality

$r \leftarrow \cosh x$

Prototypes

```
vsip_scalar_f vsip_cosh_f(
                    vsip_scalar_f x);
```

Arguments

`x`	Argument

Return Value

Returns the cosh.

Restrictions

The maximum domain without overflow is implementation dependent.

Errors

Notes/References

Examples

See Also

`vsip_sinh_p`, and `vsip_tanh_p`

vsip_exp_*p* Scalar Exponential

Computes the exponential of a scalar.

Functionality

$r \leftarrow e^x$

Prototypes

```
vsip_scalar_f vsip_exp_f(
                    vsip_scalar_f x);
```

Arguments

`x`	Argument

Return Value

Returns the exponent.

Restrictions

Overflow will occur if the argument is greater than the $\log_e$ of the maximum representable number. If this occurs, the result is implementation dependent.

Errors

Notes/References

Examples

See Also

`vsip_exp10_p`, `vsip_log_p`, `vsip_log10_p`, `vsip_pow_p`, `vsip_cexp_p`, and `vsip_clog_p`

vsip_exp10_*p* Scalar Exponential Base 10

Computes the base 10 exponential of a scalar.

Functionality

$r \leftarrow 10^x$

Prototypes

```
vsip_scalar_f vsip_exp10_f(
                    vsip_scalar_f x);
```

Arguments

`x`	Argument

Return Value

Returns the exponent.

Restrictions

Overflow will occur if the argument is greater than the $\log_{10}$ of the maximum representable number. If this occurs, the result is implementation dependent.

Underflow will occur if the argument is less than the negative of the $\log_{10}$ of the maximum representable number. If this occurs, the result is implementation dependent.

Errors

Notes/References

Examples

See Also

`vsip_exp_p`, `vsip_log_p`, `vsip_log10_p`, `vsip_pow_p`, `vsip_cexp_p`, and `vsip_clog_p`

vsip_floor_*p* Floor

Computes the floor of a scalar.

Functionality

$r \leftarrow \lfloor x \rfloor$

Returns the largest integral value less than or equal to the argument.

Prototypes

```
vsip_scalar_f vsip_floor_f(
                    vsip_scalar_f x);
```

Arguments

`x`	Argument

Return Value

Returns the floor.

Restrictions

Errors

Notes/References

Examples

See Also

`vsip_ceil_p`, and `vsip_fmod_p`

vsip_fmod_*p* Modulo

Computes the remainder of the quotient (modulo) of two scalars.

Functionality

Returns the floating point remainder of x/y.

$r \leftarrow x - n \cdot y$

For some integer n such that, if y is non-zero, the result has the same sign as x and magnitude less than the magnitude of y.

Prototypes

```
vsip_scalar_f vsip_fmod_f(
                    vsip_scalar_f x,
                    vsip_scalar_f y);
```

Arguments

`x`	Argument
`y`	Argument

Return Value

Returns the remainder.

Restrictions

Errors

Notes/References

If y is zero, whether a domain error occurs or the function returns zero is implementation dependent.

Examples

See Also

`vsip_floor_p`, and `vsip_ceil_p`

vsip_hypot_*p* Scalar Hypotenuse

Computes the square root of the sum of the squares (hypotenuse) of the two scalars.

Functionality

$r \leftarrow \sqrt{a^2 + b^2}$

Prototypes

```
vsip_scalar_f vsip_hypot_f(
                    const vsip_scalar_f a,
                    const vsip_scalar_f b);
```

Arguments

a	Argument
b	Argument

Return Value

Return the hypotenuse.

Restrictions

Errors

Notes/References

Intermediate overflows will not occur.

Examples

See Also

`vsip_atan2_p`, `vsip_sqrt_p`, `vsip_rsqrt_p`, `vsip_hypot_p`, and `vsip_csqrt_p`

This function is related to `atan2` (in the Elementary Math section), in the same way as rectangular to polar conversion is related through the magnitude and the argument.

vsip_log_*p* Scalar Log

Computes the natural logarithm of a scalar.

Functionality

$r \leftarrow \log_e x$

Prototypes

```
vsip_scalar_f vsip_log_f(
                    vsip_scalar_f x);
```

Arguments

`x`	Argument

Return Value

Returns the natural logarithm.

Restrictions

Arguments less than or equal to zero are not in the domain of log and the result is implementation dependent.

Errors

Notes/References

Examples

See Also

`vsip_exp_p`, `vsip_exp10_p`, `vsip_log10_p`, `vsip_pow_p`, `vsip_cexp_p`, and `vsip_clog_p`

vsip_log10_*p* Scalar Log Base Ten

Computes the base 10 logarithm of a scalar.

Functionality

$r \leftarrow \log_{10} x$

Prototypes

```
vsip_scalar_f vsip_log10_f(
                    vsip_scalar_f x);
```

Arguments

`x`	Argument

Return Value

Returns the base 10 logarithm.

Restrictions

Arguments less than or equal to zero are not in the domain of log and the result is implementation dependent.

Errors

Notes/References

Examples

See Also

`vsip_exp_p`, `vsip_exp10_p`, `vsip_log_p`, `vsip_pow_p`, `vsip_cexp_p`, and `vsip_clog_p`

vsip_mag_*p* Scalar Magnitude (Abs)

Computes the magnitude (absolute value) of a scalar.

Functionality

$r \leftarrow |a|$

Prototypes

```
vsip_scalar_f vsip_mag_f(
                    vsip_scalar_f a);
vsip_scalar_i vsip_mag_i(
                    vsip_scalar_i a);
```

Arguments

`a` Argument

Return Value

Returns the magnitude.

Restrictions

Errors

Notes/References

Examples

See Also

`vsip_cmag_p`, and `vsip_cmagsq_p`

vsip_max_*p* Scalar Maximum

Computes the maximum of two scalars.

Functionality

$r \leftarrow \max\{a, b\}$

Prototypes

```
vsip_scalar_f vsip_max_f(
                    vsip_scalar_f a,
                    vsip_scalar_f b);
vsip_scalar_i vsip_max_i(
                    vsip_scalar_i a,
                    vsip_scalar_i b);
```

Arguments

a	Argument
b	Argument

Return Value

Returns the maximum value.

Restrictions

Errors

Notes/References

Examples

See Also

vsip_min_*p*

vsip_min_*p* Scalar Minimum

Computes the minimum of two scalars.

Functionality

$r \leftarrow \min\{a, b\}$

Prototypes

```
vsip_scalar_f vsip_min_f(
                    vsip_scalar_f a,
                    vsip_scalar_f b);
vsip_scalar_i vsip_min_i(
                    vsip_scalar_i a,
                    vsip_scalar_i b);
```

Arguments

a	Argument
b	Argument

Return Value

Returns the minimum value.

Restrictions

Errors

Notes/References

Examples

See Also

vsip_max_*p*

vsip_pow_*p* Scalar Power

Computes the power function of two scalars.

Functionality

$r \leftarrow x^y$

Prototypes

```
vsip_scalar_f vsip_pow_f(
                    vsip_scalar_f x,
                    vsip_scalar_ f y );
```

Arguments

`x`	Argument
`y`	Argument

Return Value

Returns the power function.

Restrictions

Errors

Notes/References

Examples

See Also

`vsip_exp_p`, `vsip_exp10_p`, `vsip_log_p`, `vsip_log10_p`, `vsip_cexp_p`, and `vsip_clog_p`

vsip_rsqrt_*p* Scalar Reciprocal Square Root

Computes the reciprocal square root of a scalar.

Functionality

$r \leftarrow 1/\sqrt{a}$

Prototypes

```
vsip_scalar_f vsip_rsqrt_f(
                    vsip_scalar_f a);
```

Arguments

`a`	Argument

Return Value

Returns the reciprocal square root.

Restrictions

For reciprocal square root calculation the argument must be greater than zero to be within the domain of the function. Results for cases where the argument is less than or equal zero is implementation dependent.

Errors

Notes/References

Examples

See Also

`vsip_sqrt_p`, `vsip_hypot_p`, and `vsip_csqrt_p`

vsip_sin_*p* Scalar Sine

Computes the sine of a scalar angle in radians.

Functionality

$r \leftarrow \sin \theta$

Prototypes

```
vsip_scalar_f vsip_sin_f(
                    vsip_scalar_f theta);
```

Arguments

`theta`	Angle in radians

Return Value

Returns the sine.

Restrictions

Errors

Notes/References

Input arguments are expressed in radians.

Examples

See Also

`vsip_acos_p`, `vsip_asin_p`, `vsip_atan_p`, `vsip_atan2_p`, `vsip_cos_p`, and `vsip_tan_p`

vsip_sinh_*p* Scalar Hyperbolic Sine

Computes the hyperbolic sine of a scalar.

Functionality

$r \leftarrow \sinh x$

Prototypes

```
vsip_scalar_f vsip_sinh_f(
                    vsip_scalar_f x);
```

Arguments

`x`	Argument

Return Value

Returns the hyperbolic sine.

Restrictions

The maximum domain without overflow is implementation dependent.

Errors

Notes/References

Examples

See Also

`vsip_cosh_p`, and `vsip_tanh_p`

vsip_sqrt_*p* Scalar Square Root

Computes the square root of a scalar.

Functionality

$r \leftarrow \sqrt{a}$

Prototypes

```
vsip_scalar_f vsip_sqrt_f(
                    vsip_scalar_f a);
```

Arguments

a	Argument

Return Value

Returns the square root.

Restrictions

For the square root calculation, the argument must be greater than or equal to zero to be within the domain of the function. Results for cases where the argument is less than zero is implementation dependent.

Errors

Notes/References

Examples

See Also

`vsip_rsqrt_p`, `vsip_hypot_p`, and `vsip_csqrt_p`

vsip_tan_*p* Scalar Tangent

Computes the tangent of a scalar angle in radians.

Functionality

$r \leftarrow \tan \theta$

Prototypes

```
vsip_scalar_f vsip_tan_f(
                    vsip_scalar_f theta);
```

Arguments

`theta`	Angle in radians

Return Value

Returns the tangent.

Restrictions

For arguments $\left(n + \frac{1}{2}\right)\pi$, the tangent function has a singularity. The result of these argument values are implementation dependent.

Errors

Notes/References

Examples

See Also

`vsip_acos_p`, `vsip_asin_p`, `vsip_atan_p`, `vsip_atan2_p`, `vsip_cos_p`, and `vsip_sin_p`

vsip_tanh_*p* Scalar Hyperbolic Tangent

Computes the hyperbolic tangent of a scalar.

Functionality

$r \leftarrow \tanh x$

Prototypes

```
vsip_scalar_f vsip_tanh_f(
                    vsip_scalar_f x);
```

Arguments

`x`	Argument

Return Value

Returns the hyperbolic tangent.

Restrictions

The maximum domain without overflow is implementation dependent.

Errors

Notes/References

Examples

See Also

`vsip_cosh_p`, and `vsip_sinh_p`

Complex Scalar Functions

Since ANSI C does not currently support a "complex" data type, a minimum set of complex scalar function is needed to support complex functions.

Many of these functions are very simple and may be implemented as macros for C compilers that do not support inline functions or for C compilers that generate more efficient code from macro forms.

Macros and functions are not interchangeable. The prototypes in this manual are defined as functions to avoid the potential ambiguities of macro forms. If a function is implemented as a macro, the documentation should specify such and state any consequent restrictions.

In order to support both macro and function implementations, many have a form similar to the complex scalar addition:

```
void vsip_CADD_f(
                    vsip_cscalar_f a,
                    vsip_cscalar_f b,
                    vsip_cscalar_f *r);
```

A macro only version could have been specified as:

```
vsip_CADD_f(
                    vsip_cscalar_f a,
                    vsip_cscalar_f b,
                    vsip_cscalar_f r);
```

because the macro handles the argument "*r*" through text replacement.

By using the first form, both the macro and function form work equally well and both macro forms generate equivalent code.

The notation for function foo:

$$r \leftarrow \text{foo}(x)$$

The "*r*" denotes the return value of the function for both the "Return By Argument Reference" and the "Return By Return Value" forms.

vsip_**arg**_*p*	Complex Scalar Argument
vsip_**cadd**_*p*	Complex Scalar Add
vsip_**cdiv**_*p*	Complex Scalar Divide
vsip_**cexp**_*p*	Complex Scalar Exponential
vsip_**cjmul**_*p*	Complex Conjugate Scalar Multiply
vsip_**clog**_*p*	Complex Scalar Log
vsip_**cmag**_*p*	Complex Scalar Magnitude
vsip_**cmagsq**_*p*	Complex Scalar Magnitude Squared
vsip_**cmplx**_*p*	Complex Scalar
vsip_**cmul**_*p*	Complex Scalar Multiply
vsip_**cneg**_*p*	Complex Scalar Negate
vsip_**conj**_*p*	Complex Scalar Conjugate
vsip_**crecip**_*p*	Complex Scalar Reciprocal
vsip_**csqrt**_*p*	Complex Scalar Square Root
vsip_**csub**_*p*	Complex Scalar Subtract
vsip_**imag**_*p*	Complex Scalar Imaginary

`vsip_polar_p`	Complex Scalar Polar
`vsip_real_p`	Complex Scalar Real
`vsip_rect_p`	Complex Scalar Rectangular

vsip_arg_*p* Complex Scalar Argument

Returns the *argument* in radians [-π, π] of a complex scalar.

Functionality

$$r \leftarrow \tan^{-1} \frac{\mathrm{Im}\{a\}}{\mathrm{Re}\{a\}}$$

Prototype

```
vsip_scalar_f vsip_arg_f(
                    vsip_cscalar_f a);
```

Arguments

`a`	Complex scalar argument

Return Value

Returns real scalar - in radians.

Restrictions

$\mathrm{Re}\{a\} = \mathrm{Im}\{a\} = 0$, invalid argument

$|\mathrm{Re}\{a\}| = |\mathrm{Im}\{a\}| = \infty$, invalid argument

This function may be implemented as a macro and may have restrictions on it usage.

Errors

Notes/References

This function is based on `vsip_atan2_p`.

Examples

See Also

`vsip_atan2_p`, and `vsip_polar_p`

vsip_cadd_*p* Complex Scalar Add

Computes the complex sum of two scalars.

Functionality

$r \leftarrow a + b$

Prototypes

Return By Argument Reference:

```
void vsip_CADD_f(
                    vsip_cscalar_f a,
                    vsip_cscalar_f b,
                    vsip_cscalar_f *r);
void vsip_RCADD_f(
                    vsip_scalar_f a,
                    vsip_cscalar_f b,
                    vsip_cscalar_f *r);
```

Return By Return Value:

```
vsip_cscalar_f vsip_cadd_f(
                    vsip_cscalar_f a,
                    vsip_cscalar_f b);
vsip_cscalar_f vsip_rcadd_f(
                    vsip_scalar_f a,
                    vsip_cscalar_f b);
```

Arguments

`a`	Augend - real/complex scalar
`b`	Addend - complex scalar
`*r`	Pointer to output sum - complex scalar

Return Value

Return complex sum.

Restrictions

The return by argument reference forms may be implemented as macros and may have restrictions on their usage.

Errors

Notes/References

Examples

See Also

`vsip_conj_p`, `vsip_cdiv_p`, `vsip_cexp_p`, `vsip_cjmul_p`, `vsip_clog_p`, `vsip_cmag_p`, `vsip_cmagsq_p`, `vsip_cmplx_p`, `vsip_cmul_p`, `vsip_cneg_p`, `vsip_crecip_p`, `vsip_csub_p`, and `vsip_csqrt_p`

vsip_cdiv_*p* Complex Scalar Divide

Computes the complex quotient of two scalars.

Functionality

$$r \leftarrow \frac{a}{b}$$

Prototypes

Return By Argument Reference:

```
void vsip_CRDIV_f(
                    vsip_cscalar_f a,
                    vsip_scalar_f b,
                    vsip_cscalar_f *r);
void vsip_CDIV_f(
                    vsip_cscalar_f a,
                    vsip_cscalar_f b,
                    vsip_cscalar_f *r);
```

Return By Return Value:

```
vsip_cscalar_f vsip_crdiv_f(
                    vsip_cscalar_f a,
                    vsip_scalar_f b);
vsip_cscalar_f vsip_cdiv_f(
                    vsip_cscalar_f a,
                    vsip_cscalar_f b);
```

Arguments

`a`	Numerator - complex scalar argument
`b`	Divisor - real/complex scalar argument
`*r`	Pointer to output quotient - complex scalar

Return Value

Returns complex scalar - quotient.

Restrictions

The return by argument reference forms may be implemented as macros and may have restrictions on their usage. The result of dividing by zero is implementation dependent.

Errors

Notes/References

The return by argument reference forms may be implemented as macros and may have restrictions on it usage.

Examples

See Also

`vsip_cadd_p`, `vsip_conj_p`, `vsip_cexp_p`, `vsip_cjmul_p`, `vsip_clog_p`, `vsip_cmag_p`, `vsip_cmagsq_p`, `vsip_cmplx_p`, `vsip_cmul_p`, `vsip_cneg_p`, `vsip_crecip_p`, `vsip_csub_p`, and `vsip_csqrt_p`

vsip_cexp_*p* Complex Scalar Exponential

Computes the complex exponential of a scalar.

Functionality

$r \leftarrow e^{a} = (\cos \omega + j \sin \omega) \cdot e^{\sigma}$

Where: $\mathrm{Re}\{a\} = \sigma$, $\mathrm{Im}\{a\} = \omega$

And $j \equiv \sqrt{-1}$

Prototypes

Return By Argument Reference:

```
void vsip_CEXP_f(
                    vsip_cscalar_f a,
                    vsip_cscalar_f *r);
```

Return By Return Value:

```
vsip_cscalar_f vsip_cexp_f(
                    vsip_cscalar_f a);
```

Arguments

`a`	Complex scalar argument
`*r`	Pointer to output exponential - complex scalar

Return Value

Returns complex scalar – complex scalar exponential.

Restrictions

$|\mathrm{Re}\{a\}| = \infty$, invalid argument

The return by argument reference form may be implemented as a macro and may have restrictions on its usage.

Errors

Notes/References

Examples

See Also

`vsip_cadd_p`, `vsip_conj_p`, `vsip_cdiv_p`, `vsip_cjmul_p`, `vsip_clog_p`, `vsip_cmag_p`, `vsip_cmagsq_p`, `vsip_cmplx_p`, `vsip_cmul_p`, `vsip_cneg_p`, `vsip_crecip_p`, `vsip_csub_p`, and `vsip_csqrt_p`

vsip_cjmul_*p* Complex Conjugate Scalar Multiply

Computes the product a complex scalar with the conjugate of a second complex scalar.

Functionality

$r \leftarrow a \cdot b^*$

Where "*" denotes complex conjugate.

Prototypes

Return By Argument Reference:

```
void vsip_CJMUL_f(
                    vsip_cscalar_f a,
                    vsip_cscalar_f b,
                    vsip_cscalar_f *r);
```

Return By Return Value:

```
vsip_cscalar_f vsip_cjmul_f(
                    vsip_cscalar_f a,
                    vsip_cscalar_f b);
```

Arguments

`a`	Multiplicand - complex scalar argument
`b`	Conjugate multiplier - complex scalar argument
`*r`	Pointer to output product - complex scalar

Return Value

Returns complex scalar - conjugate product.

Restrictions

The return by argument reference form may be implemented as a macro and may have restrictions on its usage.

Errors

Notes/References

Examples

See Also

`vsip_cadd_p`, `vsip_conj_p`, `vsip_cdiv_p`, `vsip_cexp_p`, `vsip_clog_p`, `vsip_cmag_p`, `vsip_cmagsq_p`, `vsip_cmplx_p`, `vsip_cmul_p`, `vsip_cneg_p`, `vsip_crecip_p`, `vsip_csub_p`, and `vsip_csqrt_p`

vsip_clog_*p* Complex Scalar Log

Computes the complex natural logarithm of a scalar.

Functionality

$r \leftarrow \log_e a = \log_e|a| + j \cdot \arg(a)$

Where $j \equiv \sqrt{-1}$

Prototypes

Return By Argument Reference:

```
void vsip_CLOG_f(
                    vsip_cscalar_f a,
                    vsip_cscalar_f *r);
```

Return By Return Value:

```
vsip_cscalar_f vsip_clog_f(
                    vsip_cscalar_f a);
```

Arguments

`a`	Complex scalar argument
`*r`	Pointer to output natural log - complex scalar

Return Value

Returns complex scalar - natural log of a complex scalar.

Restrictions

$\mathrm{Re}\{a\} = \mathrm{Im}\{a\} = 0$, invalid argument

$|\mathrm{Re}\{a\}| = |\mathrm{Im}\{a\}| = \infty$, invalid argument

The return by argument reference form may be implemented as a macro and may have restrictions on its usage.

Errors

Notes/References

Examples

See Also

`vsip_cadd_p`, `vsip_conj_p`, `vsip_cdiv_p`, `vsip_cexp_p`, `vsip_cjmul_p`, `vsip_cmag_p`, `vsip_cmagsq_p`, `vsip_cmplx_p`, `vsip_cmul_p`, `vsip_cneg_p`, `vsip_crecip_p`, `vsip_csub_p`, and `vsip_csqrt_p`

vsip_cmag_*p* Complex Scalar Magnitude

Computes the magnitude of a complex scalar.

Functionality

$$r \leftarrow |a| = \sqrt{a \cdot a^*}$$

Where "*" denotes complex conjugate.

Prototypes

Return By Return Value:

```
vsip_scalar_f  vsip_cmag_f(
                    vsip_cscalar_f a);
```

Arguments

`a`	Complex scalar argument

Return Value

Returns real scalar - magnitude.

Restrictions

This function may be implemented as a macro and may have restrictions on its usage.

Errors

Notes/References

Implementation with intermediate overflow is allowed but must be documented.

Examples

See Also

`vsip_cadd_p`, `vsip_conj_p`, `vsip_cdiv_p`, `vsip_cexp_p`, `vsip_cjmul_p`, `vsip_clog_p`, `vsip_cmagsq_p`, `vsip_cmplx_p`, `vsip_cmul_p`, `vsip_cneg_p`, `vsip_crecip_p`, `vsip_csub_p`, and `vsip_csqrt_p`

vsip_cmagsq_*p* Complex Scalar Magnitude Squared

Computes the magnitude squared of a complex scalar.

Functionality

$r \leftarrow |a|^2 = a \cdot a^*$

Where "*" denotes complex conjugate.

Prototypes

Return By Return Value:

```
vsip_scalar_f  vsip_cmagsq_f(
                    vsip_cscalar_f a);
```

Arguments

a	Complex scalar argument

Return Value

Returns real magnitude squared.

Restrictions

This function may be implemented as a macro and may have restrictions on its usage.

Errors

Notes/References

Examples

See Also

vsip_cadd_*p*, vsip_conj_*p*, vsip_cdiv_*p*, vsip_cexp_*p*, vsip_cjmul_*p*, vsip_clog_*p*, vsip_cmag_*p*, vsip_cmplx_*p*, vsip_cmul_*p*, vsip_cneg_*p*, vsip_crecip_*p*, vsip_csub_*p*, and vsip_csqrt_*p*

vsip_cmplx_*p* Complex Scalar

Form a complex scalar from two real scalars.

Functionality

$r \leftarrow a + j \cdot b;\ a,\ b \in \Re$

Where $j \equiv \sqrt{-1}$

Prototypes

Return By Argument Reference:

```
void vsip_CMPLX_f(
                    vsip_scalar_f a,
                    vsip_scalar_f b,
                    vsip_cscalar_f *r);
```

Return By Return Value:

```
vsip_cscalar_f vsip_cmplx_f(
                    vsip_scalar_f a,
                    vsip_scalar_f b);
```

Arguments

`a`	Real part - real scalar argument
`b`	Imaginary part - real scalar argument
`*r`	Pointer to output - complex scalar

Return Value

Returns complex scalar.

Restrictions

The return by argument reference form may be implemented as a macro and may have restrictions on its usage.

Errors

Notes/References

Examples

See Also

`vsip_polar_p`, `vsip_real_p`, `vsip_imag_p`, and `vsip_rect_p`

vsip_cmul_*p* Complex Scalar Multiply

Computes the complex product of two scalars.

Functionality

$r \leftarrow a \cdot b$

Prototypes

Return By Argument Reference:

```
void vsip_CMUL_f(
                    vsip_cscalar_f a,
                    vsip_cscalar_f b,
                    vsip_cscalar_f *r);
void vsip_RCMUL_f(
                    vsip_scalar_f a,
                    vsip_cscalar_f b,
                    vsip_cscalar_f *r);
```

Return By Return Value:

```
vsip_cscalar_f vsip_cmul_f(
                    vsip_cscalar_f a,
                    vsip_cscalar_f b);
vsip_cscalar_f vsip_rcmul_f(
                    vsip_scalar_f a,
                    vsip_cscalar_f b);
```

Arguments

`a`	Multiplicand - real/complex scalar argument
`b`	Multiplier - complex scalar argument
`*r`	Pointer to output product - complex scalar

Return Value

Returns complex scalar - product.

Restrictions

The return by argument reference forms may be implemented as macros and may have restrictions on their usage.

Errors

Notes/References

Examples

See Also

`vsip_cadd_p`, `vsip_conj_p`, `vsip_cdiv_p`, `vsip_cexp_p`, `vsip_cjmul_p`, `vsip_clog_p`, `vsip_cmag_p`, `vsip_cmagsq_p`, `vsip_cmplx_p`, `vsip_cneg_p`, `vsip_crecip_p`, `vsip_csub_p`, and `vsip_csqrt_p`

vsip_cneg_*p* Complex Scalar Negate

Computes the negation of a complex scalar.

Functionality

$r \leftarrow -a$

Prototypes

Return By Argument Reference:

```
void vsip_CNEG_f(
                    vsip_cscalar_f a,
                    vsip_cscalar_f *r);
```

Return By Return Value:

```
vsip_cscalar_f vsip_cneg_f(
                    vsip_cscalar_f a);
```

Arguments

`a`	Complex scalar argument
`*r`	Pointer to output - complex scalar

Return Value

Returns negated complex scalar.

Restrictions

The return by argument reference form may be implemented as a macro and may have restrictions on its usage.

Errors

Notes/References

Examples

See Also

`vsip_cadd_p`, `vsip_conj_p`, `vsip_cdiv_p`, `vsip_cexp_p`, `vsip_cjmul_p`, `vsip_clog_p`, `vsip_cmag_p`, `vsip_cmagsq_p`, `vsip_cmplx_p`, `vsip_cmul_p`, `vsip_crecip_p`, `vsip_csub_p`, and `vsip_csqrt_p`

vsip_conj_*p* Complex Scalar Conjugate

Computes the complex conjugate of a scalar.

Functionality

$r \leftarrow a^*$

Where "*" denotes complex conjugate.

Prototypes

Return By Argument Reference:

```
void vsip_CONJ_f(
                    vsip_cscalar_f a,
                    vsip_cscalar_f *r);
```

Return By Return Value:

```
vsip_cscalar_f vsip_conj_f(
                    vsip_cscalar_f a);
```

Arguments

`a`	Complex scalar argument
`*r`	Pointer to output conjugate - complex scalar

Return Value

Returns complex scalar - conjugate.

Restrictions

The return by argument reference form may be implemented as a macro and may have restrictions on its usage.

Errors

Notes/References

Examples

See Also

`vsip_cadd_p`, `vsip_cdiv_p`, `vsip_cexp_p`, `vsip_cjmul_p`, `vsip_clog_p`, `vsip_cmag_p`, `vsip_cmagsq_p`, `vsip_cmplx_p`, `vsip_cmul_p`, `vsip_cneg_p`, `vsip_crecip_p`, `vsip_csub_p`, and `vsip_csqrt_p`

vsip_crecip_*p* Complex Scalar Reciprocal

Computes the reciprocal of a complex scalar.

Functionality

$$r \leftarrow \frac{1}{a}$$

Prototypes

Return By Argument Reference:

```
vsip_CRECIP_f(
                    vsip_cscalar_f a,
                    vsip_cscalar_f *r);
```

Return By Return Value:

```
vsip_cscalar_f vsip_crecip_f(
                    vsip_cscalar_f a);
```

Arguments

`a`	Complex scalar argument
`*r`	Pointer to output reciprocal – complex scalar

Return Value

Returns complex scalar - reciprocal.

Restrictions

The return by argument reference form may be implemented as a macro and may have restrictions on its usage. The result of dividing by zero is implementation dependent.

Errors

Notes/References

Examples

See Also

`vsip_cadd_p`, `vsip_conj_p`, `vsip_cdiv_p`, `vsip_cexp_p`, `vsip_cjmul_p`, `vsip_clog_p`, `vsip_cmag_p`, `vsip_cmagsq_p`, `vsip_cmplx_p`, `vsip_cmul_p`, `vsip_cneg_p`, `vsip_csub_p`, and `vsip_csqrt_p`

vsip_csqrt_*p* Complex Scalar Square Root

Computes the square root of a complex scalar.

Functionality

Let $\theta \equiv \arctan\left(\frac{\mathrm{Im}\{a\}}{\mathrm{Re}\{a\}}\right)$

$$r \leftarrow \sqrt{|a|} \cdot \left[\cos\left(\frac{\theta}{2}\right) + j\sin\left(\frac{\theta}{2}\right)\right]$$

Where $j \equiv \sqrt{-1}$

Prototypes

Return By Argument Reference:

```
void vsip_CSQRT_f(
                    vsip_cscalar_f a,
                    vsip_cscalar_f *r,);
```

Return By Return Value:

```
vsip_cscalar_f vsip_csqrt_f(
                    vsip_cscalar_f a);
```

Arguments

`a`	Complex scalar argument
`*r`	Pointer to output square root - complex scalar

Return Value

Returns square root of complex scalar.

Restrictions

The return by argument reference form may be implemented as a macro and may have restrictions on its usage.

Errors

Notes/References

Examples

See Also

`vsip_cadd_p`, `vsip_conj_p`, `vsip_cdiv_p`, `vsip_cexp_p`, `vsip_cjmul_p`, `vsip_clog_p`, `vsip_cmag_p`, `vsip_cmagsq_p`, `vsip_cmplx_p`, `vsip_cmul_p`, `vsip_cneg_p`, `vsip_crecip_p`, and `vsip_csub_p`

vsip_csub_*p* Complex Scalar Subtract

Computes the complex difference of two scalars.

Functionality

$r \leftarrow a - b$

Prototypes

Return By Argument Reference:

```
void vsip_CSUB_f(
                    vsip_cscalar_f a,
                    vsip_cscalar_f b,
                    vsip_cscalar_f *r,);
void vsip_RCSUB_f(
                    vsip_scalar_f a,
                    vsip_cscalar_f b,
                    vsip_cscalar_f *r,);
void vsip_CRSUB_f(
                    vsip_cscalar_f a,
                    vsip_scalar_f b,
                    vsip_cscalar_f *r,);
```

Return By Return Value:

```
vsip_cscalar_f vsip_csub_f(
                    vsip_cscalar_f a,
                    vsip_cscalar_f b);
vsip_cscalar_f vsip_rcsub_f(
                    vsip_scalar_f a,
                    vsip_cscalar_f b);
vsip_cscalar_f vsip_crsub_f(
                    vsip_cscalar_f a,
                    vsip_scalar_f b);
```

Arguments

`a`	Minuend - real/complex scalar argument
`b`	Subtrahend - real/complex scalar
`*r`	Pointer to output difference – complex scalar

Return Value

Returns complex scalar – difference.

Restrictions

The return by argument reference forms may be implemented as macros and may have restrictions on their usage.

Errors

Notes/References

Examples

See Also

`vsip_cadd_p`, `vsip_conj_p`, `vsip_cdiv_p`, `vsip_cexp_p`, `vsip_cjmul_p`, `vsip_clog_p`, `vsip_cmag_p`, `vsip_cmagsq_p`, `vsip_cmplx_p`, `vsip_cmul_p`, `vsip_cneg_p`, `vsip_crecip_p`, and `vsip_csqrt_p`

vsip_imag_*p* Complex Scalar Imaginary

Extract the imaginary part of a complex scalar.

Functionality

$r \leftarrow \mathrm{Im}\{a\}$

Prototypes

Return By Return Value:

```
vsip_scalar_f vsip_imag_f(
                    vsip_cscalar_f a);
```

Arguments

`a`	Complex scalar argument

Return Value

Returns a real scalar containing the imaginary part of the complex scalar input.

Restrictions

This function may be implemented as a macro and may have restrictions on its usage.

Errors

Notes/References

Examples

See Also

`vsip_cmplx_p`, `vsip_polar_p`, `vsip_real_p`, and `vsip_rect_p`

vsip_polar_*p* Complex Scalar Polar

Convert a complex scalar from rectangular to polar form. The polar data consists of a real scalar containing the radius and a corresponding real scalar containing the argument (angle) of the complex scalar.

Functionality

$$\rho \leftarrow |a|$$

$$\theta \leftarrow \arctan\left(\frac{\mathrm{Im}\{a\}}{\mathrm{Re}\{a\}}\right)$$

Prototypes

Return By Argument Reference:

```
void vsip_polar_f(
                    vsip_cscalar_f a,
                    vsip_scalar_f *radius,
                    vsip_scalar_f *theta);
```

Arguments

`a`	Complex scalar argument (rectangular form)
`*radius`	Pointer to output radius - real scalar
`*theta`	Pointer to output angle - real scalar

Return Value

None.

Restrictions

$\mathrm{Re}\{a\} = \mathrm{Im}\{a\} = 0$, invalid argument

$|\mathrm{Re}\{a\}| = |\mathrm{Im}\{a\}| = \infty$, invalid argument

This function may be implemented as a macro and may have restrictions on its usage.

Errors

Notes/References

In VSIPL, complex numbers are always in rectangular (Cartesian) format. The polar form is represented by two real scalars.

Examples

See Also

`vsip_cmplx_p`, `vsip_imag_p`, `vsip_real_p`, and `vsip_rect_p`

vsip_real_*p* Complex Scalar Real

Extract the real part of a complex scalar.

Functionality

$r \leftarrow \mathrm{Re}\{a\}$

Prototypes

Return By Return Value:

```
vsip_scalar_f vsip_real_f(
                    vsip_cscalar_f a);
```

Arguments

`a`	Complex scalar argument

Return Value

Returns a real scalar containing the real part of the complex scalar input.

Restrictions

This function may be implemented as a macro and may have restrictions on its usage.

Errors

Notes/References

Examples

See Also

`vsip_cmplx_p`, `vsip_imag_p`, `vsip_polar_p`, and `vsip_rect_p`

vsip_rect_*p* Complex Scalar Rectangular

Convert a pair of real scalars from complex polar to complex rectangular form.

Functionality

$r \leftarrow \text{radius} \cdot (\cos\theta + j \cdot \sin\theta)$

Where $j \equiv \sqrt{-1}$

Prototypes

Return By Argument Reference:

```
void vsip_RECT_f(
                    vsip_scalar_f radius,
                    vsip_scalar_f theta,
                    vsip_cscalar_f *r);
```

Return By Return Value:

```
vsip_cscalar_f vsip_rect_f(
                    vsip_scalar_f radius,
                    vsip_scalar_f theta);
```

Arguments

`radius`	Radius – real scalar
`theta`	Angle – real scalar
`*r`	Pointer to output – complex scalar in rectangular form

Return Value

Returns complex scalar in rectangular form.

Restrictions

The return by argument reference form may be implemented as a macro and may have restrictions on its usage.

In VSIPL, complex numbers are always in rectangular (Cartesian) format. The polar form is represented by two real scalars.

Errors

Notes/References

Examples

See Also

`vsip_cmplx_p`, `vsip_imag_p`, `vsip_polar_p`, and `vsip_real_p`

Index Scalar Functions

`vsip_matindex`	Matrix Row Index
`vsip_mcolindex`	Matrix Column Index
`vsip_mrowindex`	Matrix Index
`vsip_tenindex`	Tensor Index
`vsip_txindex`	Tensor *x* Index
`vsip_tyindex`	Tensor *y* Index
`vsip_tzindex`	Tensor *z* Index

vsip_matindex Matrix Index

Form a matrix index from two vector indices.

Functionality

(*row*, *col*) ← *row*, *col*

Prototypes

Return By Argument Reference:

```
void vsip_MATINDEX(
                    vsip_scalar_vi row,
                    vsip_scalar_vi col,
                    vsip_scalar_mi *r);
```

Return By Return Value:

```
vsip_scalar_mi vsip_matindex(
                    vsip_scalar_vi row,
                    vsip_scalar_vi col);
```

Arguments

`row`	Row - vector index
`col`	Column - vector index
`*r`	Pointer to output - matrix index

Return Value

Returns the matrix index.

Restrictions

Errors

Notes/References

Note to implementors: since VSIPL does not specify what the *base integer* type is; the arguments must be of type `vsip_scalar_vi`, which in an unsigned integer type sufficient to index the largest supported vector.

Examples

See Also

`vsip_mcolindex`, and `vsip_mrowindex`

vsip_mcolindex Matrix Column Index

Returns the column vector index from a matrix index.

Functionality

$col \leftarrow (row, col)$

Prototypes

```
vsip_scalar_vi vsip_mcolindex(
                     vsip_scalar_mi index);
```

Arguments

`index` Matrix index

Return Value

Returns the column vector index.

Restrictions

Errors

Notes/References

Examples

See Also

`vsip_matindex`, and `vsip_mcolindex`

vsip_mrowindex Matrix Row Index

Returns the row vector index from a matrix index.

Functionality

row ← (*row*, *col*)

Prototypes

```
vsip_scalar_vi vsip_mrowindex(
                     vsip_scalar_mi index);
```

Arguments

`index` Matrix index

Return Value

Returns the row vector index.

Restrictions

Errors

Notes/References

Examples

See Also

`vsip_matindex`, and `vsip_mrowindex`

vsip_tenindex Tensor Index

Form a tensor index from three vector indices.

Functionality

$(z, y, x) \leftarrow z, y, x$

Prototypes

Return By Argument Reference:

```
void vsip_TENINDEX(
                    vsip_scalar_vi z,
                    vsip_scalar_vi y,
                    vsip_scalar_vi x,
                    vsip_scalar_ti *r);
```

Return By Return Value:

```
vsip_scalar_ti vsip_tenindex(
                    vsip_scalar_vi z,
                    vsip_scalar_vi y,
                    vsip_scalar_vi x);
```

Arguments

`z`	*z* - vector index
`y`	*y* - vector index
`x`	*x* - vector index
`*r`	Pointer to output - tensor index

Return Value

Returns the tensor index.

Restrictions

Errors

Notes/References

Examples

See Also

`vsip_tenindex`, `vsip_txindex`, `vsip_tyindex`, and `vsip_tzindex`

vsip_txindex Tensor X Index

Returns the *x* index from a tensor index.

Functionality

$x \leftarrow (z, y, x)$

Prototypes

```
vsip_scalar_vi vsip_txindex(
                      vsip_scalar_ti index);
```

Arguments

`index` Tensor index

Return Value

Returns the tensor *x* index.

Restrictions

Errors

Notes/References

Examples

See Also

`vsip_tenindex`, `vsip_tyindex`, and `vsip_tzindex`

vsip_tyindex Tensor Y Index

Returns the *y* index from a tensor index.

Functionality

$y \leftarrow (z, y, x)$

Prototypes

```
vsip_scalar_vi vsip_tyindex(
                      vsip_scalar_ti index);
```

Arguments

`index` Tensor index

Return Value

Returns the tensor *y* index.

Restrictions

Errors

Notes/References

Examples

See Also

`vsip_tenindex`, `vsip_txindex`, and `vsip_tzindex`

vsip_tzindex Tensor Z Index

Returns the z index from a tensor index.

Functionality

$z \leftarrow (z, y, x)$

Prototypes

```
vsip_scalar_vi vsip_tzindex(
                      vsip_scalar_ti index);
```

Arguments

`index` Tensor index

Return Value

Returns the tensor z index.

Restrictions

Errors

Notes/References

Examples

See Also

`vsip_tenindex`, `vsip_txindex`, and `vsip_tyindex`

RANDOM NUMBER GENERATION

Introduction

This section covers the VSIPL random number generation functionality. Two pseudo random number generators are defined. The first generates uniform random numbers over the open interval zero to one. The second approximates a Gaussian random number with zero mean and unit variance.

Random Numbers

The *create* and *destroy* random number generator state are used for both scalar, and "by element" random number generation. The "by element" random number generators are required to produce the equivalent result to applying the scalar generators to each element in the order that proceeds from the minimum stride to the maximum stride dimension.

[For a matrix Z, where the stride between elements of a row is less than the stride between elements of a column, where x_j denotes the jth output of the generator:

$z_{0,0}, \leftarrow x_i,\quad z_{0,1}, \leftarrow x_{i+1},\quad z_{0,2}, \leftarrow x_{i+2}, \ldots, z_{1,0}, \leftarrow x_{i+N}, z_{1,1}, \leftarrow x_{i+N+1}, \ldots, z_{M-1,N-1}, \leftarrow x_{i+M\cdot N-1}]$

VSIPL Random Number Generator Functions

VSIPL specifies a portable random number generator. The details of the generator follow.

The code below implements a combined 32 bit random number generator comprising variants (RAN0 and RAN1) of two popular 32 bit random number generators.

RAN0 is loosely based (the constants chosen by D.E Knuth and H.W. Lewis are used) on a linear congruential random number generator popularized in Numerical Recipes [1],

$x_i \leftarrow (a \cdot x_{i-1} + c) \bmod m$

where $a = 1664525$, $c = 1013904223$, and $m = 2^{32}$.

RAN1 is based on the popular random number generator,

$y_i \leftarrow (a \cdot y_{i-1} + c) \bmod m$

where $a = 69069$, $c = 1$, and $m = 2^{32}$. In the VSIPL version of RAN1, the addend c is set equal to three instead of one.

The uniformly distributed 32 bit unsigned random integer created by the combined generator represents the difference between the uniformly distributed 32 bit unsigned integers created by RAN0 and RAN1. The combined random number generator maintains two separate seed sequences. One for RAN0, and the other for RAN1.

Both RAN0 and RAN1 have periods of 2^{32}-1 iterations. Starting with a given pair of seeds for the initial state, RAN0 and RAN1 complete periods and generate a state with the starting seed pair at the same iteration. To increase the period of the combined generator, the seed for RAN1 is incremented by one at the end of each full period. This ensures that the starting state (pair of seeds) will not be generated until RAN1 has passed through 2^{32} periods. Thus, the period for the combined generator is 2^{64} iterations.

To promote independence in a parallel processing environment, two actions are taken. The period of RAN0 is evenly divided among the parallel threads, and the addend constant used by RAN1 is uniquely assigned to each thread. These actions ensure that no two threads ever use exactly the same sequence of initial state seed pairs.

To divide the period of RAN0 evenly among threads we first note that we can generate the sequence $x_0, x_1, x_2, x_4, x_8, \ldots$ by applying the following recursion:

$$
\begin{aligned}
x_0 & \\
x_1 &= (x_0 \cdot a + c) \bmod m \\
x_2 &= (x_1 \cdot a + c) \bmod m \\
&= ((x_0 \cdot a + c) \cdot a + c) \bmod m \\
&= (x_0 \cdot a^2 + a \cdot c + c) \bmod m \\
&= (x_0 \cdot a \cdot a + (a+1) \cdot c) \bmod m \\
x_4 &= (x_0 \cdot a^4 + a^3 \cdot c + a^2 \cdot c + a \cdot c + c) \bmod m \\
&= (x_0 \cdot a^2 \cdot a^2 + (a^3 + a^2 + a + 1) \cdot c) \bmod m \\
&= (x_0 \cdot a^2 \cdot a^2 + (a^2 \cdot (a+1) + (a+1)) \cdot c) \bmod m \\
&= (x_0 \cdot a^2 \cdot a^2 + (a^2+1) \cdot (a+1) \cdot c) \bmod m \\
x_8 &= (x_0 \cdot a^4 \cdot a^4 + (a^4+1) \cdot (a^2+1) \cdot (a+1) \cdot c) \bmod m \\
&\ldots \\
x_{2^i} &= (A \cdot x_0 + C) \bmod m \\
&= \left(x_0 \cdot a^{2^i} + \left(a^{2^{i-1}} + 1\right) \cdot \left(a^{2^{i-2}} + 1\right) \cdot \ldots \cdot (a+1) \cdot c\right) \bmod m
\end{aligned}
$$

After i-1 iterations, each term of new generator (using A and C) skips over 2^i terms of the basic sequence generated by RAN0.

The term A may be calculated by repeated squaring of a, (i-1) times modulo m. The term C may be calculated in parallel with A by repeatedly adding one to the previous partial product of A and multiplying the previous partial product of C by the sum, (i-1) times. The algorithm is shown below.

Iteration	A	C
0	a	c
1	a^2	$(a + 1) \cdot c$
2	a^4	$(a^2 + 1) \cdot (a + 1) \cdot c$
3	a^8	$(a^4 + 1) \cdot (a^2 + 1) \cdot (a + 1) \cdot c$

The technique is used in the first "`for`" loop within `vsip_randcreate` to efficiently skip to the starting state assigned to a given thread for use by RAN0.

In a parallel environment, each thread is assigned an addend for RAN1 which corresponds to the prime number taken from the sequence of prime numbers greater than or equal to three. Thus the first thread uses three as the addend for RAN1, the second uses five as the addend for RAN1, the third uses seven as the addend for RAN1, etc. This technique assigns a different RAN1 random number generator to each thread.

By using the two techniques together, we define a very low overhead random number generator with sub-sequence statistical characteristics better than those of either RAN0 or RAN1. The new generator has a period of 2^{64} and can be used safely in many parallel processing environments. In particular, statistically independent subsequences of lengths up to approximately $2^{64}/N$, where N is the number of parallel sub-sequences required, are ensured.

References:

[1] See pp. 275-276 of *Numerical Recipes in FORTRAN, The Art of Scientific Computing*, Second Edition, William H. Press, et al., Cambridge University Press, 1992.

Sample Implementation

The following is a sample C implementation of two uniform random number generators. The comments have been formatted for readability.

```
#include <float.h>
#include <limits.h>
#include <stdlib.h>
#include <math.h>

#include "private/vsip_scalar_typedefs.h"
#include "vsip.h"
/* Typedefs for unsigned and signed exactly 32 bit integers uint32_t and int32_t   */
#include "inttypes.h"
#define A0 1664525                    /* Parameters for RAN0                 */
#define C0 1013904223
#define A1 69069                      /* Parameters for RAN1                 */
#define C1 3
#define RAN(state, A, C ) A*( state )+C
#define RAN0( ptr ) RAN( ptr->seed0,ptr->a0,ptr->c0 )
#define RAN1( ptr ) RAN( ptr->seed1,ptr->a1,ptr->c1 )
/*in*******************************************************************
```

The type of `vsip_randstate` in "`vsip.h`" should be an incomplete type definition. The actual type should be opaque and defined in a place like "`private/pvsip_scalar_typedefs.h`"

```
typedef struct vsip_rand_object {
   double   double_scale;
   float    float_scale;
   uint32_t seed0;
   uint32_t seed1;
   uint32_t seed2;
   int32_t  numseqs;
   int32_t  id;
   uint32_t a0;
   uint32_t c0;
   uint32_t a1;
   uint32_t c1;
} vsip_randstate;
**out*******************************************************************/
vsip_randstate *vsip_randcreate(
  vsip_index seed,                    /* Initial user seed                   */
  vsip_index numseqs,                 /* Number of sub-sequences             */
  vsip_index id,                      /* Subsequence id (0 < id ≤ numseqs)   */
  vsip_rng portable)                  /* Portable or non-portable sequence   */
{ /*  portable is ignored */
  uint32_t A=A0,C=C0;                 /* Initialize RAN                      */
  int32_t i;
  uint32_t mask,skip;
  vsip_randstate* state;
  state = (vsip_randstate* )malloc(sizeof(vsip_randstate));
  state->double_scale = (double)pow((double)2.0,(double)(-32));
  state->float_scale  = (float)pow((double)2.0,(double)(-24));
  state->seed0   = seed;
  state->seed1   = 1;
```

```
  state->seed2   = 1;
  state->numseqs = numseqs;
  state->id      = id;
  state->a0      = A0;
  state->c0      = C0;
  state->a1      = A1;
  state->c1      = C1;
/*in*********************************************************************
```

Find the *skip* size by dividing 2^{31}-1 (2^{32} cannot be represented in 32 bits), and multiplying the quotient by the relative thread *id*.

```
**out********************************************************************/
  skip = ((UINT_MAX/numseqs)*(id-1));
/*in*********************************************************************
```

Given a starting *seed*, the code below generates the starting seed for the *id*'th sub-sequence of a set of *numseqs* sub-sequences of RAN0.

With each loop iteration, a new random number generator is created – on the fly – from the base random number generator RAN0. The new generator issues a sub-sequence of the original sequence generated by RAN0, such that given the same starting *seed*, each term of the sub-sequence corresponds to every (2^i)'th term of the original RAN0 sequence, where *i* corresponds to the loop iteration.

To understand how this is used below, first note that the 32 bit unsigned integer *skip* may be viewed as a polynomial

$$skip \equiv P(31) \cdot 2^{31} + P(30) \cdot 2^{30} + \ldots + P(1) \cdot 2 + P(0)$$

where

$$P(i) \in \{0,1\}, \text{ for } i = 0, 1, 2, \ldots, 31$$

Thus, each value $P(i)$ is the setting (1 or 0) of the ith bit of *skip*.

In the loop below, the current generator (starting with RAN0) is applied to the current seed *S* if and only if the ith (*i* is the loop iterator) bit of *skip* is a one, i.e. $P(i) = 1$. At each loop iteration, a new generator is created that skips the next 2^{i+1} (*i* is the loop iterator) terms of the original RAN0 sequence. After 32 iterations, the thirty two bits of *skip* are exhausted and the current value of seed *S* is returned as the starting seed for the id^{th} thread.

As an example of how the scheme works, assume that we are creating the seed for the fifth of fifteen threads. The value of *skip* is

$$skip = ((2^{32}-1)/15)*(5-1) = (4294967295/15)*4 = 286331153*4 = 1145324612$$

which in hexadecimal is

$$skip = 44444444 = 2^{30} + 2^{26} + 2^{22} + 2^{18} + 2^{14} + 2^{10} + 2^{6} + 2^{2}$$

Thus, to create the correct seed we need to skip, in succession, 2^2 terms, 2^6 terms, 2^{10} terms, 2^{14} terms, 2^{18} terms, 2^{22} terms, and 2^{26} terms of the base RAN0 sequence. This is accomplished in the loop below by applying the current random number generator only on iterations 2, 6, 10, 14, 18, 22, 26, and 30. Although used on only eight of thirty two iterations, a new generator is created on each iteration.

```
**out********************************************************************/
mask = 1;
  for( i=0; i<32; i++ ){                /* Update seed if bit is set          */
    if( mask & skip ){
      state->seed0 = A*(state->seed0) + C;
```

```
    }
    C = (A+1)*C;                          /* Generate new offset constant        */
    A = A*A;                              /* Generate new multiplier constant    */
    mask <<= 1;
  }
/*in*********************************************************************
```

Set $C1$ to the id^{th} prime, starting at $C1 = 3$.

(The following is a very simple prime number generator. A pre-computed table of primes could be substituted.)

```
**out********************************************************************/
  for( i=1; i<id; i++ ){
    int32_t loop_max;
    state->c1 += 2;
    loop_max = (int32_t)sqrt( (double)state->c1 );
    C = 3;
    while( C <= loop_max ){
      while( ( (state->c1 % C) != 0 ) && ( C <= loop_max ) ) C += 2;
        if( (state->c1 % C) == 0 ){
          C = 3;
          state->c1 += 2;
          loop_max = (int32_t)sqrt( (double)state->c1 );
        }
     }
  }
  return( state );
}
void vsip_randdestroy( vsip_randstate* state )
{
   free(state);
}
/*in*********************************************************************
```

IEEE-754 Dependent Code

The following is implementation dependent. It is defined for IEEE-754 single and double precision floating point where `float` is single and `double` is double precision. (For non-IEEE 754 implementations the closest representable floating point number to the IEEE-754 number should be returned.)

Create the mask used to convert an unsigned integer to a single precision IEEE-754 float. The mask forces the sign and excess 127 exponent for a single precision IEEE-754 floating point number (1.0) into the upper nine bits of a 32 bit unsigned integer that has been shifted right nine bit positions. The low mantissa bit is set to one to force the number away from a true IEEE zero when it is normalized by subtracting 1.0.

```
**out********************************************************************/
static uint32_t vsip_random_fmask = 0x3f800001;
/*in*********************************************************************
```

Function `vsip_randu_f`:

- Takes a `vsip_randstate*` argument (which was created by `vsip_randcreate`);
- Updates the state seeds using macro functions RAN0 and RAN1;
- Forms a temporary 32 bit unsigned seed from the difference between the two state seeds;

- Right shifts the temporary seed eight bit positions (the width of a single precision IEEE-754 floating point number's exponent field);
- Bitwise OR's a one into the low bit; converts the integer contained in the interval $(0,2^{24}-1)$ to a single precision IEEE-754 floating point number;
- Scales the floating point number by 2^{-24} to force the result into the open interval (0.0,1.0);
- And finally returns the floating point result as the function return value.

```
**out*****************************************************************/
float vsip_randu_f( vsip_randstate* state )
/* Returns a float uniform random number within the open interval (0, 1)          */
{
  float    temp;
  uint32_t itemp;
  state->seed0 = RAN0( state );
  state->seed1 = RAN1( state );
  itemp  = (state->seed0 - state->seed1);
  if (state->seed1 == state->seed2) {
    state->seed1 += 1;
    state->seed2 += 1;
  }
  itemp = (itemp>>8)|0x00000001;
  temp  = (float)(itemp)*state->float_scale;
  return( temp );
}
/*in*******************************************************************
```

Function `vsip_randu_d`:

1. Accepts a `vsip_randstate*` argument (which was created by `vsip_randcreate`);
2. Updates the state seeds using macro functions RAN0 and RAN1;
3. Forms a temporary 32 bit unsigned seed from the difference between the two state seeds;
4. Bitwise OR's a one into the low bit; converts the integer contained in the interval $(0,2^{32}-1)$ to a double precision IEEE-754 floating point number;
5. Scales the floating point number by 2^{-32} to force the result into the open interval (0.0,1.0);
6. And finally returns the floating point result as the function return value.

```
**out*****************************************************************/
double vsip_randu_d( vsip_randstate *state )
{
  double   temp;
  uint32_t itemp;
  state->seed0 = RAN0( state );
  state->seed1 = RAN1( state );
  itemp  = (state->seed0 - state->seed1);
  if (state->seed1 == state->seed2) {
    state->seed1 += 1;
    state->seed2 += 1;
  }
  temp = ((double)(itemp) + 0.5)*state->double_scale;
  return( temp );
}
```

Random Number Functions

vsip_**randcreate**	Create Random State
vsip_**randdestroy**	Destroy Random State
vsip_*ds***randu**_*p*	Uniform Random Numbers
vsip_*ds***randn**_*p*	Gaussian Random Numbers

vsip_randcreate Create Random State

Create a random number generator state object.

Functionality

Creates a state object for use by a VSIPL random number generation function. The random number generator is characterized by specifying the number of random number generators the application is expected to create, and the index of this generator. If the portable sequence is specified, than the number of random number generators specifies how many sub-sequences the primary sequence is partitioned into. If the non-portable sequence is specified, the characteristics of the random number generator are implementation dependent.

The function returns a random state object which holds the state information for the random number sequence generator, or *null* if the create fails.

Prototype

```
vsip_randstate  *vsip_randcreate (
                     vsip_index seed,
                     vsip_index numprocs,
                     vsip_index id,
                     vsip_rng portable);
```

Arguments

`seed` Seed to initialize generator.

`numprocs` Number of processors (number of sub-sequences sequences into which the primary sequence is to be partitioned).

`id` Processor ID (index to select a particular sub-sequence from the group of *numprocs* sub-sequences).

`portable` Select between portable and non-portable random number sequences.

```
typedef enum {
  VSIP_PRNG = 0,          Portable random number generator
  VSIP_NPRNG = 1          Non-portable random number generator
} vsip_rng;
```

Return Value

Returns a pointer to a random number state object of type `vsip_randstate`, or *null* if the create fails.

Restrictions

Errors

The arguments must conform to the following:

1. $0 < id \leq numprocs \leq 2^{31} - 1$

Notes/References

You must call `vsip_randcreate` for each random number sequence/stream the application needs. This might be one per processor, one per thread, etc. For the portable sequence to have the desired pseudo-random properties, each create must specify the same number of processors/sub-sequences.

Note to Implementors: All implementations of `vsip_randcreate` must support the portable sequence. The vendor defined non-portable sequence may be the same sequence as the defined portable sequence, or an implementation dependent uniform random number generator.

Examples

See `vsip_`**`dsrandu`**`_`*`p`* for example.

See Also

`vsip_`*`ds`*`randn_`*`p`*, `vsip_`*`ds`*`randu_`*`p`*, and `vsip_randdestroy_`*`p`*

vsip_randdestroy Destroy Random State

Destroy a random number generator state object.

Functionality

Destroys a random number state object created by a `vsip_randcreate`.

Prototype

```
int  vsip_randdestroy (
                    vsip_randstate *state);
```

Arguments

`*state`	Pointer to random number state object.

Return Value

Returns zero on success and non-zero on failure.

Restrictions

Errors

The arguments must conform to the following:

1. The random number state object must be valid. An argument of *null* is not an error.

Notes/References

An argument of *null* is not an error.

Examples

See `vsip_dsrandu_`*p* for example.

See Also

`vsip_randcreate`, `vsip_`*ds*`randu_`*p*, and `vsip_`*ds*`randn_`*p*

vsip_*ds*randu_*p* Uniform Random Numbers

Generate a uniformly distributed (pseudo-)random number. Floating point values are uniformly distributed over the open interval (0,1). Integer deviates are uniformly distributed over the open interval $(0,2^{31}-1)$.

Functionality

Real

$r \leftarrow \text{Uniform}(0, 1)$

$r_n \leftarrow \text{Uniform}(0, 1)$ for $n = 0, 1, \ldots, N-1$

$r_{n,m} \leftarrow \text{Uniform}(0, 1)$ for $n = 0, 1, \ldots, N-1$; for $m = 0, 1, \ldots, M-1$

Complex

$r \leftarrow \text{Uniform}(0, 1) + j\cdot\text{Uniform}(0, 1)$

$r_n \leftarrow \text{Uniform}(0, 1) + j\cdot\text{Uniform}(0, 1)$ for $n = 0, 1, \ldots, N-1$

$r_{n,m} \leftarrow \text{Uniform}(0, 1) + j\cdot\text{Uniform}(0, 1)$ for $n = 0, 1, \ldots, N-1$; for $m = 0, 1, \ldots, M-1$

Prototype

Scalar:

```
vsip_scalar_f  vsip_randu_f(
                     vsip_randstate *state);
vsip_cscalar_f  vsip_crandu_f(
                     vsip_randstate *state);
```

By Element:

```
void vsip_vrandu_f(
                     vsip_randstate *state,
                     const vsip_vview_f *r);
void vsip_cvrandu_f(
                     vsip_randstate *state,
                     const vsip_cvview_f *r);
void vsip_mrandu_f(
                     vsip_randstate *state,
                     const vsip_mview_f *r);
void vsip_cmrandu_f(
                     vsip_randstate *state,
                     const vsip_cmview_f *r);
```

Arguments

`*state`	Pointer to random number state object.
`*r`	Output vector or matrix view object.

Return Value

The arguments must conform to the following:

1. The random number state object must be valid.
2. The output view object must be valid.

Restrictions

Errors

The pointer to a random number state object must be valid.

Notes/References

The complex random number has real and imaginary components where each component is Uniform(0,1). The mean of the complex sequence is therefore $\sqrt{2}/2$.

The "by element" random number generators are required to produce the equivalent result to applying the scalar generators to each element in the order that proceeds from the minimum stride to the maximum stride dimension. For example for a matrix Z, where the stride between elements of a row is less than the stride between elements of a column, where x_j denotes the jth output of the generator:

$z_{0,0},\leftarrow x_i,\quad z_{0,1},,\leftarrow x_{i+1},\quad z_{0,2},,\leftarrow x_{i+2},\ \ldots,\ z_{1,0},,\leftarrow x_{i+N},\ z_{1,1},,\leftarrow x_{i+N+1},\ \ldots,\ z_{M-1,N-1},,\leftarrow x_{i+M\cdot N-1}$

Examples

Generate 10 Uniform random numbers in the interval -π to π.

```
#include <stdio.h>
#include <vsip.h>

main(){
  int i;
  int seed =0, num_procs=1, id=1;
  vsip_scalar_d   x;
  vsip_cscalar_d  z;
  vsip_rand_state *state;

  vsip_init((void *)0);

  state = vsip_randcreate(seed, num_procs, id, VSIP_PRNG);

  printf("Uniform\n");
  for(i=0; i<10; i++) {
    x = 2*M_PI*vsip_randu_d(state) - M_PI;
    printf("%g\n",x);
  }
  printf("Complex Uniform\n");
  for(i=0; i<10; i++) {
    vsip_rcmul_d(M_PI, vsip_crandu_d(state), &z);
    vsip_rcadd_d(-M_PI, z, &z);
    printf("(%f, %f)\n",vsip_real_d(z),vsip_imag_d(z));
  }
  vsip_randdestroy(state);
  vsip_finalize((void *)0);
  return 0;
}
```

See Also

`vsip_randcreate`, `vsip_`***ds***`randn_`***p***, and `vsip_`***ds***`randdestroy`

vsip_*ds*randn_*p* Gaussian Random Numbers

Generate an approximately normally distributed (pseudo-)random deviate having mean zero and unit variance; N(0,1).

Functionality

Real

$$r \leftarrow 6 - \sum_{k=0}^{11} x_k$$

$$r_n \leftarrow 6 - \sum_{k=0}^{11} x_{k+12n} \qquad \text{for } n = 0, 1, \ldots, N-1$$

$$r_{n,m} \leftarrow 6 - \sum_{k=0}^{11} x_{k+12(n+mN)} \qquad \text{for } n = 0, 1, \ldots, N\text{-}1;\ \text{for } m = 0, 1, \ldots, M-1 \text{ (row major)}$$

$$r_{n,m} \leftarrow 6 - \sum_{k=0}^{11} x_{k+12(m+nM)} \qquad \text{for } n = 0, 1, \ldots, N\text{-}1;\ \text{for } m = 0, 1, \ldots, M-1 \text{ (column. major)}$$

Complex

$$t_1 \leftarrow \sum_{k=0}^{2} x_k, \quad t_2 \leftarrow \sum_{k=0}^{2} x_{k+3}$$
$$r \leftarrow 3 - (t_1 + t_2) + j \cdot (t_1 - t_2)$$

$$t_1 \leftarrow \sum_{k=0}^{2} x_{k+6n}, \quad t_2 \leftarrow \sum_{k=0}^{2} x_{k+6n+3} \qquad \text{for } n = 0, 1, \ldots, N-1$$
$$r_n \leftarrow 3 - (t_1 + t_2) + j \cdot (t_1 - t_2)$$

$$t_1 \leftarrow \sum_{k=0}^{2} x_{k+6(n+mN)}, \quad t_2 \leftarrow \sum_{k=0}^{2} x_{k+6(n+mN)+3} \qquad \text{for } n = 0, 1, \ldots, N\text{-}1;\ \text{for } m = 0, 1, \ldots, M-1$$
$$r_{n,m} \leftarrow 3 - (t_1 + t_2) + j \cdot (t_1 - t_2) \qquad \text{(row major)}$$

$$t_1 \leftarrow \sum_{k=0}^{2} x_{k+6(m+nM)}, \quad t_2 \leftarrow \sum_{k=0}^{2} x_{k+6(m+nM)+3} \qquad \text{for } n = 0, 1, \ldots, N\text{-}1;\ \text{for } m = 0, 1, \ldots, M-1$$
$$r_{n,m} \leftarrow 3 - (t_1 + t_2) + j \cdot (t_1 - t_2) \qquad \text{(column major)}$$

Where:

x_k is a uniformly distributed random number over the open interval (0,1). x_k is generated in order using the same method as the uniform scalar function `vsip_randu_f`. For matrices the dimension with the smallest stride is filled first.

Prototype

Scalar:

```
vsip_scalar_f  vsip_randn_f(
                    vsip_randstate *state);
vsip_cscalar_f  vsip_crandn_f(
                    vsip_randstate *state);
```

By Element:

```
void vsip_vrandn_f(
                    vsip_randstate *state,
                    const vsip_vview_f *r);
void vsip_cvrandn_f(
                    vsip_randstate *state,
                    const vsip_cvview_f *r);
void vsip_mrandn_f(
                    vsip_randstate *state,
                    const vsip_mview_f *r);
void vsip_cmrandn_f(
                    vsip_randstate *state,
                    const vsip_cmview_f *r);
```

Arguments

`*state`	Pointer to random number state object.
`*r`	Output vector or matrix view object.

Return Value

Returns a Gaussian random number.

Restrictions

Errors

The arguments must conform to the following:

1. The random number state object must be valid.
2. The output view object must be valid.

Notes/References

Both the real and complex Gaussian random number are N(0,1). The complex random number has real and imaginary components that are uncorrelated.

If a true Gaussian random deviate is needed, the Box-Muller algorithm should be used. See Knuth, Donald E., *Seminumerical Algorithms*, 2nd ed., vol. 2, pp. 117 of *The Art of Computer Programming*, Addison-Wesley, 1981.

Examples

Generate 10 Uniform Gaussian numbers with zero mean and a variance of π.

```
#include <stdio.h>
#include <vsip.h>

main(){
  int i;
  int seed =0, num_procs=1, id=1;
  vsip_scalar_d   x;
  vsip_cscalar_d  z;
  vsip_rand_state *state;

  vsip_init((void *)0);
  state = vsip_randcreate(seed, num_procs, id, VSIP_PRNG);
  printf("Normal\n");
  for(i=0; i<10; i++) {
    x = M_PI*vsip_randn_d(state);
    printf("%g\n",x);
  }

  printf("Complex Normal\n");
  for(i=0; i<10; i++) {
    vsip_rcmul_d(M_PI, vsip_crandn_d(state), &z);
    printf("(%f, %f)\n",vsip_real_d(z),vsip_imag_d(z));
  }

  vsip_randdestroy(state);
  vsip_finalize((void *)0);
  return 0;
}
```

See Also

`vsip_randcreate`, `vsip_`***ds***`randu_`***p***, and `vsip_`***ds***`randdestroy`

VECTOR & ELEMENTWISE OPERATIONS

Vector & Elementwise - Introduction

This section covers, for the most part, simple operations that are done on individual elements or corresponding pairs of elements of VSIPL objects. Historically this section started out as only pertaining to vector objects; however it was soon realized that, for instance, adding two vectors by element and adding two matrices by element, or taking a Sine or Cosine for each element of a vector or a matrix, are fundamentally the same. So the section was extended to cover as many elementwise operations as seemed reasonable without regard to the shape of the underlying object.

Name Space

An attempt has been made to regularize the name space. For the root names the following contractions have been used. Not all the contractions used are here, but these are the most common

add	a *or* add
divide	div
equal	e
greater than	g *or* gt
less than	l *or* lt
logical	l
magnitude	mg *or* mag
maximum	max
minimum	min
multiply	m *or* mul
not equal	ne
root	rt
scalar	s
square	sq
subtract	sb *or* sub
value	val

NOTE:

- Magnitude has its normal meaning. For example for real we have
  $a \Leftrightarrow |a| = \sqrt{a^2}$
  and for Complex we have
  $a \Leftrightarrow |a| = \sqrt{(\mathrm{Re}(a))^2 + (\mathrm{Im}(a))^2}$
  where the positive square root is used.
- Most functions which return a value and are not void will have a "val" contraction at the end of their root name. One of the exceptions are any functions which are typed as boolean.

Root Names

The following table contains all the root names for the elementwise operations. Using these roots, and the function name encoder, names may be derived for all the functions. The root names are in alphabetical order. Note that the root name may encompass some functionality that might otherwise be done using the function name encoder rules. For instance, the root *arg* used for finding the argument of a complex number has no depth associated with it since its classical definition encompasses the depth functionality. Some functions have purposely had some of the function name made part of the root name. For instance, *cmagsq* is only done on complex, so the *c* was made part of the root name. Moving function name encoder functionality into the root name can only be done if the functionality is degenerate over the name space of the function. For instance, *cmagsq* is only defined in VSIPL for *complex* inputs so the complex portion of the functionality may be made part of the root name since there is no need to include *real* input functionality.

Note that some root names (such as *add*, *mul*, etc.) have more than one man page associated with them. This is because Scalar operations with Vectors (Matrices) are placed on a separate man page from Vector/matrix operations with Vectors (Matrices).

It also should be noted that not every possible expansion of the function name, with root, are being included as VSIPL defined functions. This is a very large name space so only functionality which is considered as necessary and useful is being included. What is necessary and useful is an active area of debate and discussion.

Root Name	Expansion	Associated Man Pages and Comments
acos	Arccosine	`vsip_`***ds***`acos_`***p***
add	Add	`vsip_`***ds***`add_`***p*** (Vector/Matrix *add* Vector/Matrix) `vsip_`***ds***`sadd_`***p*** (Scalar *add* Vector/Matrix)
alltrue	All True	`vsip_`***s***`alltrue_bl`
am	Add and Multiply	`vsip_`***d***`vam_`***p***
and	AND	`vsip_`***s***`and_`***p***
anytrue	Any True	`vsip_`***s***`anytrue_bl`
arg	Argument	`vsip_`***s***`arg_`***p***
asin	Arcsine	`vsip_`***ds***`asin_`***p***
atan	Arctangent	`vsip_`***ds***`atan_`***p***
atan2	Arctangent of Two Arguments	`vsip_`***s***`atan2_`***p***
conj	Complex Conjugate	`vsip_`***s***`conj_`***p***
clip	Clip	`vsip_`***s***`clip_`***p***
cmagsq	Complex Magnitude Squared	`vsip_`***s***`cmagsq_`***p***
cmaxmgsq	Complex Max Magnitude Squared	`vsip_`***s***`cmaxmgsq_`***p***

Root Name	Expansion	Associated Man Pages and Comments
cmaxmgsqval	Complex Max Mag Squared Value	vsip_*s*cmaxmgsqval_*p*
cminmgsq	Complex Min Magnitude Squared	vsip_*s*cminmgsq_*p*
cminmgsqval	Complex Min Mag Squared Value	vsip_*s*cminmgsqval_*p*
cmplx	Complex	vsip_*s*cmplx_*p*
cos	Cosine	vsip_*ds*cos_*p*
cosh	Hyperbolic Cosine	vsip_*ds*cosh_*p*
div	Divide	vsip_*ds*div_*p* (Vector/Matrix *divide* Vector/Matrix) vsip_*d*s*s*div_*p* (Scalar *divide* Vector/Matrix) vsip_*ds*sdiv_*p* (Vector/Matrix *divide* Scalar)
euler	Euler	vsip_*s*euler_*p* (Vector/Matrix Euler)
exp	Exponential (Base e)	vsip_*ds*exp_*p*
exp10	Exponential Base 10	vsip_*ds*exp10_*p*
expoavg	Exponential Average	vsip_*ds*expoavg_*p*
fill	Fill	vsip_*ds*fill_*p*
gather	Gather	vsip_*ds*gather
hypot	Hypotenuse	vsip_*s*hypot_*p*
imag	Imaginary	vsip_*s*imag_*p*
indexbool	Index a Boolean	vsip_*s*indexbool
invclip	Inverse Clip	vsip_*s*invclip_*p*
jmul	Conjugate Multiply	vsip_c*s*jmul_*p*
leq	Logical Equal	vsip_*s*leq_*p*
lge	Logical Greater Than or Equal	vsip_*s*lge_*p*
lgt	Logical Greater Than	vsip_*s*lgt_*p*
lle	Logical Less Than or Equal	vsip_*s*lle_*p*
llt	Logical Less Than	vsip_*s*llt_*p*
lne	Logical Not Equal	vsip_*s*lne_*p*
log	Log (Base e)	vsip_*ds*log_*p*
log10	Log Base Ten	vsip_*ds*log10_*p*
ma	Multiply and Add	vsip_*d*vma_*p*
mag	Magnitude	vsip_*ds*mag_*p*

Root Name	Expansion	Associated Man Pages and Comments
max	Maximum	vsip_*s*max_*p*
maxmg	Maximum Magnitude	vsip_*ds*maxmg_*p*
maxmgval	Maximum Magnitude Value	vsip_*s*maxmgval_*p*
maxval	Maximum Value	vsip_*s*maxval_*p*
meansqval	Mean Square Value	vsip_*ds*meansqval_*p*
meanval	Mean Value	vsip_*ds*meanval_*p*
min	Minimum	vsip_*s*min_*p*
minmg	Minimum Magnitude	vsip_*ds*minmg_*p*
minmgval	Minimum Magnitude Value	vsip_*s*minmgval_*p*
minval	Minimum Value	vsip_*s*minval_*p*
msa	Multiply, Scalar Add	vsip_*d*vmsa_*p*
msb	Multiply and Subtract	vsip_*d*vmsb_*p*
mul	Multiply	vsip_*ds*mul_*p* (Vector/Matrix *multiply* Vector/Matrix) vsip_*d*s*s*mul_*p* (Scalar *multiply* Vector/Matrix)
vmmul	Vector Matrix Elementwise Multiply	vsip_*d*v*d*mmul_*p*
neg	Negate	vsip_*ds*neg_*p*
not	NOT	vsip_*s*not_*p*
or	OR	vsip_*s*or_*p*
polar	Polar	vsip_*s*polar_*p*
ramp	Ramp	vsip_vramp_*p*
real	Real	vsip_*s*real_*p*
recip	Reciprocal	vsip_*ds*recip_*p*
rect	Rectangular	vsip_*s*rect_*p*
rsqrt	Reciprocal Square Root	vsip_*ds*rsqrt_*p*
sam	Scalar Add, Multiply	vsip_*d*vsam_*p*
sbm	Subtract and Multiply	vsip_*d*vsbm_*p*
scatter	Scatter	vsip_*ds*scatter
sin	Sine	vsip_*ds*sin_*p*
sinh	Hyperbolic Sine	vsip_*ds*sinh_*p*
sma	Scalar Multiply, Add	vsip_*d*vsma_*p*

Root Name	Expansion	Associated Man Pages and Comments
smsa	Scalar Multiply, Scalar Add	`vsip_dvsmsa_p`
sq	Square	`vsip_dssq_p`
sqrt	Square Root	`vsip_dssqrt_p`
sub	Subtract	`vsip_dssub_p` (Vector/Matrix *subtract* Vector/Matrix) `vsip_dssub_p` (Scalar *subtract* Vector/Matrix)
sumsqval	Sum of Squares Value	`vsip_dssumsqval_p`
sumval	Sum Value	`vsip_dssumval_p`
swap	Swap	`vsip_dsswap_p`
tan	Tangent	`vsip_dstan_p`
tanh	Hyperbolic Tangent	`vsip_dstanh_p`
vmodulate	Vector Modulate	`vsip_vmodulate_p`
xor	Exclusive OR	`vsip_sxor_p`

In-Place Functionality

Most simple elementwise functions may be done in-place. The meaning of in-place is not necessarily always clear. The following rules will define in-place for the Vector and Elementwise functions.

1. The sign (negative or positive) of the stride of objects used for in place must be the same.
2. For functions, like **cos,** of a single argument with a single result of the same type and precision, in-place means the result replaces the input.
3. For functions, like **add,** of multiple arguments with a single result of the same type and precision, in-place means the result replaces one of the inputs.
4. For functions**,** of one or more vector/matrix/tensor arguments and one or more scalar, with a single result of the same type and precision, in-place means the result replaces one of the inputs.
5. For a function, like **arg,** which takes a complex input and outputs a real output of the same precision in-place means the output can be placed in a real view or an imaginary view of the input.
6. For **polar** which takes a complex input and outputs two real outputs in-place means one output may be placed in a real view of the input and the other output may be placed in the imaginary view of the input. Note either output can go in either view.
7. For **rect** which takes two real inputs and gives a complex output in-place means the inputs can be views of the real and imaginary portion of the output. Note either input can go in either view.
8. For **euler** which takes a single real input and produces a complex output then in-place means the real input can be either a real view or imaginary view of the output.

It should be noted that for functions where no input is conformant with any output then no in-place operation is defined. In-place operations which transform real views to complex views or complex views to real views are only allowed when the real view is an imaginary view or a

real view of the complex view. This means, for instance, that a view of a block of data which overlays a complex views data so that both real and imaginary words of the complex view are incorporated in the real view are not to be used for in-place operations. It also means that the input or output real view must exactly overlay the real or imaginary view of the complex view, and the stride through the vectors (real and imaginary) must both go in the same direction.

The application programmer should only use in-place functionality for functions required to support this functionality by the specification. If an implementation provides an in-place functionality that is not required by the VSIPL specification then the application programmer should not use it. Using non-required functionality will result in portability problems.

The following table indicates required in-place functionality for vector and elementwise functions.

Root Name	In-Place?	Reason/Comment
acos	yes	
add	yes	
alltrue	NA	Returns scalar value
am	yes	
and	yes	
anytrue	NA	Returns scalar value
arg	yes	For in-place output must be into a real view or imaginary view. Output to a real view encompassing both real and imaginary words of the input vector is not allowed.
asin	yes	
atan	yes	
atan2	yes	
conj	yes	
clip	yes	
cmagsq	yes	
cmaxmgsq	yes	
cmaxmgsqval	NA	Returns scalar value
cminmgsq	yes	
cminmgsqval	NA	Returns scalar value
cmplx	no	
cos	yes	
cosh	yes	
div	yes	
euler	yes	In-place input must be a real or imaginary view of the output vector.
exp	yes	

Root Name	In-Place?	Reason/Comment
exp10	yes	
expoavg	no	
fill	NA	No source
gather	no	Not an elementwise operation
hypot	yes	
imag	yes	Harmless if output is an imaginary view of the input vector, destructive if output is a real view. Output to a compacted real view containing both real and imaginary words from the complex input view is not allowed.
indexbool	no	Different source and destination types
invclip	yes	
jmul	yes	
leq	no	Different source and destination types
lge	no	Different source and destination types
lgt	no	Different source and destination types
lle	no	Different source and destination types
llt	no	Different source and destination types
lne	no	Different source and destination types
log	yes	
log10	yes	
ma	yes	
mag	yes	
max	yes	
maxmg	yes	
maxmgval	NA	Returns scalar value
maxval	NA	Returns scalar value
meansqval	NA	Returns scalar value
meanval	NA	Returns scalar value
min	yes	
minmg	yes	
minmgval	NA	Returns scalar value
minval	NA	Returns scalar value
msa	yes	
msb	yes	

Root Name	In-Place?	Reason/Comment
mul	yes	
neg.	yes	
not	yes	
or	yes	
polar	yes	
ramp	NA	No source
randcreate	NA	Not a function on views
randg	NA	No source
randu	NA	No source
real	yes	Harmless if output is a real view of the input, destructive if output is imaginary view. Output to a compacted real view containing both real and imaginary words from the complex input view is not allowed.
recip	yes	
rect	yes	
rsqrt	yes	
sam	yes	
sbm	yes	
scatter	no	Not an elementwise operation
sin	yes	
sinh	yes	
sma	yes	
smsa	yes	
sq	yes	
sqrt	yes	
sub	yes	
sumsqval	NA	Returns scalar value
sumval	NA	Returns scalar value
swap	Harmless	
tan	yes	
tanh	yes	
vmodulate	no	
xor	yes	

Example Programs

Many of the examples in this chapter are designed to be run-able, although no guarantees are made. Note that all code requires including the standard VSIPL header file, `"vsip.h"`.

Note that for brevity, the examples don't follow a careful programming style. For instance when doing a *create,* the return should always be checked for a *null* pointer to determine if the create failed.

Elementary Math Functions

The following functions constitute by element application of elementary math operations on vectors and matrices. These include trigonometric functions, natural (base *e*) and base 10 logarithmic and exponential functions, and the square root function. These functions are defined for real floats with the exception of base *e* logarithm, the base *e* exponential, and the square root which are also defined for complex floats.

vsip_***S*acos_*p***	Vector/Matrix Arccosine
vsip_***S*asin_*p***	Vector/Matrix Arcsine
vsip_***S*atan_*p***	Vector/Matrix Arctangent
vsip_***S*atan2_*p***	Vector/Matrix Arctangent of Two Arguments
vsip_***S*cos_*p***	Vector/Matrix Cosine
vsip_***S*cosh_*p***	Vector/Matrix Hyperbolic Cosine
vsip_***dS*exp_*p***	Vector/Matrix Exponential
vsip_***S*exp10_*p***	Vector/Matrix Exponential Base 10
vsip_***dS*log_*p***	Vector/Matrix Log
vsip_***S*log10_*p***	Vector/Matrix Log Base 10
vsip_***S*sin_*p***	Vector/Matrix Sine
vsip_***S*sinh_*p***	Vector/Matrix Hyperbolic Sine
vsip_***dS*sqrt_*p***	Vector/Matrix Square Root
vsip_***S*tan_*p***	Vector/Matrix Tangent
vsip_***S*tanh_*p***	Vector/Matrix Hyperbolic Tangent

vsip_*sacos*_*p* Vector/Matrix Arccosine

Computes the principal radian value [0, π] of the arccosine for each element of a vector/matrix.

Functionality

$r_j \leftarrow \cos^{-1} a_j$ for $j = 0, 1, \ldots, N\text{-}1$

$r_{i,j} \leftarrow \cos^{-1} a_{i,j}$ for $i = 0, 1, \ldots, M\text{-}1$; for $j = 0, 1, \ldots, N\text{-}1$

Prototypes

```
void vsip_vacos_f(
                    const vsip_vview_f *a,
                    const vsip_vview_f *r);
void vsip_macos_f(
                    const vsip_mview_f *a,
                    const vsip_mview_f *r);
```

Arguments

`*a`	View of input vector/matrix
`*r`	View of output vector/matrix

Return Value

None.

Restrictions

Element values outside the interval [-1, 1] are a domain error. Results of domain errors are implementation dependent.

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.
3. The input and output views must be identical views of the same block (in-place), or must not overlap.

Notes/References

Examples

```
/* calculate arccos for seven evenly spaced values from -1 to 1 */
#include<stdio.h>
#include "vsip.h"

#define L 7          /* length */

int main()
{
    double data[L];  /*a user defined data space */
    int i;
    vsip_vview_d* ramp;
    vsip_vview_d* output;

    vsip_init((void *)0);
    ramp = vsip_vcreate_d(L,VSIP_MEM_NONE);
    output = vsip_vbind_d(vsip_blockbind_d(data, L, VSIP_MEM_NONE),0,1,L);
```

```
    /* admit the user block with no update */
    vsip_blockadmit_d(vsip_vgetblock_d(output),VSIP_FALSE);

    /*compute a ramp from -1 to 1 */
    vsip_vramp_d(-1.0, 2.0/(L-1) , ramp);

    /*compute the Arccosine value */
    vsip_vacos_d(ramp, output);

    /* release the user block with update */
    vsip_blockrelease_d(vsip_vgetblock_d(output),VSIP_TRUE);

    /*print it */
    for(i=0; i<L; i++)
        printf("%f ",data[i]);
    printf("\n");

    /*destroy the vector views and any associated blocks */
    vsip_blockdestroy_d(vsip_vdestroy_d(ramp));
    vsip_blockdestroy_d(vsip_vdestroy_d(output));
    vsip_finalize((void *)0);
    return 0;
}
/* output */
/* 3.141593 2.300524 1.910633 1.570796 1.230959 0.841069 0.000000 */
```

See Also

vsip_*s*asin_*p*, vsip_*s*atan_*p*, vsip_*s*atan2_*p*, vsip_*s*cos_*p*, vsip_*s*sin_*p*, and vsip_*s*tan_*p*

vsip_sasin_p Vector/Matrix Arcsine

Computes the principal radian value [-π/2, π/2] of the arcsine for each element of a vector/matrix.

Functionality

$r_j \leftarrow \sin^{-1} a_j$ for $j = 0, 1, \ldots, N\text{-}1$

$r_{i,j} \leftarrow \sin^{-1} a_{i,j}$ for $i = 0, 1, \ldots, M\text{-}1$; for $j = 0, 1, \ldots, N\text{-}1$

Prototypes

```
void vsip_vasin_f(
                    const vsip_vview_f *a,
                    const vsip_vview_f *r);
void vsip_masin_f(
                    const vsip_mview_f *a,
                    const vsip_mview_f *r);
```

Arguments

`*a`	View of input vector/matrix
`*r`	View of output vector/matrix

Return Value

None.

Restrictions

Element values outside the interval [-1, 1] are a domain error. Results of domain errors are implementation dependent.

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.
3. The input and output views must be identical views of the same block (in-place), or must not overlap.

Notes/References

Examples

```
/* calculate Arcsine for seven evenly spaced values from -1 to 1 */
#include<stdio.h>
#include "vsip.h"

#define L 7          /* length */

int main()
{
    double data[L];  /*a user created data space */
    int i;
    vsip_vview_d* ramp;
    vsip_vview_d* output;

    vsip_init((void *)0);
    ramp = vsip_vcreate_d(L,VSIP_MEM_NONE);
    output = vsip_vbind_d(vsip_blockbind_d(data,L,VSIP_MEM_NONE),0,1,L);
```

```
    /* admit the user block with no update */
    vsip_blockadmit_d(vsip_vgetblock_d(output),VSIP_FALSE);

    /*compute a ramp from -1 to 1 */
    vsip_vramp_d(-1.0, 2.0/(L-1) , ramp);

    /*compute the Arcsine value */
    vsip_vasin_d(ramp, output);

    /* release the user block with update */
    vsip_blockrelease_d(vsip_vgetblock_d(output),VSIP_TRUE);
    /*print it */
    for(i=0; i<L; i++)
        printf("%f ",data[i]);
    printf("\n");

    /*destroy the vector views and any associated blocks */
    vsip_blockdestroy_d(vsip_vdestroy_d(ramp));
    vsip_blockdestroy_d(vsip_vdestroy_d(output));
    vsip_finalize((void *)0);
    return 0;
}
/* output */
/* -1.570796 -0.729728 -0.339837 0.000000 0.339837 0.729728 1.570796 */
```

See Also

vsip_*s*acos_*p*, vsip_*s*atan_*p*, vsip_*s*atan2_*p*, vsip_*s*cos_*p*, vsip_*s*sin_*p*, and vsip_*s*tan_*p*

vsip_satan_p Vector/Matrix Arctangent

Computes the principal radian value,[-π/2, π/2] of the arctangent for each element of a vector/matrix.

Functionality

$r_j \leftarrow \tan^{-1} a_j$ for $j = 0, 1, \ldots, N\text{-}1$

$r_{i,j} \leftarrow \tan^{-1} a_{i,j}$ for $i = 0, 1, \ldots, M\text{-}1$; for $j = 0, 1, \ldots, N\text{-}1$

Prototypes

```
void vsip_vatan_f(
                    const vsip_vview_f *a,
                    const vsip_vview_f *r);
void vsip_matan_f(
                    const vsip_mview_f *a,
                    const vsip_mview_f *r);
```

Arguments

`*a`	View of input vector/matrix
`*r`	View of output vector/matrix

Return Value

None.

Restrictions

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.
3. The input and output views must be identical views of the same block (in-place), or must not overlap.

Notes/References

Examples

```
/* calculate arg = atan(y/x) in each quadrant at the midpoint and on each
axis */
#include<stdio.h>
#include "vsip.h"

#define PI 3.1415926535
#define N  8                    /* length */

int main()
{
    double data[N];             /*a user created data space */
    int i;
    vsip_vview_d* arg;
    vsip_vview_d* x;
    vsip_vview_d* y;

    vsip_init((void *)0);
    arg = vsip_vbind_d(vsip_blockbind_d(data, N, VSIP_MEM_NONE),0,1,N);
```

```
    x   = vsip_vcreate_d(N, VSIP_MEM_NONE);
    y   = vsip_vcreate_d(N, VSIP_MEM_NONE);

    vsip_vramp_d(0.0,2 * PI/N, y);
    vsip_vcos_d(y, x);
    vsip_vsin_d(y, y);

    /*In the next step we assume that x values may be small
      but will not be zero exactly */
    vsip_vdiv_d(y, x, x);

    /* admit the user block with no update */
    vsip_blockadmit_d(vsip_vgetblock_d(arg),VSIP_FALSE);

    /*compute the Arctangent value */
    vsip_vatan_d(x, arg);

    /* release the user block with update */
    vsip_blockrelease_d(vsip_vgetblock_d(arg),VSIP_TRUE);

    /*print it */
    for(i=0; i<N; i++)
      printf("%f ",data[i]);
    printf("\n");

    /*destroy the vector views and any associated blocks */
    vsip_blockdestroy_d(vsip_vdestroy_d(y));
    vsip_blockdestroy_d(vsip_vdestroy_d(x));
    vsip_blockdestroy_d(vsip_vdestroy_d(arg));
    vsip_finalize((void *)0);
    return 0;
}
/* output */
/* 0.000000 0.785398 1.570796 -0.785398 -0.000000 0.785398 1.570796 -0.785398
*/
```

See Also

vsip_*s*acos_*p*, vsip_*s*asin_*p*, vsip_*s*atan2_*p*, vsip_*s*cos_*p*, vsip_*s*sin_*p*, and vsip_*s*tan_*p*

vsip_satan2_p Vector/Matrix Arctangent of Two Arguments

Computes the four quadrant radian value [-π, π] of the arctangent of the ratio of the elements of two input vectors/matrices.

Functionality

$$r_j \leftarrow \tan^{-1}\frac{a_j}{b_j} \qquad \text{for } j = 0, 1, \ldots, N\text{-}1$$

$$r_{i,j} \leftarrow \tan^{-1}\frac{a_{i,j}}{b_{i,j}} \qquad \text{for } i = 0, 1, \ldots, M\text{-}1;\ \text{for } j = 0, 1, \ldots, N\text{-}1$$

The rules for calculating `vsip_satan2_p` are the same as for the ANSI C math function `atan2`. The following table may be used to calculate atan2, although other methods may also be used.

If	Then
$a > 0;\quad b \neq 0$	$\tan^{-1}\frac{a}{b} \equiv \cos^{-1}\left(b \Big/ \sqrt{a^2+b^2}\right)$
$a < 0;\quad b < 0$	$\tan^{-1}\frac{a}{b} \equiv \cos^{-1}\left(-b \Big/ \sqrt{a^2+b^2}\right) - \pi$
$a < 0;\quad b > 0$	$\tan^{-1}\frac{a}{b} \equiv -\cos^{-1}\left(b \Big/ \sqrt{a^2+b^2}\right)$
$a > 0;\quad b = 0$	$\tan^{-1}\frac{a}{0} \equiv \pi/2$
$a < 0;\quad b = 0$	$\tan^{-1}\frac{a}{0} \equiv -\pi/2$
$a = 0;\quad b > 0$	$\tan^{-1}\frac{0}{b} \equiv 0$
$a = 0;\quad b < 0$	$\tan^{-1}\frac{0}{b} \equiv \pi$
$a = 0;\quad b = 0$	$\tan^{-1}\frac{0}{0} \equiv$ Undefined or NaN

Note that the use of "π" is not meant to denote an exact number, but it is expected (but not required) to be accurate to the machine precision for the data type.

Prototypes

```
void vsip_vatan2_f(
                    const vsip_vview_f *a,
                    const vsip_vview_f *b,
                    const vsip_vview_f *r);
void vsip_matan2_f(
                    const vsip_mview_f *a,
                    const vsip_mview_f *b,
                    const vsip_mview_f *r);
```

Arguments

`*a` View of input vector/matrix corresponding to numerator

`*b` View of input vector/matrix corresponding to denominator

`*r` View of output vector/matrix

Return Value

None.

Restrictions

The domain of atan2(x, y) is not valid for both x and y zero and the result is implementation dependent.

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.
3. The input and output views must be identical views of the same block (in-place), or must not overlap.

Notes/References

Examples

```
/* calculate arg = atan2(y/x) in each quadrant at the midpoint and on each
axis */
#include<stdio.h>
#include "vsip.h"

#define PI 3.1415926535
#define N  8                    /* length */

int main()
{
    double data[N];             /*a user created data space */
    int i;
    vsip_vview_d* arg;
    vsip_vview_d* x;
    vsip_vview_d* y;

    vsip_init((void *)0);
    arg = vsip_vbind_d(vsip_blockbind_d(data, N, VSIP_MEM_NONE),0,1,N);
    x   = vsip_vcreate_d(N, VSIP_MEM_NONE);
    y   = vsip_vcreate_d(N, VSIP_MEM_NONE);

    vsip_vramp_d(0.0, 2 * PI/N, y);
    vsip_vcos_d(y, x);
    vsip_vsin_d(y, y);

    /* admit the user block with no update */
    vsip_blockadmit_d(vsip_vgetblock_d(arg),VSIP_FALSE);

    /*compute the arctan2 value */
    vsip_vatan2_d(y, x, arg);

    /* release the user block with update */
    vsip_blockrelease_d(vsip_vgetblock_d(arg),VSIP_TRUE);

    /*print it */
    for(i=0; i<N; i++)
```

```
      printf("%f ",data[i]);
    printf("\n");

    /*destroy the vector views and any associated blocks */
    vsip_blockdestroy_d(vsip_vdestroy_d(y));
    vsip_blockdestroy_d(vsip_vdestroy_d(x));
    vsip_blockdestroy_d(vsip_vdestroy_d(arg));
    vsip_finalize((void *)0);
    return 0;
}
/* output */
/* 0.000000 0.785398 1.570796 2.356194 3.141593 -2.356194 -1.570796 -0.785398
*/
```

See Also

vsip_***s***acos_***p***, vsip_***s***asin_***p***, vsip_***s***atan_***p***, vsip_***s***cos_***p***, vsip_***s***sin_***p***, vsip_***s***tan_***p***, and vsip_***s***hypot_***p***

vsip_scos_p Vector/Matrix Cosine

Computes the cosine for each element of a vector/matrix. Element angle values are in radians.

Functionality

$r_j \leftarrow \cos a_j$ for $j = 0, 1, \ldots, N\text{-}1$

$r_{i,j} \leftarrow \cos a_{i,j}$ for $i = 0, 1, \ldots, M\text{-}1$; for $j = 0, 1, \ldots, N\text{-}1$

Prototypes

```
void vsip_vcos_f(
                    const vsip_vview_f *a,
                    const vsip_vview_f *r);
void vsip_mcos_f(
                    const vsip_mview_f *a,
                    const vsip_mview_f *r);
```

Arguments

`*a`	View of input vector/matrix
`*r`	View of output vector/matrix

Return Value

None.

Restrictions

Behavior of element values outside of the closed interval [-2π, 2π] is implementation dependent.

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.
3. The input and output views must be identical views of the same block (in-place), or must not overlap.

Notes/References

Input arguments are expressed in radians.

Examples

```
/* print cosine values for angles evenly spaced between zero and 2 pi  */
#include<stdio.h>
#include "vsip.h"

#define PI 3.1415926535
#define L  7                /* length */

int main()
{
    double data[L];         /* a user created data space */
    int i;
    vsip_vview_d* ramp;
    vsip_vview_d* output;

    vsip_init((void *)0);
    ramp = vsip_vcreate_d(L, VSIP_MEM_NONE);
    output = vsip_vbind_d(vsip_blockbind_d(data,L, VSIP_MEM_NONE),0,1,L);
```

```
    /* compute a ramp from zero to 2 pi */
    vsip_vramp_d(0.0, (2.0 * PI / (double) (L - 1)), ramp);

    /* admit the user block with no update */
    vsip_blockadmit_d(vsip_vgetblock_d(output),VSIP_FALSE);

    /* compute the cosine value */
    vsip_vcos_d(ramp, output);

    /* release the user block with update */
    vsip_blockrelease_d(vsip_vgetblock_d(output),VSIP_TRUE);

    /* print it */
    for(i=0; i<L; i++)
        printf("%f ",data[i]);
    printf("\n");

    /* destroy the vector views and any associated blocks */
    vsip_blockdestroy_d(vsip_vdestroy_d(ramp));
    vsip_blockdestroy_d(vsip_vdestroy_d(output));
    vsip_finalize((void *)0);
    return 0;
}
/* output */
/* 1.000000 0.500000 -0.500000 -1.000000 -0.500000 0.500000 1.000000 */
```

See Also

vsip_*s*acos_*p*, vsip_*s*asin_*p*, vsip_*s*atan_*p*, vsip_*s*atan2_*p*, vsip_*s*sin_*p*, and vsip_*s*tan_*p*

vsip_scosh_*p* Vector/Matrix Hyperbolic Cosine

Computes the hyperbolic cosine for each element of a vector/matrix.

Functionality

$r_j \leftarrow \cosh a_j$ for $j = 0, 1, \ldots, N\text{-}1$

$r_{i,j} \leftarrow \cosh a_{i,j}$ for $i = 0, 1, \ldots, M\text{-}1$; for $j = 0, 1, \ldots, N\text{-}1$

Prototypes

```
void vsip_vcosh_f(
                    const vsip_vview_f *a,
                    const vsip_vview_f *r);
void vsip_mcosh_f(
                    const vsip_mview_f *a,
                    const vsip_mview_f *r);
```

Arguments

`*a`	View of input vector/matrix
`*r`	View of output vector/matrix

Return Value

None.

Restrictions

Overflow behavior and domain restrictions are implementation dependent.

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.
3. The input and output views must be identical views of the same block (in-place), or must not overlap.

Notes/References

Examples

```
/* cosh example */
#include<stdio.h>
#include "vsip.h"

#define  L 7             /* length */

int main()
{
    double data[L];      /*a user created data space   */
    int i;
    vsip_vview_d* ramp;
    vsip_vview_d* output;

    vsip_init((void *)0);
    ramp = vsip_vcreate_d(L, VSIP_MEM_NONE);
    output = vsip_vbind_d(vsip_blockbind_d(data,L, VSIP_MEM_NONE),0,1,L);

    /*compute a ramp from zero to L-1*/
```

```
    vsip_vramp_d(0.0, 1.0, ramp);

    /* admit the user block with no update */
    vsip_blockadmit_d(vsip_vgetblock_d(output),VSIP_FALSE);

    /*compute the hyperbolic cosine value */
    vsip_vcosh_d(ramp, output);

    /* release the user block with update */
    vsip_blockrelease_d(vsip_vgetblock_d(output),VSIP_TRUE);

    /*print it */
    for(i=0; i<L; i++)
        printf("%f ",data[i]);
    printf("\n");

    /*destroy the vector views and any associated blocks  */
    vsip_blockdestroy_d(vsip_vdestroy_d(ramp));
    vsip_blockdestroy_d(vsip_vdestroy_d(output));
    vsip_finalize((void *)0);
    return 0;
}
/* output */
/* 1.000000 1.543081 3.762196 10.067662 27.308233 74.209949 201.715636 */
```

See Also

`vsip_`***s***`sinh_`***p***, and `vsip_`***s***`tanh_`***p***

vsip_*ds*exp_*p* Vector/Matrix Exponential

Computes the exponential function value for each element of a vector/matrix.

Functionality

Real:

$r_j \leftarrow e^{a_j}$ for $j = 0, 1, \ldots, N\text{-}1$

$r_{i,j} \leftarrow e^{a_{i,j}}$ for $i = 0, 1, \ldots, M\text{-}1$; for $j = 0, 1, \ldots, N\text{-}1$

Complex:

Let $\sigma = \mathrm{Re}(a)$, $\omega = \mathrm{Im}(a)$

$r_k \leftarrow e^{\sigma_k} \cdot (\cos\omega_k + j \cdot \sin\omega_k) = e^{\sigma_k + j\varpi_k}$ for $k = 0, 1, \ldots, N\text{-}1$

$r_{k,l} \leftarrow e^{\sigma_{k,l}} \cdot (\cos\omega_{k,l} + j \cdot \sin\omega_{k,l}) = e^{\sigma_{k,l} + j\varpi_{k,l}}$ for $k = 0, 1, \ldots, M\text{-}1$; for $l = 0, 1, \ldots, N\text{-}1$

Prototypes

```
void vsip_vexp_f(
                    const vsip_vview_f *a,
                    const vsip_vview_f *r);
void vsip_cvexp_f(
                    const vsip_cvview_f *a,
                    const vsip_cvview_f *r);
void vsip_mexp_f(
                    const vsip_mview_f *a,
                    const vsip_mview_f *r);
void vsip_cmexp_f(
                    const vsip_cmview_f *a,
                    const vsip_cmview_f *r);
```

Arguments

`*a`	View of input vector/matrix
`*r`	View of output vector/matrix

Return Value

None.

Restrictions

Overflow will occur if a (real part for complex) element is greater than the natural log of the maximum defined number. The result of an overflow is implementation dependent.

Underflow will occur if a (real part for complex) element is less than the negative of the natural log of the maximum defined number. The result of an underflow is implementation dependent.

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.
3. The input and output views must be identical views of the same block (in-place), or must not overlap.

Notes/References

Examples

```
/* calculate a = exp(x) for x = {0,1,...,N} */
#include<stdio.h>
#include "vsip.h"

#define N  4                    /* length */

int main()
{
    double data[N];             /*a user created data space */
    int i;
    vsip_vview_d* a;
    vsip_vview_d* x;

    vsip_init((void *)0);
    a = vsip_vbind_d(vsip_blockbind_d(data,N, VSIP_MEM_NONE),0,1,N);
    x = vsip_vcreate_d(N, VSIP_MEM_NONE);
    vsip_vramp_d(0.0, 1.0, x);

    /* admit the user block with no update */
    vsip_blockadmit_d(vsip_vgetblock_d(a),VSIP_FALSE);

    /*compute the exponential value */
    vsip_vexp_d(x, a);

    /* release the user block with update */
    vsip_blockrelease_d(vsip_vgetblock_d(a),VSIP_TRUE);

    /*print it */
    for(i=0; i<N; i++)
      printf("%f ",data[i]);
    printf("\n");

    /*destroy the vector views and any associated blocks */
    vsip_blockdestroy_d(vsip_vdestroy_d(x));
    vsip_blockdestroy_d(vsip_vdestroy_d(a));
    vsip_finalize((void *)0);
    return 0;
}
/* output */
/* 1.000000 2.718282 7.389056 20.085537 */
```

See Also

`vsip_`***s***`exp10_`***p***, `vsip_`***ds***`log_`***p***, and `vsip_`***s***`log10_`***p***

vsip_sexp10_p Vector/Matrix Exponential Base 10

Computes the base 10 exponential for each element of a vector/matrix.

Functionality

$r_j \leftarrow 10^{a_j}$ for $j = 0, 1, \ldots, N\text{-}1$

$r_{i,j} \leftarrow 10^{a_{i,j}}$ for $i = 0, 1, \ldots, M\text{-}1$; for $j = 0, 1, \ldots, N\text{-}1$

Prototypes

```
void vsip_vexp10_f(
                      const vsip_vview_f *a,
                      const vsip_vview_f *r);
void vsip_mexp10_f(
                      const vsip_mview_f *a,
                      const vsip_mview_f *r);
```

Arguments

`*a`	View of input vector/matrix
`*r`	View of output vector/matrix

Return Value

None.

Restrictions

Overflow will occur if an element is greater than the base ten log of the maximum defined number. The result of an overflow is implementation dependent.

Underflow will occur if an element is less than the negative of the base ten log of the maximum defined number. The result of an underflow is implementation dependent.

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.
3. The input and output views must be identical views of the same block (in-place), or must not overlap.

Notes/References

Examples

```
/* calculate a = 10^x  for x = {0,1,...,N} */
#include<stdio.h>
#include "vsip.h"

#define N  4                    /* length */

int main()
{
    double data[N];             /*a user created data space */
    int i;
    vsip_vview_d* a;
    vsip_vview_d* x;

    vsip_init((void *)0);
```

```
    a = vsip_vbind_d(vsip_blockbind_d(data, N, VSIP_MEM_NONE),0,1,N);
    x = vsip_vcreate_d(N, VSIP_MEM_NONE);
    vsip_vramp_d(0.0, 1.0, x);

    /* admit the user block with no update */
    vsip_blockadmit_d(vsip_vgetblock_d(a),VSIP_FALSE);

    /*compute the 10^x value */
    vsip_vexp10_d(x,  a);

    /* release the user block with update */
    vsip_blockrelease_d(vsip_vgetblock_d(a),VSIP_TRUE);

    /*print it */
    for(i=0; i<N; i++)
      printf("%f ",data[i]);
    printf("\n");

    /*destroy the vector views and any associated blocks */
    vsip_blockdestroy_d(vsip_vdestroy_d(x));
    vsip_blockdestroy_d(vsip_vdestroy_d(a));
    vsip_finalize((void *)0);
    return 0;
}
/* output */
/* 1.000000 10.000000 100.000000 1000.000000 */
```

See Also

vsip_*ds*exp_*p*, vsip_*ds*log_*p*, and vsip_*s*log10_*p*

vsip_*ds*log_*p* Vector/Matrix Log

Computes the natural logarithm for each element of a vector/matrix.

Functionality

Real:

$r_j \leftarrow \log_e a_j$ for $j = 0, 1, \ldots, N\text{-}1$

$r_{i,j} \leftarrow \log_e a_{i,j}$ for $i = 0, 1, \ldots, M\text{-}1$; for $j = 0, 1, \ldots, N\text{-}1$

Complex:

$r_k \leftarrow \log_e |a_k| + j \cdot \arg(a_k)$ for $k = 0, 1, \ldots, N\text{-}1$

$r_{k,l} \leftarrow \log_e |a_{k,l}| + j \cdot \arg(a_{k,l})$ for $k = 0, 1, \ldots, M\text{-}1$; for $l = 0, 1, \ldots, N\text{-}1$

Prototypes

```
void vsip_vlog_f(
                      const vsip_vview_f *a,
                      const vsip_vview_f *r);
void vsip_cvlog_f(
                      const vsip_cvview_f *a,
                      const vsip_cvview_f *r);
void vsip_mlog_f(
                      const vsip_mview_f *a,
                      const vsip_mview_f *r);
void vsip_cmlog_f(
                      const vsip_cmview_f *a,
                      const vsip_cmview_f *r);
```

Arguments

`*a`	View of input vector/matrix
`*r`	View of output vector/matrix

Return Value

Restrictions

For the real case, arguments less than or equal to zero are not in the domain of log and the result is implementation dependent.

For the complex case, arguments where both the real and imaginary portions are zero are not defined and the result is implementation dependent.

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.
3. The input and output views must be identical views of the same block (in-place), or must not overlap.

Notes/References

Examples

```
/* calculate a = log(x^2) for x = {1,2,...,N} */
#include<stdio.h>
#include "vsip.h"

#define N  4                    /* length */

int main()
{
    double data[N];             /*a user created data space */
    int i;
    vsip_vview_d* a;
    vsip_vview_d* x;

    vsip_init((void *)0);
    a = vsip_vbind_d(vsip_blockbind_d(data,N, VSIP_MEM_NONE),0,1,N);
    x = vsip_vcreate_d(N, VSIP_MEM_NONE);

    vsip_vramp_d(1.0, 1.0, x);
    vsip_vsq_d(x, x);

    /* admit the user block with no update */
    vsip_blockadmit_d(vsip_vgetblock_d(a),VSIP_FALSE);

    /* compute the log value */
    vsip_vlog_d(x, a);

    /* release the user block with update */
    vsip_blockrelease_d(vsip_vgetblock_d(a),VSIP_TRUE);

    /* print it */
    for(i=0; i<N; i++)
      printf("%f ",data[i]);
    printf("\n");

    /* destroy the vector views and any associated blocks */
    vsip_blockdestroy_d(vsip_vdestroy_d(x));
    vsip_blockdestroy_d(vsip_vdestroy_d(a));
    vsip_finalize((void *)0);
    return 0;
}
/* output */
/* 0.000000 1.386294 2.197225 2.772589 */
```

See Also

`vsip_`***ds***`exp_`***p***, `vsip_`***s***`exp10_`***p***, and `vsip_`***s***`log10_`***p***

vsip_slog10_*p* Vector/Matrix Log Base 10

Compute the base ten logarithm for each element of a vector/matrix.

Functionality

$r_j \leftarrow \log_{10} a_j$ for $j = 0, 1, \ldots, N\text{-}1$

$r_{i,j} \leftarrow \log_{10} a_{i,j}$ for $i = 0, 1, \ldots, M\text{-}1$; for $j = 0, 1, \ldots, N\text{-}1$

Prototypes

```
void vsip_vlog10_f(
                    const vsip_vview_f *a,
                    const vsip_vview_f *r);
void vsip_mlog10_f(
                    const vsip_mview_f *a,
                    const vsip_mview_f *r);
```

Arguments

`*a`	View of input vector/matrix
`*r`	View of output vector/matrix

Return Value

None.

Restrictions

Arguments less than or equal to zero are not in the domain of log and the result is implementation dependent.

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.
3. The input and output views must be identical views of the same block (in-place), or must not overlap.

Notes/References

Examples

```
/* calculate a = log10(x^2) for x = {1,2,...,N} */
#include<stdio.h>
#include "vsip.h"

#define N  4                    /* length */

int main()
{
    double data[N];             /*a user created data space */
    int i;
    vsip_vview_d* a;
    vsip_vview_d* x;

    vsip_init((void *)0);
    a = vsip_vbind_d(vsip_blockbind_d(data,N, VSIP_MEM_NONE),0,1,N);
    x = vsip_vcreate_d(N, VSIP_MEM_NONE);
    vsip_vramp_d(1.0, 1.0, x);
```

```
    vsip_vsq_d(x, x);

    /* admit the user block with no update */
    vsip_blockadmit_d(vsip_vgetblock_d(a),VSIP_FALSE);

    /*compute the log value */
    vsip_vlog10_d(x, a);

    /* release the user block with update */
    vsip_blockrelease_d(vsip_vgetblock_d(a),VSIP_TRUE);

    /*print it */
    for(i=0; i<N; i++)
      printf("%f ",data[i]);
    printf("\n");

    /*destroy the vector views and any associated blocks */
    vsip_blockdestroy_d(vsip_vdestroy_d(x));
    vsip_blockdestroy_d(vsip_vdestroy_d(a));
    vsip_finalize((void *)0);
    return 0;
}
/* output */
/* 0.000000 0.602060 0.954243 1.204120 */
```

See Also

vsip_***ds***exp_***p***, vsip_***s***exp10_***p***, and vsip_***s***log_***p***

vsip_ssin_p Vector/Matrix Sine

Compute the sine for each element of a vector/matrix. Element angle values are in radians.

Functionality

$r_j \leftarrow \sin a_j$ for $j = 0, 1, \ldots, N\text{-}1$

$r_{i,j} \leftarrow \sin a_{i,j}$ for $i = 0, 1, \ldots, M\text{-}1$; for $j = 0, 1, \ldots, N\text{-}1$

Prototypes

```
void vsip_vsin_f(
                    const vsip_vview_f *a,
                    const vsip_vview_f *r);
void vsip_msin_f(
                    const vsip_mview_f *a,
                    const vsip_mview_f *r);
```

Arguments

`*a`	View of input vector/matrix
`*r`	View of output vector/matrix

Return Value

None.

Restrictions

Behavior of element values outside of the closed interval [-2π, 2π] is implementation dependent.

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.
3. The input and output views must be identical views of the same block (in-place), or must not overlap.

Notes/References

Input arguments are expressed in radians.

Examples

```
/* print a sine wave for seven evenly spaced values between */
/* zero and two pi. */
#include<stdio.h>
#include "vsip.h"

#define PI 3.1415926535
#define L 7                    /* length */

int main()
{
    double data[L];            /*a user created data space */
    int i;
    vsip_vview_d* ramp;
    vsip_vview_d* output;

    vsip_init((void *)0);
    ramp = vsip_vcreate_d(L, VSIP_MEM_NONE);
```

```
    output = vsip_vbind_d(vsip_blockbind_d(data,L, VSIP_MEM_NONE),0,1,L);

    /*compute a ramp from zero to 2 pi */
    vsip_vramp_d(0.0, (2.0 * PI / (double) (L - 1)), ramp);

    /* admit the user block with no update */
    vsip_blockadmit_d(vsip_vgetblock_d(output),VSIP_FALSE);

    /*compute the sine value */
    vsip_vsin_d(ramp, output);

    /* release the user block with update */
    vsip_blockrelease_d(vsip_vgetblock_d(output),VSIP_TRUE);

    /*print it */
    for(i=0; i<L; i++)
        printf("%f ",data[i]);
    printf("\n");

    /*destroy the vector views and any associated blocks */
    vsip_blockdestroy_d(vsip_vdestroy_d(ramp));
    vsip_blockdestroy_d(vsip_vdestroy_d(output));
    vsip_finalize((void *)0);
    return 0;
}
/* output */
/* 0.000000 0.866025 0.866025 0.000000 -0.866025 -0.866025 -0.000000 */
```

See Also

vsip_*s*acos_*p*, vsip_*s*asin_*p*, vsip_*s*atan_*p*, vsip_*s*atan2_*p*, vsip_*s*cos_*p*, and vsip_*s*tan_*p*

vsip_ssinh_p Vector/Matrix Hyperbolic Sine

Compute the hyperbolic sine for each element of a vector/matrix.

Functionality

$r_j \leftarrow \sinh a_j$ for $j = 0, 1, \ldots, N\text{-}1$

$r_{i,j} \leftarrow \sinh a_{i,j}$ for $i = 0, 1, \ldots, M\text{-}1$; for $j = 0, 1, \ldots, N\text{-}1$

Prototypes

```
void vsip_vsinh_f(
                    const vsip_vview_f *a,
                    const vsip_vview_f *r);
void vsip_msinh_f(
                    const vsip_mview_f *a,
                    const vsip_mview_f *r);
```

Arguments

`*a`	View of input vector/matrix
`*r`	View of output vector/matrix

Return Value

None.

Restrictions

Overflow behavior and domain restrictions are implementation dependent.

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.
3. The input and output views must be identical views of the same block (in-place), or must not overlap.

Notes/References

Examples

```
/* Print sinh(x) for x=0,1,...,6 */
#include<stdio.h>
#include "vsip.h"

#define L 7                 /* length */

int main()
{
    double data[L];         /*a user created data space */
    int i;
    vsip_vview_d* ramp;
    vsip_vview_d* output;

    vsip_init((void *)0);
    ramp = vsip_vcreate_d(L, VSIP_MEM_NONE);
    output = vsip_vbind_d(vsip_blockbind_d(data,L, VSIP_MEM_NONE),0,1,L);

    /*compute a ramp from zero to L-1*/
```

```
    vsip_vramp_d(0.0, 1.0, ramp);

    /* admit the user block with no update */
    vsip_blockadmit_d(vsip_vgetblock_d(output),VSIP_FALSE);

    /*compute the hyperbolic sine value */
    vsip_vsinh_d(ramp, output);

    /* release the user block with update */
    vsip_blockrelease_d(vsip_vgetblock_d(output),VSIP_TRUE);

    /*print it */
    for(i=0; i<L; i++)
        printf("%f ",data[i]);
    printf("\n");

    /*destroy the vector views and any associated blocks */
    vsip_blockdestroy_d(vsip_vdestroy_d(ramp));
    vsip_blockdestroy_d(vsip_vdestroy_d(output));
    vsip_finalize((void *)0);
    return 0;
}
/* output */
/*0.000000 1.175201 3.626860 10.017875 27.289917 74.203211 201.713157 */
```

See Also

`vsip_`***s***`cosh_`***p***, and `vsip_`***s***`tanh_`***p***

vsip_*ds*sqrt_*p* Vector/Matrix Square Root

Compute the square root for each element of a vector/matrix.

Functionality

Real:

$$r_j \leftarrow \sqrt{a_j} \qquad \text{for } j = 0, 1, \ldots, N\text{-}1$$

$$r_{i,j} \leftarrow \sqrt{a_{i,j}} \qquad \text{for } i = 0, 1, \ldots, M\text{-}1;\ \text{for } j = 0, 1, \ldots, N\text{-}1$$

Complex:

$$\phi_k \equiv \arg(a_k)$$

$$r_k \leftarrow \sqrt{|a_k|} \cdot \left(\cos\frac{\phi_k}{2} + j \cdot \sin\frac{\phi_k}{2}\right) \qquad \text{for } k = 0, 1, \ldots, N\text{-}1$$

$$\phi_{k,l} \equiv \arg(a_{k,l})$$

$$r_{k,l} \leftarrow \sqrt{|a_{k,l}|} \cdot \left(\cos\frac{\phi_{k,l}}{2} + j \cdot \sin\frac{\phi_{k,l}}{2}\right) \qquad \text{for } k = 0, 1, \ldots, M\text{-}1;\ \text{for } l = 0, 1, \ldots, N\text{-}1$$

Prototypes

```
void vsip_vsqrt_f(
                    const vsip_vview_f *a,
                    const vsip_vview_f *r,);
void vsip_msqrt_f(
                    const vsip_mview_f *a,
                    const vsip_mview_f *r,);
void vsip_cvsqrt_f(
                    const vsip_cvview_f *a,
                    const vsip_cvview_f *r,);
void vsip_cmsqrt_f(
                    const vsip_cmview_f *a,
                    const vsip_cmview_f *r,);
```

Arguments

`*a`	View of input vector/matrix
`*r`	View of output vector/matrix

Return Value

None.

Restrictions

For square root calculation in the real case the argument must be greater than or equal to zero to be within the domain of the function. Results for cases where the argument is less than zero is implementation dependent.

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.
3. The input and output views must be identical views of the same block (in-place), or must not overlap.

Notes/References

Examples

Compute $\sqrt{2 \cdot e^{jn\frac{\pi}{4}}}$ for $n = 0,1,\cdots,7$

```
#include<stdio.h>
#include "vsip.h"

#define PI 3.1415926535
#define N 8                        /* length */

int main()
{
    vsip_scalar_d R = 2.0;
    vsip_scalar_d dataR[N];  /* a user created data space for real */
    vsip_scalar_d dataI[N];  /* a user created data space for imaginary */
    int i;
    vsip_vview_d*  radius;
    vsip_vview_d*  arg;
    vsip_cvview_d* cVsipVector;
    vsip_cblock_d* UsrBlock;
    vsip_cvview_d* cUsrVector;

    vsip_init((void *)0);
    radius = vsip_vcreate_d(N, VSIP_MEM_NONE);
    arg = vsip_vcreate_d(N, VSIP_MEM_NONE);
    cVsipVector = vsip_cvcreate_d(N, VSIP_MEM_NONE);
    UsrBlock = vsip_cblockbind_d(dataR,dataI, N, VSIP_MEM_NONE);
    cUsrVector = vsip_cvbind_d(UsrBlock, 0,1,N);

    /* Admit the user block as Complex data */
    vsip_cblockadmit_d(UsrBlock,VSIP_FALSE);

    /* compute arg */
    vsip_vramp_d(0.0, (2.0 * PI / (double) N ), arg);

    /* compute radius */
    vsip_vfill_d(R, radius);

    /* make the input vector */
    vsip_veuler_d(arg,cVsipVector);
    vsip_rcvmul_d(radius, cVsipVector, cVsipVector);

    /* compute the sqrt value */
    vsip_cvsqrt_d(cVsipVector,cUsrVector);

    /* release the usr block */
```

```
    { vsip_scalar_d *a,*b;
      vsip_cblockrelease_d(UsrBlock, VSIP_TRUE, &a, &b);
    }
    /* print it */
    for(i=0; i < N; i++)
       printf("%7.4f + %7.4fi = sqrt(%7.4f + %7.4fi) \n",
                dataR[i],dataI[i],
                vsip_real_d(vsip_cvget_d(cVsipVector,i)),
                vsip_imag_d(vsip_cvget_d(cVsipVector,i)));

    /* destroy the vector views and any associated blocks */
    vsip_blockdestroy_d(vsip_vdestroy_d(radius));
    vsip_blockdestroy_d(vsip_vdestroy_d(arg));
    vsip_cblockdestroy_d(vsip_cvdestroy_d(cUsrVector));
    vsip_cblockdestroy_d(vsip_cvdestroy_d(cVsipVector));
    vsip_finalize((void *)0);
    return 0;
}
/* output */
/*
 1.4142 +  0.0000i = sqrt( 2.0000 +  0.0000i)
 1.3066 +  0.5412i = sqrt( 1.4142 +  1.4142i)
 1.0000 +  1.0000i = sqrt( 0.0000 +  2.0000i)
 0.5412 +  1.3066i = sqrt(-1.4142 +  1.4142i)
 0.0000 +  1.4142i = sqrt(-2.0000 +  0.0000i)
 0.5412 + -1.3066i = sqrt(-1.4142 + -1.4142i)
 1.0000 + -1.0000i = sqrt(-0.0000 + -2.0000i)
 1.3066 + -0.5412i = sqrt( 1.4142 + -1.4142i) */
```

See Also

vsip_*ds*rsqrt_*p*

vsip_stan_p Vector/Matrix Tangent

Compute the tangent for each element of a vector/matrix. Element angle values are in radians.

Functionality

$r_j \leftarrow \tan a_j$ for $j = 0, 1, \ldots, N\text{-}1$

$r_{i,j} \leftarrow \tan a_{i,j}$ for $i = 0, 1, \ldots, M\text{-}1$; for $j = 0, 1, \ldots, N\text{-}1$

Prototypes

```
void vsip_vtan_f(
                    const vsip_vview_f *a,
                    const vsip_vview_f *r);
void vsip_mtan_f(
                    const vsip_mview_f *a,
                    const vsip_mview_f *r);
```

Arguments

`*a`	View of input vector/matrix
`*r`	View of output vector/matrix

Return Value

None.

Restrictions

Behavior of element values outside of the closed interval [-2π, 2π] is implementation dependent. For element values $\left(n + \frac{1}{2}\right)\pi$, the tan function has a singularity. The results of these values are implementation dependent.

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.
3. The input and output views must be identical views of the same block (in-place), or must not overlap.

Notes/References

Examples

```
/* calculate a= tan(x) in each quadrant at the midpoint */
#include<stdio.h>
#include "vsip.h"

#define PI 3.1415926535
#define N  4                    /* length */

int main()
{
    double data[N];             /*a user created data space */
    int i;
    vsip_vview_d* a;
    vsip_vview_d* x;

    vsip_init((void *)0);
```

```
    a = vsip_vbind_d(vsip_blockbind_d(data,N, VSIP_MEM_NONE),0,1,N);
    x = vsip_vcreate_d(N, VSIP_MEM_NONE);
    vsip_vramp_d(PI/4.0,2.0 * PI/N,x);

    /* admit the user block with no update */
    vsip_blockadmit_d(vsip_vgetblock_d(a),VSIP_FALSE);

    /*compute the tan value */
    vsip_vtan_d(x,a);

    /* release the user block with update */
    vsip_blockrelease_d(vsip_vgetblock_d(a),VSIP_TRUE);

    /*print it */
    for(i=0; i<N; i++)
      printf("%f ",data[i]);
    printf("\n");

    /*destroy the vector views and any associated blocks */
    vsip_blockdestroy_d(vsip_vdestroy_d(x));
    vsip_blockdestroy_d(vsip_vdestroy_d(a));
    vsip_finalize((void *)0);
    return 0;
}
/* output */
/* 1.000000 -1.000000 1.000000 -1.000000 */
```

See Also

vsip_*s*acos_*p*, vsip_*s*asin_*p*, vsip_*s*atan_*p*, vsip_*s*atan2_*p*, vsip_*s*cos_*p*, and vsip_*s*sin_*p*

vsip_stanh_p Vector/Matrix Hyperbolic Tangent

Compute the hyperbolic tangent for each element of a vector/matrix.

Functionality

$r_j \leftarrow \tanh a_j$ for $j = 0, 1, \ldots, N\text{-}1$

$r_{i,j} \leftarrow \tanh a_{i,j}$ for $i = 0, 1, \ldots, M\text{-}1$; for $j = 0, 1, \ldots, N\text{-}1$

Prototypes

```
void vsip_vtanh_f(
                    const vsip_vview_f *a,
                    const vsip_vview_f *r);
void vsip_mtanh_f(
                    const vsip_mview_f *a,
                    const vsip_mview_f *r);
```

Arguments

`*a`	View of input vector/matrix
`*r`	View of output vector/matrix

Return Value

None.

Restrictions

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.
3. The input and output views must be identical views of the same block (in-place), or must not overlap.

Notes/References

Examples

```
/* print tanh(x) for x=0,1,...,6 */
#include<stdio.h>
#include "vsip.h"

#define L 7           /* length */

int main()
{
    double data[L];   /*a user created data space */
    int i;
    vsip_vview_d* ramp;
    vsip_vview_d* output;

    vsip_init((void *)0);
    ramp = vsip_vcreate_d(L, VSIP_MEM_NONE);
    output = vsip_vbind_d(vsip_blockbind_d(data,L, VSIP_MEM_NONE),0,1,L);

    /*compute a ramp from zero to L-1*/
    vsip_vramp_d(0.0, 1.0, ramp);
```

```
    /* admit the user block with no update */
    vsip_blockadmit_d(vsip_vgetblock_d(output),VSIP_FALSE);

    /*compute the hyperbolic tangent values */
    vsip_vtanh_d(ramp, output);

    /* release the user block with update */
    vsip_blockrelease_d(vsip_vgetblock_d(output),VSIP_TRUE);

    /*print it */
    for(i=0; i<L; i++)
        printf("%f ",data[i]);
    printf("\n");

    /*destroy the vector views and any associated blocks */
    vsip_blockdestroy_d(vsip_vdestroy_d(ramp));
    vsip_blockdestroy_d(vsip_vdestroy_d(output));
    vsip_finalize((void *)0);
    return 0;
}
/* output */
/*0.000000 0.761594 0.964028 0.995055 0.999329 0.999909 0.999988*/
```

See Also

`vsip_`***s***`cosh_`***p***, and `vsip_`***s***`sinh_`***p***

Unary Operations

The following functions represent operations done on a single vector/matrix.

vsip_*s*arg_*p*	Vector/Matrix Argument
vsip_*s*ceil_*p*_*p*	Vector Matrix Ceiling
vsip_c*s*conj_*p*	Vector/Matrix Conjugate
vsip_*ds*cumsum_*p*	Cumulative Sum
vsip_*s*euler_*p*	Vector/Matrix Euler
vsip_*s*floor_*p*_*p*	Vector/Matrix Floor
vsip_*ds*mag_*p*	Vector/Matrix Magnitude
vsip_*s*cmagsq_*p*	Vector/Matrix Complex Magnitude Squared
vsip_*ds*meanval_*p*	Vector/Matrix Mean Value
vsip_*ds*meansqval_*p*	Vector/Matrix Mean Square Value
vsip_*d*vmodulate_*p*	Vector Modulate
vsip_*ds*neg_*p*	Vector/Matrix Negate
vsip_*ds*recip_*p*	Vector/Matrix Reciprocal
vsip_*s*round_*p*_*p*	Vector/Matrix Round
vsip_*ds*rsqrt_*p*	Vector/Matrix Reciprocal Square Root
vsip_*ds*sq_*p*	Vector/Matrix Square
vsip_*ds*sumval_*p*	Vector/Matrix Sum Value
vsip_*ds*sumsqval_*p*	Vector/Matrix Sum of Squares Value

vsip_*sarg*_*p* Vector/Matrix Argument

Compute the radian value argument, in the interval [-π, π], for each element of a complex vector/matrix.

Functionality

$$r_j \leftarrow \tan^{-1}\left(\mathrm{Im}(a_j) \big/ \mathrm{Re}(a_j)\right) \qquad \text{for } j = 0, 1, \ldots, N\text{-}1$$

$$r_{i,j} \leftarrow \tan^{-1}\left(\mathrm{Im}(a_{i,j}) \big/ \mathrm{Re}(a_{i,j})\right) \qquad \text{for } i = 0, 1, \ldots, M\text{-}1;\ \text{for } j = 0, 1, \ldots, N\text{-}1$$

Where $\arctan(x/y)$ here is the inverse tangent corresponding to the ANSI C function `atan2`. For a more complete definition see functionality section of `vsip_satan2_p` definition.

Prototypes

```
void vsip_varg_f(
                      const vsip_cvview_f *a,
                      const vsip_vview_f *r);
void vsip_marg_f(
                      const vsip_cmview_f *a,
                      const vsip_mview_f *r);
```

Arguments

`*a`	View of input vector/matrix
`*r`	View of output vector/matrix

Return Value

None.

Restrictions

For in-place the output argument may be either the derived real view or the imaginary view of the input complex view. In-place is not defined if the output data is not either a real or imaginary view of the input data.

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.
3. The input and output views must be "in-place" as described in the restrictions, or must not overlap.

Notes/References

Examples

```
/* Example of arg*/
#include<stdio.h>
#include "vsip.h"

#define PI 3.1415926535
#define L 7                    /* length */

int main()
{
    int i;
    vsip_vview_d* dataArg;
    vsip_cvview_d* dataCmplx;
    vsip_vview_d* dataRe;
    vsip_vview_d* dataIm;

    vsip_init((void *)0);
    dataArg = vsip_vcreate_d(L, VSIP_MEM_NONE);
    dataCmplx = vsip_cvcreate_d(L, VSIP_MEM_NONE);
    dataRe = vsip_vrealview_d(dataCmplx);
    dataIm = vsip_vimagview_d(dataCmplx);

    /* Make up some data to find the arg of */
    /* First compute a ramp from zero to 2pi */
    vsip_vramp_d(0.0, (2.0 * PI / (double) (L - 1)), dataArg);

    /*and transform it to a sin (real) and a cos (imaginary).*/
    vsip_vsin_d(dataArg,dataRe);
    vsip_vcos_d(dataArg,dataIm);

    /* Find the argument */
    vsip_varg_d(dataCmplx,dataArg);

    /*now print out dataCmplex and its argument*/
    for(i=0; i<L; i++)
        printf("(%7.4f, %7.4f) => %7.4f\n",
             vsip_vget_d(dataRe,i),
             vsip_vget_d(dataIm,i),
             vsip_vget_d(dataArg,i));

    /*destroy the vector views and any associated blocks */
    vsip_blockdestroy_d(vsip_vdestroy_d(dataArg));
    vsip_vdestroy_d(dataRe);
    vsip_vdestroy_d(dataIm);
    vsip_cblockdestroy_d(vsip_cvdestroy_d(dataCmplx));
    vsip_finalize((void *)0);
    return 0;
}
/* output */
/* ( 0.0000,  1.0000) =>  1.5708
   ( 0.8660,  0.5000) =>  0.5236
   ( 0.8660, -0.5000) => -0.5236
   ( 0.0000, -1.0000) => -1.5708
   (-0.8660, -0.5000) => -2.6180
   (-0.8660,  0.5000) =>  2.6180
   (-0.0000,  1.0000) =>  1.5708 */
```

See Also

`vsip_`***s***`atan2_`***p***, `vsip_`***s***`euler_`***p***, and `vsip_`***s***`polar_`***p***

vsip_sceil_p_p Vector/Matrix Ceiling

For each element in the input view round to the smallest integral value not less than the input and place the result in the output view elementwise.

Functionality

$r_i \leftarrow \lceil x_i \rceil$ for $i = 0, 1, \ldots, N\text{-}1$

$r_{i,j} \leftarrow \lceil x_{i,j} \rceil$ for $i = 0, 1, \ldots, M\text{-}1$; for $j = 0, 1, \ldots, N\text{-}1$

Prototypes

```
void vsip_vceil_f_f(
                    const vsip_vview_f *x,
                    const vsip_vview_f *r);
void vsip_vceil_f_i(
                    const vsip_vview_f *x,
                    const vsip_vview_i *r);
void vsip_mceil_f_f(
                    const vsip_mview_f *x,
                    const vsip_mview_f *r);
void vsip_mceil_f_i(
                    const vsip_mview_f *x,
                    const vsip_mview_i *r);
```

Arguments

`*x`	View of input vector/matrix
`*r`	View of output vector/matrix

Return Value

None

Restrictions

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.
3. The input and output views must be identical views of the same block (in-place), or must not overlap.

Notes/References

Although there is no VSIPL requirement to fully support IEEE 754 the intent of this function is to support the *round to +INFINITY* functionality as described in IEEE 754.

Examples

See Also

See also `vsip_sfloor_p_p` and `vsip_sround_p_p`.

vsip_csconj_*p* Vector/Matrix Conjugate

Compute the conjugate for each element of a complex vector/matrix.

Functionality

$r_j \leftarrow a_j^*$ for $j = 0, 1, \ldots, N\text{-}1$

$r_{i,j} \leftarrow a_{i,j}^*$ for $i = 0, 1, \ldots, M\text{-}1$; for $j = 0, 1, \ldots, N\text{-}1$

Where "*" denotes complex conjugate.

Prototypes

```
void vsip_cvconj_f(
                    const vsip_cvview_f *a,
                    const vsip_cvview_f *r);
void vsip_cmconj_f(
                    const vsip_cmview_f *a,
                    const vsip_cmview_f *r);
```

Arguments

`*a`	View of input vector/matrix
`*r`	View of output vector/matrix

Return Value

None.

Restrictions

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.
3. The input and output views must be identical views of the same block (in-place), or must not overlap.

Notes/References

Examples

```
/* Example of cvconj*/
#include<stdio.h>
#include "vsip.h"

#define PI 3.1415926535
#define L 7                    /* length */

int main()
{
    int i;
    vsip_vview_d*  dataRe;
    vsip_vview_d*  dataIm;
    vsip_cvview_d* dataCmplx;
    vsip_cvview_d* dataCmplxConj;
    vsip_cscalar_d cscalar;
    vsip_cscalar_d cconjscalar;

    vsip_init((void *)0);
    dataRe = vsip_vcreate_d(L, VSIP_MEM_NONE);
```

```
    dataIm = vsip_vcreate_d(L, VSIP_MEM_NONE);
    dataCmplx = vsip_cvcreate_d(L, VSIP_MEM_NONE);
    dataCmplxConj = vsip_cvcreate_d(L, VSIP_MEM_NONE);

    /* Make up some data to find the complex conjugate of */
    /* First compute a ramp from zero to 2pi */
    vsip_vramp_d(0.0, (2.0 * PI / (double) (L - 1)), dataRe);

    /*and transform it to a sin.*/
    vsip_vsin_d(dataRe,dataRe);

    /* Then compute a ramp from zero to 3pi */
    vsip_vramp_d(0.0, (3.0 * PI / (double) (L - 1)), dataIm);

    /* and transform it to a cos. */
    vsip_vcos_d(dataIm,dataIm);

    /* Finally make a complex vector. */
    vsip_vcmplx_d(dataRe, dataIm, dataCmplx);

    /* Find the Complex Conjugate */
    vsip_cvconj_d(dataCmplx,dataCmplxConj);

    /* now print out dataCmplex and its Conjugate */
    for(i=0; i<L; i++){
        cscalar = vsip_cvget_d(dataCmplx, (vsip_scalar_vi) i);
        cconjscalar = vsip_cvget_d(dataCmplxConj, (vsip_scalar_vi) i);
        printf("(%7.4f, %7.4f) => (%7.4f, %7.4f)\n",cscalar.r, cscalar.i,
             cconjscalar.r, cconjscalar.i);
    }
    vsip_blockdestroy_d(vsip_vdestroy_d(dataRe));
    vsip_blockdestroy_d(vsip_vdestroy_d(dataIm));
    vsip_cblockdestroy_d(vsip_cvdestroy_d(dataCmplx));
    vsip_cblockdestroy_d(vsip_cvdestroy_d(dataCmplxConj));
    vsip_finalize((void *)0);
    return 0;
}
/* output */
/* ( 0.0000,  1.0000) => ( 0.0000, -1.0000)
   ( 0.8660,  0.0000) => ( 0.8660, -0.0000)
   ( 0.8660, -1.0000) => ( 0.8660,  1.0000)
   ( 0.0000, -0.0000) => ( 0.0000,  0.0000)
   (-0.8660,  1.0000) => (-0.8660, -1.0000)
   (-0.8660,  0.0000) => (-0.8660, -0.0000)
   (-0.0000, -1.0000) => (-0.0000,  1.0000) */
```

See Also

vsip_*ds*neg_*p*

vsip_*ds*cumsum_*p* Cumulative Sum

Compute the cumulative sum of the elements of a vector.

Functionality

$$r_n \leftarrow \sum_{i=0}^{n} x_i \qquad \text{for } n = 0, 1, \ldots, N\text{-}1$$

Row:

$$r_{m,n} \leftarrow \sum_{i=0}^{n} x_{m,i} \qquad \text{for } m = 0, 1, \ldots, M\text{-}1; \text{ for } n = 0, 1, \ldots, N\text{-}1$$

Column:

$$r_{m,n} \leftarrow \sum_{i=0}^{n} x_{i,n} \qquad \text{for } m = 0, 1, \ldots, M\text{-}1; \text{ for } n = 0, 1, \ldots, N\text{-}1$$

Prototypes

```
void vsip_vcumsum_f(
                      const vsip_vview_f *x,
                      const vsip_vview_f *r);
void vsip_vcumsum_i(
                      const vsip_vview_i *x,
                      const vsip_vview_i *r);
void vsip_cvcumsum_f(
                      const vsip_cvview_f *x,
                      const vsip_cvview_f *r);
void vsip_cvcumsum_i(
                      const vsip_cvview_i *x,
                      const vsip_cvview_i *r);
void vsip_mcumsum_f(
                      const vsip_mview_f *x,
                      vsip_major dir,
                      const vsip_mview_f *r);
void vsip_mcumsum_i(
                      const vsip_mview_i *x,
                      vsip_major dir,
                      const vsip_mview_i *r);
void vsip_cmcumsum_f(
                      const vsip_cmview_f *x,
                      vsip_major dir,
                      const vsip_cmview_f *r);
void vsip_cmcumsum_i(
                      const vsip_cmview_i *x,
                      vsip_major dir,
                      const vsip_cmview_i *r);
```

Arguments

`*x` Pointer to input view object.

`dir` For matrix cumulative sum specifies the direction the sum is done over.

```
typedef enum {
  VSIP_ROW,
  VSIP_COL
} vsip_major;
```

`*r` Pointer to output view object.

Return Value

None.

Restrictions

Errors

The following cause a runtime VSIPL error if compiled in development mode. If compiled in production mode the results will be implementation dependent.

1. Arguments for input and output must be the same size.
2. Arguments passed to the function must be defined and must not be *null*.
3. The input and output views must either be the same or must not overlap.

Notes/References

Used to compute boxcar integration, moving average, and other sliding window functions.

Examples

See Also

vsip_seuler_*p* Vector/Matrix Euler

Computes the complex numbers corresponding to the angle of a unit vector in the complex plane for each element of a vector/matrix.

Functionality

$r_k \leftarrow \cos a_k + j \cdot \sin a_k = e^{j \cdot a_k}$ for $k = 0, 1, \ldots, N\text{-}1$

$r_{k,l} \leftarrow \cos a_{k,l} + j \cdot \sin a_{k,l} = e^{j \cdot a_{k,l}}$ for $k = 0, 1, \ldots, M\text{-}1$; for $l = 0, 1, \ldots, N\text{-}1$

Prototypes

```
void vsip_veuler_f(
                      const vsip_vview_f *a,
                      const vsip_cvview_f *r);
void vsip_meuler_f(
                      const vsip_mview_f *a,
                      const vsip_cmview_f *r);
```

Arguments

`*a`	View of input vector/matrix
`*r`	View of output vector/matrix

Return Value

None.

Restrictions

In-place operation implies that the input is either a derived real or imaginary view of the output view.

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.
3. The input and output views must be “in-place” as described in the restrictions, or must not overlap.

Notes/References

The result for large arguments may not be accurate and is limited by the method of conversion of the argument to its principal value

The speed may be adversely impacted for large arguments because of conversion of the argument to its principal value.

Examples

```
/* Example of euler */
#include<stdio.h>
#include "vsip.h"

#define PI 3.1415926535
#define L 7                    /* length */

int main()
{
    int i;
    vsip_cvview_d* dataEuler;
    vsip_vview_d*  data;

    vsip_init((void *)0);
    dataEuler = vsip_cvcreate_d(L, VSIP_MEM_NONE);
    data      = vsip_vcreate_d(L, VSIP_MEM_NONE);

    /* Make up some data */
    /* Compute a ramp from zero to 2pi */
    vsip_vramp_d(0.0, (2.0 * PI / (double) (L - 1)), data);

    /* Compute Euler */
    vsip_veuler_d(data,dataEuler);

    /* Now print out data and dataEuler */
    for(i=0; i<L; i++){
        printf(" %7.4f => (%7.4f, %7.4f)\n",vsip_vget_d(data,i),
                      vsip_real_d(vsip_cvget_d(dataEuler,i)),
                      vsip_imag_d(vsip_cvget_d(dataEuler,i)));
    }
    /* Destroy the vector views and any associated blocks */
    vsip_blockdestroy_d(vsip_vdestroy_d(data));
    vsip_cblockdestroy_d(vsip_cvdestroy_d(dataEuler));
    vsip_finalize((void *)0);
    return 0;
}
/* output */
/*  0.0000 => ( 1.0000,  0.0000)
    1.0472 => ( 0.5000,  0.8660)
    2.0944 => (-0.5000,  0.8660)
    3.1416 => (-1.0000,  0.0000)
    4.1888 => (-0.5000, -0.8660)
    5.2360 => ( 0.5000, -0.8660)
    6.2832 => ( 1.0000, -0.0000) */
```

See Also

`vsip_`***ds***`exp_`***p***, and `vsip_`***d***`vmodulate_`***p***

vsip_*sfloor_p_p* Vector/Matrix Floor

For each element in the input view round to the largest integral value not greater than the input and place the result in the output view elementwise.

Functionality

$r_i \leftarrow \lfloor x_i \rfloor$ for $i = 0, 1, \ldots, N\text{-}1$

$r_{i,j} \leftarrow \lfloor x_{i,j} \rfloor$ for $i = 0, 1, \ldots, M\text{-}1$; for $j = 0, 1, \ldots, N\text{-}1$

Prototypes

```
void vsip_vfloor_f_f(
                     const vsip_vview_f *x,
                     const vsip_vview_f *r);
void vsip_vfloor_f_i(
                     const vsip_vview_f *x,
                     const vsip_vview_i *r);
void vsip_mfloor_f_f(
                     const vsip_mview_f *x,
                     const vsip_mview_f *r);
void vsip_mfloor_f_i(
                     const vsip_mview_f *x,
                     const vsip_mview_i *r);
```

Arguments

`*x`	View of input vector/matrix
`*r`	View of output vector/matrix

Return Value

None

Restrictions

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.
3. The input and output views must be identical views of the same block (in-place), or must not overlap.

Notes/References

Although there is no VSIPL requirement to fully support IEEE 754 the intent of this function is to support the *round to -INFINITY* functionality as described in IEEE 754.

Examples

See Also

See also `vsip_sceil_p_p` and `vsip_sround_p_p`.

vsip_*ds*mag_*p* Vector/Matrix Magnitude

Compute the magnitude for each element of a vector/matrix.

Functionality

$r_j \leftarrow |a_j|$ for $j = 0, 1, \ldots, N\text{-}1$

$r_{i,j} \leftarrow |a_{i,j}|$ for $i = 0, 1, \ldots, M\text{-}1$; for $j = 0, 1, \ldots, N\text{-}1$

Prototypes

```
void vsip_vmag_f(
                      const vsip_vview_f *a,
                      const vsip_vview_f *r);
void vsip_cvmag_f(
                      const vsip_cvview_f *a,
                      const vsip_vview_f *r);
void vsip_mmag_f(
                      const vsip_mview_f *a,
                      const vsip_mview_f *r);
void vsip_cmmag_f(
                      const vsip_cmview_f *a,
                      const vsip_mview_f *r);
```

Arguments

`*a`	View of input vector/matrix
`*r`	View of output vector/matrix

Return Value

None.

Restrictions

In the complex case in-place implies that the output is either a real view or an imaginary view of the input vector. Output views that do not exactly match either a real view, or an imaginary view, are not defined for in-place.

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.
3. Arguments passed to the function whose data space overlap with different offsets or strides may cause overwriting of data before it is used.

Notes/References

Examples

```
/* Example of mag*/
#include<stdio.h>
#include "vsip.h"

#define PI 3.1415926535
#define L 7                    /* length */

int main()
```

```
{
    int i;
    vsip_vview_d*  dataRe;
    vsip_vview_d*  dataIm;
    vsip_vview_d*  dataMag;
    vsip_cvview_d* dataCmplx;
    vsip_cscalar_d cscalar;

    vsip_init((void *)0);
    dataRe    = vsip_vcreate_d(L, VSIP_MEM_NONE);
    dataIm    = vsip_vcreate_d(L, VSIP_MEM_NONE);
    dataMag   = vsip_vcreate_d(L, VSIP_MEM_NONE);
    dataCmplx = vsip_cvcreate_d(L, VSIP_MEM_NONE);

    /* Make up some data to find the magnitude of */
    /* First compute a ramp from zero to 2pi and apply sin */
    vsip_vramp_d(0.0, (2.0 * PI / (double) (L - 1)), dataRe);
    vsip_vsin_d(dataRe, dataRe);

    /* Then compute a ramp from zero to 3pi and apply cos */
    vsip_vramp_d(0.0, (3.0 * PI / (double) (L - 1)), dataIm);
    vsip_vcos_d(dataIm, dataIm);

    /* Finally make a complex vector. */
    vsip_vcmplx_d(dataRe, dataIm, dataCmplx);

    /* Find the Magnitude */
    vsip_cvmag_d(dataCmplx,dataMag);

    /*now print out dataCmplex and its arguments*/
    for(i=0; i<L; i++){
        cscalar = vsip_cvget_d(dataCmplx, (vsip_scalar_vi) i);
        printf("(%7.4f, %7.4f) => %7.4f\n",cscalar.r, cscalar.i,
             vsip_vget_d(dataMag, (vsip_scalar_vi) i));
    }
    /*destroy the vector views and any associated blocks */
    vsip_blockdestroy_d(vsip_vdestroy_d(dataRe));
    vsip_blockdestroy_d(vsip_vdestroy_d(dataIm));
    vsip_blockdestroy_d(vsip_vdestroy_d(dataMag));
    vsip_cblockdestroy_d(vsip_cvdestroy_d(dataCmplx));
    vsip_finalize((void *)0);
    return 0;
}
/* output */
/* ( 0.0000,  1.0000) =>  1.0000
   ( 0.8660,  0.0000) =>  0.8660
   ( 0.8660, -1.0000) =>  1.3229
   ( 0.0000, -0.0000) =>  0.0000
   (-0.8660,  1.0000) =>  1.3229
   (-0.8660,  0.0000) =>  0.8660
   (-0.0000, -1.0000) =>  1.0000 */
```

See Also

vsip_*s*cmagsq_*p*

vsip_scmagsq_*p* Vector/Matrix Complex Magnitude Squared

Computes the square of the magnitudes for each element of a vector/matrix.

Functionality

$$r_j \leftarrow \left(\mathrm{Re}\left(a_j\right)\right)^2 + \left(\mathrm{Im}\left(a_j\right)\right)^2 = \left|a_j\right|^2 \qquad \text{for } j = 0, 1, \ldots, N\text{-}1$$

$$r_{i,j} \leftarrow \left(\mathrm{Re}\left(a_{i,j}\right)\right)^2 + \left(\mathrm{Im}\left(a_{i,j}\right)\right)^2 = \left|a_{i,j}\right|^2 \qquad \text{for } i = 0, 1, \ldots, M\text{-}1;\ \text{for } j = 0, 1, \ldots, N\text{-}1$$

Prototypes

```
void vsip_vcmagsq_f(
                    const vsip_cvview_f *a,
                    const vsip_vview_f *r);
void vsip_mcmagsq_f(
                    const vsip_cmview_f *a,
                    const vsip_mview_f *r);
```

Arguments

`*a`	View of input vector/matrix
`*r`	View of output vector/matrix

Return Value

None.

Restrictions

There is no requirement that intermediate overflows do not occur. Domain restrictions and overflow behavior are implementation dependent.

In-place functionality requires that the output be either a real view or an imaginary view of the input vector. Output views that encompass both real and imaginary portions of the input, or which do not exactly overlay a real or imaginary view of the input are not defined for in-place operations.

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.
3. Arguments passed to the function whose data space overlap with different offsets or strides may cause overwriting of data before it is used.

Notes/References

The order of summation is not specified, therefore significant numerical errors can potentially occur.

Examples

```
/* Example of cmagsq*/
#include<stdio.h>
#include "vsip.h"

#define PI 3.1415926535
#define L 7                    /* length */

int main()
```

```
{
    int i;
    vsip_vview_d*  dataRe;
    vsip_vview_d*  dataIm;
    vsip_vview_d*  dataMagsq;
    vsip_cvview_d* dataCmplx;
    vsip_cscalar_d cscalar;

    vsip_init((void *)0);
    dataRe    = vsip_vcreate_d(L, VSIP_MEM_NONE);
    dataIm    = vsip_vcreate_d(L, VSIP_MEM_NONE);
    dataMagsq = vsip_vcreate_d(L, VSIP_MEM_NONE);
    dataCmplx = vsip_cvcreate_d(L, VSIP_MEM_NONE);

    /* Make up some data to find the magnitude of */
    /* First compute a ramp from zero to 2pi and apply sin */
    vsip_vramp_d(0.0, (2.0 * PI / (double) (L - 1)), dataRe);
    vsip_vsin_d(dataRe,dataRe);

    /* Then compute a ramp from zero to 3pi and apply cos */
    vsip_vramp_d(0.0, (3.0 * PI / (double) (L - 1)), dataIm);
    vsip_vcos_d(dataIm,dataIm);

    /* Finally make a complex vector. */
    vsip_vcmplx_d(dataRe, dataIm, dataCmplx);

    /* Find the Magnitude */
    vsip_vcmagsq_d(dataCmplx,dataMagsq);

    /*now print out dataCmplex and its arguments*/
    for(i=0; i<L; i++){
        cscalar = vsip_cvget_d(dataCmplx, (vsip_scalar_vi) i);
        printf("(%7.4f, %7.4f) => %7.4f\n",cscalar.r, cscalar.i,
             vsip_vget_d(dataMagsq, (vsip_scalar_vi) i));
    }
    /*destroy the vector views and any associated blocks */
    vsip_blockdestroy_d(vsip_vdestroy_d(dataRe));
    vsip_blockdestroy_d(vsip_vdestroy_d(dataIm));
    vsip_blockdestroy_d(vsip_vdestroy_d(dataMagsq));
    vsip_cblockdestroy_d(vsip_cvdestroy_d(dataCmplx));
    vsip_finalize((void *)0);
    return 0;
}
/* output */
/* ( 0.0000,  1.0000) =>  1.0000
   ( 0.8660,  0.0000) =>  0.7500
   ( 0.8660, -1.0000) =>  1.7500
   ( 0.0000, -0.0000) =>  0.0000
   (-0.8660,  1.0000) =>  1.7500
   (-0.8660,  0.0000) =>  0.7500
   (-0.0000, -1.0000) =>  1.0000 */
```

See Also

vsip_*ds*mag_*p*

vsip_*ds*meanval_*p* Vector/Matrix Mean Value

Returns the mean value of the elements of a vector/matrix.

Functionality

$$\text{mean} \leftarrow \frac{1}{N}\sum_{j=0}^{N-1} a_j$$

$$\text{mean} \leftarrow \frac{1}{M \cdot N}\sum_{j=0}^{N-1}\sum_{i=0}^{M-1} a_{i,j}$$

Prototypes

```
vsip_scalar_f vsip_vmeanval_f(
                      const vsip_vview_f *a);
vsip_cscalar_f vsip_cvmeanval_f(
                      const vsip_cvview_f *a);
vsip_scalar_f vsip_mmeanval_f(
                      const vsip_mview_f *a);
vsip_cscalar_f vsip_cmmeanval_f(
                      const vsip_cmview_f *a);
```

Arguments

`*a` View of input vector/matrix

Return Value

Returns the mean value of the elements of the vector/matrix.

Restrictions

Errors

The arguments must conform to the following:

1. All view objects must be valid.

Notes/References

Examples

```
/* Example of meanval */
#include<stdio.h>
#include "vsip.h"

#define L 7                    /* length */

int main()
{
    vsip_vview_d* data;

    vsip_init((void *)0);
    data = vsip_vcreate_d(L, VSIP_MEM_NONE);

    /* Make up some data to find the mean value of */
    /* Compute a ramp from zero to L-1 */
    vsip_vramp_d(0.0, 1.0, data);

    /* And find and print its mean */
    printf("%f \n", vsip_vmeanval_d(data));
    /*destroy the vector views and any associated blocks */
```

```
    vsip_blockdestroy_d(vsip_vdestroy_d(data));
    vsip_finalize((void *)0);
    return 0;
}
/* output */
/* 3.000000 */
```

See Also

vsip_*ds*meansqval_*p*, vsip_*ds*sumval_*p*, and vsip_*ds*sumsqval_*p*

vsip_*ds*meansqval_*p* Vector/Matrix Mean Square Value

Returns the mean magnitude squared value of the elements of a vector/matrix.

Functionality

$$\text{meansq} \leftarrow \frac{1}{N}\sum_{j=0}^{N-1}\left|a_j\right|^2$$

$$\text{meansq} \leftarrow \frac{1}{M \cdot N}\sum_{j=0}^{N-1}\sum_{i=0}^{M-1}\left|a_{i,j}\right|^2$$

Prototypes

```
vsip_scalar_f vsip_vmeansqval_f(
                      const vsip_vview_f *a);
vsip_scalar_f vsip_cvmeansqval_f(
                      const vsip_cvview_f *a);
vsip_scalar_f vsip_mmeansqval_f(
                      const vsip_mview_f *a);
vsip_scalar_f vsip_cmmeansqval_f(
                      const vsip_cmview_f *a);
```

Arguments

`*a` View of input vector/matrix

Return Value

Returns the mean of the squares of all the elements of a vector/matrix

Restrictions

Errors

The arguments must conform to the following:

1. The vector/matrix passed to the function must be defined and must not be *null*.

Notes/References

Examples

```
/* Example of meansqval */
#include<stdio.h>
#include "vsip.h"

#define L 7                    /* length */

int main()
{
    vsip_vview_d* data;

    vsip_init((void *)0);
    data  = vsip_vcreate_d(L, VSIP_MEM_NONE);

    /* Make up some data to find the mean square value of */
    /* Compute a ramp from zero to L-1 */
    vsip_vramp_d(0.0, 1.0, data);

    /* And find and print its mean square value */
    printf("%f \n",vsip_vmeansqval_d(data));
    /*destroy the vector views and any associated blocks */
```

```
    vsip_blockdestroy_d(vsip_vdestroy_d(data));
    vsip_finalize((void *)0);
    return 0;
}
/* output */
/* 13.000000 */
```

See Also

vsip_*ds*meanval_*p*, vsip_*ds*sumval_*p*, and vsip_*ds*sumsqval_*p*

vsip_*d*vmodulate_*p* Vector Modulate

Computes the modulation of a real vector by a specified complex frequency.

Functionality

$$r_k \leftarrow a_k \cdot \left(\cos\left(k \cdot \nu + \phi\right) + j \cdot \sin\left(k \cdot \nu + \phi\right)\right) = a_k \cdot e^{j \cdot (k \cdot \nu + \phi)} \qquad \text{for } k = 0, 1, \ldots, N\text{-}1$$

Where ν is the frequency in radians per index and ϕ is the initial phase.

Prototypes

```
vsip_scalar_f vsip_vmodulate_f(
                    const vsip_vview_f *a,
                    vsip_scalar_f nu,
                    vsip_scalar_f phi,
                    const vsip_cvview_f *r);
vsip_scalar_f vsip_cvmodulate_f(
                    const vsip_cvview_f *a,
                    vsip_scalar_f nu,
                    vsip_scalar_f phi,
                    const vsip_cvview_f *r);
```

Arguments

`*a`	View of input vector
`nu`	Scalar frequency in radians per index
`phi`	Scalar initial phase in radians.
`*r`	View of output vector

Return Value

Returns a value to be used as the initial phase argument for the next call of `vsip_vmodulate_f`. For a vector of length N, the return value would be $N\nu + \phi$.

Restrictions

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.
3. The input and output views must be identical views of the same block (in-place), or must not overlap.

Notes/References

The return value would be used as the initial phase ϕ in the next frame if the modulation function intended to be continuous but processed by frames.

Examples

```
/* Example of vmodulate */
#include<stdio.h>
#include "vsip.h"

#define PI 3.1415926535
#define L 7                    /* length */
#define M 6                    /* L-1 */

int main()
{
    int i, j;
    vsip_scalar_d nu = 1.0, phi = 0.0, start = 0.0;
    vsip_vview_d  *data;
    vsip_cvview_d *dataR;
    vsip_vview_d  *dataA;

    vsip_init((void *)0);
    data  = vsip_vcreate_d(L, VSIP_MEM_NONE);
    dataR = vsip_cvcreate_d(M, VSIP_MEM_NONE);
    dataA = vsip_vbind_d(vsip_vgetblock_d(data),0,1,M);

    /* Make up some data and modulate */
    vsip_vfill_d(0.0,data);
    printf("  input           output\n");
    for(i=0; i<3; i++){
       vsip_vramp_d(start,(1.65 * PI / (double) (M)), data);
       start = vsip_vget_d(data,M);
       vsip_vcos_d(dataA,dataA);
       phi = vsip_vmodulate_d(dataA, nu, phi, dataR);
       for(j = 0; j < M; j++)
           printf("%7.4f => ( %7.4f, %7.4f)\n",
                vsip_vget_d(dataA,j),
                vsip_real_d(vsip_cvget_d(dataR,j)),
                vsip_imag_d(vsip_cvget_d(dataR,j)));
     }
     vsip_vdestroy_d(dataA);
     vsip_cvalldestroy_d(dataR);
     vsip_valldestroy_d(data);
     vsip_finalize((void *)0);
     return 0;
}
/* output */
/*  input           output
   1.0000 => (  1.0000,  0.0000)
   0.6494 => (  0.3509,  0.5465)
  -0.1564 => (  0.0651, -0.1422)
   First three and last three results
   0.9239 => ( -0.7019,  0.6008)
   0.3090 => ( -0.2959, -0.0890)
  -0.5225 => (  0.1438,  0.5023) */
```

See Also

vsip_*ds*euler_*p*

vsip_*ds*neg_*p* Vector/Matrix Negate

Computes the negation for each element of a vector/matrix.

Functionality

$r_j \leftarrow -a_j$ for $j = 0, 1, \ldots, N\text{-}1$

$r_{i,j} \leftarrow -a_{i,j}$ for $i = 0, 1, \ldots, M\text{-}1$; for $j = 0, 1, \ldots, N\text{-}1$

Prototypes

```
void vsip_vneg_i(
                    const vsip_vview_i *a, ,
                    const vsip_vview_i *r);
void vsip_vneg_f(
                    const vsip_vview_f *a,
                    const vsip_vview_f *r);
void vsip_cvneg_f(
                    const vsip_cvview_f *a,
                    const vsip_cvview_f *r);
void vsip_mneg_i(
                    const vsip_mview_i *a, ,
                    const vsip_mview_i *r);
void vsip_mneg_f(
                    const vsip_mview_f *a,
                    const vsip_mview_f *r);
void vsip_cmneg_f(
                    const vsip_cmview_f *a,
                    const vsip_cmview_f *r);
```

Arguments

`*a`	View of input vector/matrix
`*r`	View of output vector/matrix

Return Value

None.

Restrictions

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.
3. The input and output views must be identical views of the same block (in-place), or must not overlap.

Notes/References

Examples

```
/* Example of negation */
#include<stdio.h>
#include "vsip.h"

#define PI 3.1415926535
#define L 7                    /* length */

int main()
{
    int i;
    vsip_cvview_d* dataCmplx;
    vsip_vview_d*  dataRe;
    vsip_vview_d*  dataIm;
    vsip_cvview_d* dataNeg;
    vsip_cscalar_d cscalar;
    vsip_cscalar_d cnegscalar;

    vsip_init((void *)0);
    dataCmplx = vsip_cvcreate_d(L, VSIP_MEM_NONE);
    dataRe    = vsip_vrealview_d(dataCmplx);
    dataIm    = vsip_vimagview_d(dataCmplx);
    dataNeg   = vsip_cvcreate_d(L, VSIP_MEM_NONE);
    /* Make up some data to find the negative of */
    /* First compute a ramp from zero to 2pi and apply sin */
    vsip_vramp_d(0.0, (2.0 * PI / (double) (L - 1)), dataRe);
    vsip_vsin_d(dataRe,dataRe);
    /* Then compute a ramp from zero to 3pi and apply sin */
    vsip_vramp_d(0.0, (3.0 * PI / (double) (L - 1)), dataIm);
    vsip_vcos_d(dataIm,dataIm);

    /* Find the negative */
    vsip_cvneg_d(dataCmplx,dataNeg);

    /*now print out dataCmplex and its negative*/
    for(i=0; i<L; i++){
        cscalar = vsip_cvget_d(dataCmplx, (vsip_scalar_vi) i);
        cnegscalar = vsip_cvget_d(dataNeg, (vsip_scalar_vi) i);
        printf("(%7.4f, %7.4f) => (%7.4f, %7.4f)\n",
             vsip_real_d(cscalar), vsip_imag_d(cscalar),
             vsip_real_d(cnegscalar), vsip_imag_d(cnegscalar));
    }
    vsip_vdestroy_d(dataRe); vsip_vdestroy_d(dataIm);
    vsip_cblockdestroy_d(vsip_cvdestroy_d(dataCmplx));
    vsip_cblockdestroy_d(vsip_cvdestroy_d(dataNeg));
    vsip_finalize((void *)0);
    return 0;
}
/* output */
/* ( 0.0000,  1.0000) => (-0.0000, -1.0000)
   ( 0.8660,  0.0000) => (-0.8660, -0.0000)
   ( 0.8660, -1.0000) => (-0.8660,  1.0000)
   ( 0.0000, -0.0000) => (-0.0000,  0.0000)
   (-0.8660,  1.0000) => ( 0.8660, -1.0000)
   (-0.8660,  0.0000) => ( 0.8660, -0.0000)
   (-0.0000, -1.0000) => ( 0.0000,  1.0000) */
```

See Also

`vsip_c`***s***`conj_`***p***, and `vsip_`***ds***`sub_`***p***

vsip_*ds*recip_*p* Vector/Matrix Reciprocal

Computes the reciprocal for each element of a vector/matrix.

Functionality

$$r_j \leftarrow \frac{1}{a_j} \qquad \text{for } j = 0, 1, \ldots, N\text{-}1$$

$$r_{i,j} \leftarrow \frac{1}{a_{i,j}} \qquad \text{for } i = 0, 1, \ldots, M\text{-}1;\ \text{for } j = 0, 1, \ldots, N\text{-}1$$

Prototypes

```
void vsip_vrecip_f(
                      const vsip_vview_f *a,
                      const vsip_vview_f *r);
void vsip_cvrecip_f(
                      const vsip_cvview_f *a,
                      const vsip_cvview_f *r);
void vsip_mrecip_f(
                      const vsip_mview_f *a,
                      const vsip_mview_f *r);
void vsip_cmrecip_f(
                      const vsip_cmview_f *a,
                      const vsip_cmview_f *r);
```

Arguments

`*a`	View of input vector/matrix
`*r`	View of output vector/matrix

Return Value

None.

Restrictions

The inverse of zero is not defined. The result of the reciprocal of zero is implementation dependent.

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.
3. The input and output views must be identical views of the same block (in-place), or must not overlap.

Notes/References

Examples

```
/* Example of Reciprocal */

#include<stdio.h>
#include "vsip.h"

#define L 7                    /* length */

int main()
{
    int i;
    vsip_cvview_d* dataCmplx;
    vsip_cvview_d* dataRecip;
    vsip_vview_d*  dataRe;
    vsip_vview_d*  dataIm;
    vsip_cscalar_d cscalar;
    vsip_cscalar_d crecipscalar;

    vsip_init((void *)0);
    dataCmplx = vsip_cvcreate_d(L, VSIP_MEM_NONE);
    dataRecip = vsip_cvcreate_d(L, VSIP_MEM_NONE);
    dataRe    = vsip_vrealview_d(dataCmplx);
    dataIm    = vsip_vimagview_d(dataCmplx);

    /* Make up some data to find the reciprocal  of */
    /* First compute a ramp from 1 to L */
    vsip_vramp_d(1.0, 1.0, dataRe);

    /* Then compute a ramp from 1 to 0 */
    vsip_vramp_d(1.0, -1.0/(double)(L-1), dataIm);

    /* Find the Reciprocal of dataCmplx*/
    vsip_cvrecip_d(dataCmplx,dataRecip);

    /*now print out dataCmplex and its reciprocal */
    for(i=0; i<L; i++){
        cscalar = vsip_cvget_d(dataCmplx, (vsip_scalar_vi) i);
        crecipscalar = vsip_cvget_d(dataRecip, (vsip_scalar_vi) i);
        printf("(%7.4f, %7.4f) => (%7.4f, %7.4f)\n",
             vsip_real_d(cscalar), vsip_imag_d(cscalar),
             vsip_real_d(crecipscalar), vsip_imag_d(crecipscalar));
    }
    /*destroy the vector views and any associated blocks */
    vsip_vdestroy_d(dataRe); vsip_vdestroy_d(dataIm);
    vsip_cblockdestroy_d(vsip_cvdestroy_d(dataCmplx));
    vsip_cblockdestroy_d(vsip_cvdestroy_d(dataRecip));
    vsip_finalize((void *)0);
    return 0;
}
/* output */
/* ( 1.0000,  1.0000) => ( 0.5000, -0.5000)
   ( 2.0000,  0.8333) => ( 0.4260, -0.1775)
   ( 3.0000,  0.6667) => ( 0.3176, -0.0706)
   ( 4.0000,  0.5000) => ( 0.2462, -0.0308)
   ( 5.0000,  0.3333) => ( 0.1991, -0.0133)
   ( 6.0000,  0.1667) => ( 0.1665, -0.0046)
```

```
    ( 7.0000,  0.0000) => ( 0.1429, -0.0000) */
```

See Also

`vsip_`***ds***`rsqrt_`***p***, and `vsip_`***ds***`div_`***p***

vsip_sround_*p*_*p* Vector/Matrix Round

For each element in the input view round to the nearest integral value.

Functionality

For each element in the input view round to the nearest integral value. If two integral values are equally near, round to the value whose least significant bit is zero.

Prototypes

```
void vsip_vround_f_f(
                     const vsip_vview_f *x,
                     const vsip_vview_f *r);
void vsip_vround_f_i(
                     const vsip_vview_f *x,
                     const vsip_vview_i *r);
void vsip_mround_f_f(
                     const vsip_mview_f *x,
                     const vsip_mview_f *r);
void vsip_mround_f_i(
                     const vsip_mview_f *x,
                     const vsip_mview_i *r);
```

Arguments

`*x`	View of input vector/matrix
`*r`	View of output vector/matrix

Return Value

None

Restrictions

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.
3. The input and output views must be identical views of the same block (in-place), or must not overlap.

Notes/References

Although there is no VSIPL requirement to fully support IEEE 754 the intent of this function is to support the *round to nearest* functionality as described in IEEE 754.

Examples

See Also

See also `vsip_sceil_p_p` and `vsip_sfloor_p_p`.

vsip_*ds*rsqrt_*p* Vector/Matrix Reciprocal Square Root

Computes the reciprocal of the square root for each element of a vector/matrix.

Functionality

$$r_j \leftarrow \frac{1}{\sqrt{a_j}} \qquad \text{for } j = 0, 1, \ldots, N\text{-}1$$

$$r_{i,j} \leftarrow \frac{1}{\sqrt{a_{i,j}}} \qquad \text{for } i = 0, 1, \ldots, M\text{-}1;\ \text{for } j = 0, 1, \ldots, N\text{-}1$$

Prototypes

```
void vsip_vrsqrt_f(
                      const vsip_vview_f *a,
                      const vsip_vview_f *r);
void vsip_mrsqrt_f(
                      const vsip_mview_f *a,
                      const vsip_mview_f *r);
```

Arguments

`*a`	View of input vector/matrix
`*r`	View of output vector/matrix

Return Value

None.

Restrictions

Input arguments less than or equal to zero are not in the domain of the function. The result is implementation dependent.

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.
3. The input and output views must be identical views of the same block (in-place), or must not overlap.

Notes/References

Examples

```
/* Example of inverse square root */

#include<stdio.h>
#include "vsip.h"

#define L 7                    /* length */

int main()
{
    int i;
    vsip_vview_d* data;
    vsip_vview_d* dataRsqrt;

    vsip_init((void *)0);
```

```
    data = vsip_vcreate_d(L, VSIP_MEM_NONE);
    dataRsqrt = vsip_vcreate_d(L, VSIP_MEM_NONE);

    /* Make up some data */
    /* Compute a ramp from 1 to L */
    vsip_vramp_d(1.0, 1.0, data);

    /* Find the inverse square root and print it */
    vsip_vrsqrt_d(data,dataRsqrt);
    for(i = 0; i < L; i++)
        printf("%f => %f\n",vsip_vget_d(data, i),
               vsip_vget_d(dataRsqrt, i));

    /*destroy the vector views and any associated blocks */
    vsip_blockdestroy_d(vsip_vdestroy_d(data));
    vsip_blockdestroy_d(vsip_vdestroy_d(dataRsqrt));
    vsip_finalize((void *)0);
    return 0;
}
/* output */
/* 1.000000 => 1.000000
   2.000000 => 0.707107
   3.000000 => 0.577350
   4.000000 => 0.500000
   5.000000 => 0.447214
   6.000000 => 0.408248
   7.000000 => 0.377964 */
```

See Also

vsip_*ds*recip_*p*, and vsip_*ds*div_*p*

vsip_*ds*sq_*p* Vector/Matrix Square

Computes the square for each element of a vector/matrix.

Functionality

$r_j \leftarrow \left(a_j\right)^2$ for $j = 0, 1, \ldots, N\text{-}1$

$r_{i,j} \leftarrow \left(a_{i,j}\right)^2$ for $i = 0, 1, \ldots, M\text{-}1$; for $j = 0, 1, \ldots, N\text{-}1$

Prototypes

```
void vsip_vsq_f(
                    const vsip_vview_f *a,
                    const vsip_vview_f *r);
void vsip_msq_f(
                    const vsip_mview_f *a,
                    const vsip_mview_f *r);
```

Arguments

`*a`	View of input vector/matrix
`*r`	View of output vector/matrix

Return Value

None.

Restrictions

Overflow will occur if an element's magnitude is greater than the square root of the maximum defined number. The result of an overflow is implementation dependent.

Underflow will occur if an element's magnitude is less than the square root of the minimum defined number. The result of an underflow is implementation dependent.

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.
3. The input and output views must be identical views of the same block (in-place), or must not overlap.

Notes/References

Examples

```
/* Example of square */
#include<stdio.h>
#include "vsip.h"

#define L 7                    /* length */

int main()
{
    int i;
    vsip_vview_d* data;
    vsip_vview_d* dataSq;

    vsip_init((void *)0);
    data   = vsip_vcreate_d(L, VSIP_MEM_NONE);
```

```
    dataSq = vsip_vcreate_d(L, VSIP_MEM_NONE);

    /* Make up some data to find the square of */
    /* Compute a ramp from 1 to L */
    vsip_vramp_d(1.0, 1.0, data);

    /* Find the square */
    vsip_vsq_d(data,dataSq);

    /* print the results */
    for(i = 0; i < L; i++)
        printf("%f => %f\n",vsip_vget_d(data, i), vsip_vget_d(dataSq, i));

    /*destroy the vector views and any associated blocks */
    vsip_blockdestroy_d(vsip_vdestroy_d(data));
    vsip_blockdestroy_d(vsip_vdestroy_d(dataSq));
    vsip_finalize((void *)0);
    return 0;
}
/* output */
/* 1.000000 => 1.000000
   2.000000 => 4.000000
   3.000000 => 9.000000
   4.000000 => 16.000000
   5.000000 => 25.000000
   6.000000 => 36.000000
   7.000000 => 49.000000 */
```

See Also

For complex, use `vsip_`*s*`cmagsq_`*p*.

vsip_*ds*sumval_*p* Vector/Matrix Sum Value

Returns the sum of the elements of a vector/matrix.

Functionality

$$\text{sum} \leftarrow \sum_{j=0}^{N-1} a_j$$

$$\text{sum} \leftarrow \sum_{j=0}^{N-1} \sum_{i=0}^{M-1} a_{i,j}$$

Prototypes

```
vsip_scalar_i vsip_vsumval_i(
                    const vsip_vview_i *a);
vsip_cscalar_i vsip_cvsumval_i(
                    const vsip_cvview_i *a);
vsip_scalar_i vsip_msumval_i(
                    const vsip_mview_i *a);
vsip_cscalar_i vsip_cmsumval_i(
                    const vsip_cmview_i *a);
vsip_scalar_f vsip_vsumval_f(
                    const vsip_vview_f *a);
vsip_cscalar_f vsip_cvsumval_f(
                    const vsip_cvview_f *a);
vsip_scalar_f vsip_msumval_f(
                    const vsip_mview_f *a);
vsip_cscalar_f vsip_cmsumval_f(
                    const vsip_cmview_f *a);
vsip_scalar_vi vsip_vsumval_bl(
                    const vsip_vview_bl *a);
vsip_scalar_vi vsip_msumval_bl(
                    const vsip_mview_bl *a);
```

Arguments

`*a`	View of input vector/matrix

Return Value

If the input is a Boolean type then the function returns an integer value of type `vsip_scalar_vi` consisting of the number of true values. Otherwise, it returns a scalar sum of the elements of the same type and precision as the input vector/matrix.

Restrictions

If an overflow occurs, the result is implementation dependent.

Errors

The arguments must conform to the following:

1. The vector/matrix passed to the function must be defined and must not be *null*.

Notes/References

A Boolean is defined so that zero is false and anything else is true. The boolean sumval function returns the number of non false values in a boolean object.

The return type of `vsip_scalar_vi` for the boolean case was chosen since it resolves to an unsigned integer type large enough to represent the size of a vector, matrix, or tensor. This allows us to portably specify the integer return type.

The order of summation is not specified, therefore significant numerical errors can potentially occur.

Examples

```
/* Example of sumval */
#include<stdio.h>
#include "vsip.h"

#define L 7                    /* length */

int main()
{
    vsip_vview_d* data;

    vsip_init((void *)0);
    data = vsip_vcreate_d(L, VSIP_MEM_NONE);

    /* Make up some data to find the sum of */
    /* Compute a ramp from zero to L-1 */
    vsip_vramp_d(0.0, 1.0, data);

    /* And find and print its sum */
    printf("%f \n", vsip_vsumval_d(data));

    /*destroy the vector views and any associated blocks */
    vsip_blockdestroy_d(vsip_vdestroy_d(data));
    vsip_finalize((void *)0);
    return 0;
}
/* output */
/* 21.000000 */
```

See Also

`vsip_`***ds***`meanval_`***p***, `vsip_`***ds***`meansqval_`***p***, and `vsip_`***ds***`sumsqval_`***p***

vsip_*ds*sumsqval_*p* Vector/Matrix Sum of Squares Value

Returns the sum of the squares of the elements of a vector/matrix.

Functionality

$$\text{sumsq} \leftarrow \sum_{j=0}^{N-1} \left(a_j\right)^2$$

$$\text{sumsq} \leftarrow \sum_{j=0}^{N-1} \sum_{i=0}^{M-1} \left(a_{i,j}\right)^2$$

Prototypes

```
vsip_scalar_f vsip_vsumsqval_f(
                    const vsip_vview_f *a);
vsip_scalar_f vsip_msumsqval_f(
                    const vsip_mview_f *a);
```

Arguments

`*a` View of input vector/matrix

Return Value

Returns the sum of the vector elements squared.

Restrictions

If an overflow occurs, the result is implementation dependent.

Errors

The arguments must conform to the following:

1. Arguments passed to the function must be a valid object and must not be *null*.

Notes/References

The order of summation is not specified, therefore significant numerical errors can potentially occur.

Examples

```
/* Example of sumsqval*/
#include<stdio.h>
#include "vsip.h"
#define L 7                  /* length */
int main()
{    vsip_vview_d* data;
    vsip_init((void *)0);
    data = vsip_vcreate_d(L, VSIP_MEM_NONE);
    /* Make up some data to find the sum of squares of */
    vsip_vramp_d(0.0, 1.0, data);
    printf("%f \n", vsip_vsumsqval_d(data));
    vsip_blockdestroy_d(vsip_vdestroy_d(data));
    vsip_finalize((void *)0);
    return 0;
}  /* output */
   /* 91.000000  */
```

See Also

`vsip_`*ds*`meanval_`*p*, `vsip_`*ds*`meansqval_`*p*, and `vsip_`*ds*`sumval_`*p*

Binary Operations

The following man pages represent operations requiring two inputs, either two vectors/matrices or a vector/matrix and a scalar for input.

vsip_*d*s**add**_*p*	Vector/Matrix Add
vsip_*d*s*s***add**_*p*	Scalar Vector/Matrix Add
vsip_*d*s**div**_*p*	Vector/Matrix Divide
vsip_*d*s*s***div**_*p*	Scalar Vector/Matrix Divide
vsip_*d*s**sdiv**_*p*	Vector/Matrix Scalar Divide
vsip_*d*s**expoavg**_*p*	Vector/Matrix Exponential Average
vsip_*s***hypot**_*p*	Vector/Matrix Hypotenuse
vsip_c*s***jmul**_*p*	Vector/Matrix Conjugate Multiply (Elementwise)
vsip_*d*s**mul**_*p*	Vector/Matrix Multiply (Elementwise)
vsip_*d*s**smul**_*p*	Scalar Vector/Matrix Multiply
vsip_*d*v*d***mmul**_*p*	Vector-Matrix Multiply (Elementwise)
vsip_*d*s**sub**_*p*	Vector/Matrix Subtract
vsip_*d*s*s***sub**_*p*	Scalar Vector/Matrix Subtract

vsip_*ds*add_*p* Vector/Matrix Add

Computes the sum, by element, of two vectors/matrices.

Functionality

$r_j \leftarrow a_j + b_j$ for $j = 0, 1, \ldots, N\text{-}1$

$r_{i,j} \leftarrow a_{i,j} + b_{i,j}$ for $i = 0, 1, \ldots, M\text{-}1$; for $j = 0, 1, \ldots, N\text{-}1$

Prototypes

```
void vsip_vadd_i(
                    const vsip_vview_i *a,
                    const vsip_vview_i *b,
                    const vsip_vview_i *r);
void vsip_vadd_f(
                    const vsip_vview_f *a,
                    const vsip_vview_f *b,
                    const vsip_vview_f *r);
void vsip_rcvadd_f(
                    const vsip_vview_f *a,
                    const vsip_cvview_f *b,
                    const vsip_cvview_f *r);
void vsip_cvadd_f(
                    const vsip_cvview_f *a,
                    const vsip_cvview_f *b,
                    const vsip_cvview_f *r);
void vsip_madd_i(
                    const vsip_mview_i *a,
                    const vsip_mview_i *b,
                    const vsip_mview_i *r);
void vsip_madd_f(
                    const vsip_mview_f *a,
                    const vsip_mview_f *b,
                    const vsip_mview_f *r);
void vsip_rcmadd_f(
                    const vsip_mview_f *a,
                    const vsip_cmview_f *b,
                    const vsip_cmview_f *r);
void vsip_cmadd_f(
                    const vsip_cmview_f *a,
                    const vsip_cmview_f *b,
                    const vsip_cmview_f *r);
```

Arguments

`*a`	View of input vector/matrix
`*b`	View of input vector/matrix
`*r`	View of output vector/matrix

Return Value

None.

Restrictions

If an overflow occurs, the result is implementation dependent.

In the case of a mixed data type, for instance a real vector added to a complex vector, in-place implies the real input may be a real or imaginary view of the output. Input views which encompass both real and imaginary segments of the output, or which do not exactly overlay the real or imaginary view of the output, are not defined for in-place.

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.
3. The input and output views must be identical views of the same block (in-place), or must not overlap.

Notes/References

Examples

```
/* Example of vector add */
#include<stdio.h>
#include"vsip.h"

#define L 7                    /* length */

int main()
{
    int i;
    vsip_vview_d* dataLeft;
    vsip_vview_d* dataRight;
    vsip_vview_d* dataSum;

    vsip_init((void *)0);
    dataLeft  = vsip_vcreate_d(L, VSIP_MEM_NONE);
    dataRight = vsip_vcreate_d(L, VSIP_MEM_NONE);
    dataSum   = vsip_vcreate_d(L, VSIP_MEM_NONE);

    /* Make up some data to find the magnitude of */
    vsip_vramp_d(1.0, 1.0, dataLeft);
    vsip_vramp_d(1.0, -2.0/(double)(L-1), dataRight);

    /* Add the vectors */
    vsip_vadd_d(dataLeft, dataRight, dataSum);

    /* now print out the data and its sum */
    for(i=0; i<L; i++)
        printf("%7.4f = (%7.4f) + (%7.4f) \n", vsip_vget_d(dataSum,i),
             vsip_vget_d(dataLeft,i),vsip_vget_d(dataRight,i));
```

```
    /* destroy the vector views and any associated blocks */
    vsip_blockdestroy_d(vsip_vdestroy_d(dataLeft));
    vsip_blockdestroy_d(vsip_vdestroy_d(dataRight));
    vsip_blockdestroy_d(vsip_vdestroy_d(dataSum));
    vsip_finalize((void *)0);
    return 0;
}
/* output */
/* 2.0000 = ( 1.0000) + ( 1.0000)
   2.6667 = ( 2.0000) + ( 0.6667)
   3.3333 = ( 3.0000) + ( 0.3333)
   4.0000 = ( 4.0000) + ( 0.0000)
   4.6667 = ( 5.0000) + (-0.3333)
   5.3333 = ( 6.0000) + (-0.6667)
   6.0000 = ( 7.0000) + (-1.0000) */
```

See Also

vsip_*d*s*s*add_*p*

vsip_*dss*add_*p* Scalar Vector/Matrix Add

Computes the sum, by element, of a scalar and a vector/matrix.

Functionality

$r_j \leftarrow \alpha + b_j$ for $j = 0, 1, \ldots, N\text{-}1$

$r_{i,j} \leftarrow \alpha + b_{i,j}$ for $i = 0, 1, \ldots, M\text{-}1$; for $j = 0, 1, \ldots, N\text{-}1$

Prototypes

```
void vsip_svadd_i(
                    vsip_scalar_i alpha,
                    const vsip_vview_i *b,
                    const vsip_vview_i *r);
void vsip_svadd_f(
                    vsip_scalar_f alpha,
                    const vsip_vview_f *b,
                    const vsip_vview_f *r);
void vsip_rscvadd_f(
                    vsip_scalar_f alpha,
                    const vsip_cvview_f *b,
                    const vsip_cvview_f *r);
void vsip_csvadd_f(
                    vsip_cscalar_f alpha,
                    const vsip_cvview_f *b,
                    const vsip_cvview_f *r);
void vsip_smadd_i(
                    vsip_scalar_i alpha,
                    const vsip_mview_i *b,
                    const vsip_mview_i *r);
void vsip_smadd_f(
                    vsip_scalar_f alpha,
                    const vsip_mview_f *b,
                    const vsip_mview_f *r);
void vsip_rscmadd_f(
                    vsip_scalar_f alpha,
                    const vsip_cmview_f *b,
                    const vsip_cmview_f *r);
void vsip_csmadd_f(
                    vsip_cscalar_f alpha,
                    const vsip_cmview_f *b,
                    const vsip_cmview_f *r);
```

Arguments

`alpha`	Input scalar
`*b`	View of input vector/matrix
`*r`	View of output vector/matrix

Return Value

None.

Restrictions

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.
3. The input and output views must be identical views of the same block (in-place), or must not overlap.

Notes/References

Examples

```
/* Example of scalar vector add */
#include<stdio.h>
#include "vsip.h"

#define L 7                    /* length */

int main()
{
    int i;
    vsip_scalar_d dataLeft;
    vsip_vview_d* dataRight;
    vsip_vview_d* dataSum;

    vsip_init((void *)0);
    dataRight = vsip_vcreate_d(L, VSIP_MEM_NONE);
    dataSum   = vsip_vcreate_d(L, VSIP_MEM_NONE);

    /* Make up some data to find the magnitude of */
    /* First set the scalar equal to 1*/
    dataLeft = 1.0;

    /* Then compute a ramp from one to minus one */
    vsip_vramp_d(1.0, -2.0/(double)(L-1), dataRight);

    /* Add the scalar and the vector */
    vsip_svadd_d(dataLeft, dataRight, dataSum);

    /* now print out the data and its sum */
    for(i=0; i<L; i++)
        printf("%7.4f = (%7.4f) + (%7.4f) \n", vsip_vget_d(dataSum,i),
             dataLeft, vsip_vget_d(dataRight,i));

    /* destroy the vector views and any associated blocks */
    vsip_blockdestroy_d(vsip_vdestroy_d(dataRight));
    vsip_blockdestroy_d(vsip_vdestroy_d(dataSum));
```

```
    vsip_finalize((void *)0);
    return 0;
}
/* output */
/* 2.0000 = ( 1.0000) + ( 1.0000)
   1.6667 = ( 1.0000) + ( 0.6667)
   1.3333 = ( 1.0000) + ( 0.3333)
   1.0000 = ( 1.0000) + ( 0.0000)
   0.6667 = ( 1.0000) + (-0.3333)
   0.3333 = ( 1.0000) + (-0.6667)
   0.0000 = ( 1.0000) + (-1.0000) */
```

See Also

vsip_*ds*add_*p*

vsip_*ds*div_*p* Vector/Matrix Divide

Computes the quotient, by element, of two vectors/matrices.

Functionality

$$r_j \leftarrow \frac{a_j}{b_j} \qquad \text{for } j = 0, 1, \ldots, N\text{-}1$$

$$r_{i,j} \leftarrow \frac{a_{i,j}}{b_{i,j}} \qquad \text{for } i = 0, 1, \ldots, M\text{-}1;\ \text{for } j = 0, 1, \ldots, N\text{-}1$$

Prototypes

```
void vsip_vdiv_f(
                    const vsip_vview_f *a,
                    const vsip_vview_f *b,
                    const vsip_vview_f *r);
void vsip_rcvdiv_f(
                    const vsip_vview_f *a,
                    const vsip_cvview_f *b,
                    const vsip_cvview_f *r);
void vsip_crvdiv_f(
                    const vsip_cvview_f *a,
                    const vsip_vview_f *b,
                    const vsip_cvview_f *r);
void vsip_cvdiv_f(
                    const vsip_cvview_f *a,
                    const vsip_cvview_f *b,
                    const vsip_cvview_f *r);
void vsip_mdiv_f(
                    const vsip_mview_f *a,
                    const vsip_mview_f *b,
                    const vsip_mview_f *r);
void vsip_rcmdiv_f(
                    const vsip_mview_f *a,
                    const vsip_cmview_f *b,
                    const vsip_cmview_f *r);
void vsip_crmdiv_f(
                    const vsip_cmview_f *a,
                    const vsip_mview_f *b,
                    const vsip_cmview_f *r);
void vsip_cmdiv_f(
                    const vsip_cmview_f *a,
                    const vsip_cmview_f *b,
                    const vsip_cmview_f *r);
```

Arguments

`*a`	View of input vector/matrix
`*b`	View of input vector/matrix
`*r`	View of output vector/matrix

Return Value

None.

Restrictions

Results of division by zero are implementation dependent.

Overflows and underflows are possible. Results are implementation dependent.

In the case of a mixed data type, for instance a real vector divided by a complex vector, in-place implies the real input may be a real or imaginary view of the output. Input views which encompass both real and imaginary segments of the output, or which do not exactly overlay the real or imaginary view of the output, are not defined for in-place.

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.
3. The input and output views must be identical views of the same block (in-place), or must not overlap.

Notes/References

Examples

```
/* example of vector-vector divide */
#include<stdio.h>
#include "vsip.h"

#define L 5

int main()
{  int i;
   /* define some vectors */
   vsip_vview_d*  dataReOne;
   vsip_vview_d*  dataReTwo;
   vsip_vview_d*  dataReQuotient;
   vsip_cvview_d* dataComplex;
   vsip_cvview_d* dataComplexQuotient;

   vsip_init((void *)0);
   dataReOne = vsip_vcreate_d(L, VSIP_MEM_NONE);
   dataReTwo = vsip_vcreate_d(L, VSIP_MEM_NONE);
   dataReQuotient = vsip_vcreate_d(L, VSIP_MEM_NONE);
   dataComplex = vsip_cvcreate_d(L, VSIP_MEM_NONE);
   dataComplexQuotient = vsip_cvcreate_d(L, VSIP_MEM_NONE);

   /* make up some data */
   vsip_vramp_d(1,1,dataReOne);
   vsip_vfill_d(2,dataReTwo);
   vsip_vcmplx_d(dataReTwo,dataReOne,dataComplex);
```

```
    /*divide one by two and print the input and output */
    vsip_vdiv_d(dataReOne,dataReTwo,dataReQuotient);
    for(i=0; i<L; i++)
        printf("%7.4f / %7.4f = %7.4f\n",
               vsip_vget_d(dataReOne,i), vsip_vget_d(dataReTwo,i),
               vsip_vget_d(dataReQuotient,i)); printf("\n");

    /*divide one by complex and print the input and output */
    vsip_rcvdiv_d(dataReOne,dataComplex,dataComplexQuotient);
    for(i=0; i<L; i++)
        printf("%7.4f / (%7.4f + %7.4fi) = (%7.4f + %7.4fi)\n",
               vsip_vget_d(dataReOne,i),
               vsip_real_d(vsip_cvget_d(dataComplex,i)),
               vsip_imag_d(vsip_cvget_d(dataComplex,i)),
               vsip_real_d(vsip_cvget_d(dataComplexQuotient,i)),
               vsip_imag_d(vsip_cvget_d(dataComplexQuotient,i)));

    /* destroy created objects */
    vsip_blockdestroy_d(vsip_vdestroy_d(dataReOne));
    vsip_blockdestroy_d(vsip_vdestroy_d(dataReTwo));
    vsip_blockdestroy_d(vsip_vdestroy_d(dataReQuotient));
    vsip_cblockdestroy_d(vsip_cvdestroy_d(dataComplex));
    vsip_cblockdestroy_d(vsip_cvdestroy_d(dataComplexQuotient));
    vsip_finalize((void *)0);
    return 0;
}
/* output */
/* 1.0000 /  2.0000 =  0.5000
   2.0000 /  2.0000 =  1.0000
   3.0000 /  2.0000 =  1.5000
   4.0000 /  2.0000 =  2.0000
   5.0000 /  2.0000 =  2.5000
   1.0000 / ( 2.0000 +  1.0000i) = ( 0.4000 + -0.2000i)
   2.0000 / ( 2.0000 +  2.0000i) = ( 0.5000 + -0.5000i)
   3.0000 / ( 2.0000 +  3.0000i) = ( 0.4615 + -0.6923i)
   4.0000 / ( 2.0000 +  4.0000i) = ( 0.4000 + -0.8000i)
   5.0000 / ( 2.0000 +  5.0000i) = ( 0.3448 + -0.8621i) */
```

See Also

vsip_*d*s*s*div_*p*, and vsip_*ds*sdiv_*p*

vsip_*d*ss*div*_*p* Scalar Vector/Matrix Divide

Computes the quotient, by element, of a scalar and a vector/matrix.

Functionality

$$r_j \leftarrow \frac{\alpha}{b_j} \qquad \text{for } j = 0, 1, \ldots, N\text{-}1$$

$$r_{i,j} \leftarrow \frac{\alpha}{b_{i,j}} \qquad \text{for } i = 0, 1, \ldots, M\text{-}1;\ \text{for } j = 0, 1, \ldots, N\text{-}1$$

Prototypes

```
void vsip_svdiv_f(
                      vsip_scalar_f alpha,
                      const vsip_vview_f *b,
                      const vsip_vview_f *r);
void vsip_rscvdiv_f(
                      vsip_scalar_f alpha,
                      const vsip_cvview_f *b,
                      const vsip_cvview_f *r);
void vsip_csvdiv_f(
                      vsip_cscalar_f alpha,
                      const vsip_cvview_f *b,
                      const vsip_cvview_f *r);
void vsip_smdiv_f(
                      vsip_scalar_f alpha,
                      const vsip_mview_f *b,
                      const vsip_mview_f *r);
void vsip_rscmdiv_f(
                      vsip_scalar_f alpha,
                      const vsip_cmview_f *b,
                      const vsip_cmview_f *r);
void vsip_csmdiv_f(
                      vsip_cscalar_f alpha,
                      const vsip_cmview_f *b,
                      const vsip_cmview_f *r);
```

Arguments

`alpha`	Input scalar
`*b`	View of input vector/matrix
`*r`	View of output vector/matrix

Return Value

None.

Restrictions

The result of division by zero is implementation dependent.

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.
3. The input and output views must be identical views of the same block (in-place), or must not overlap.

Notes/References

Examples

```
/* example of scalar-vector divide */
#include<stdio.h>
#include "vsip.h"
#define L 5
int main()
{  int i;
   vsip_cvview_d* dataComplex;
   vsip_cscalar_d scalarComplex;
   vsip_cvview_d* dataComplexQuotient;

   vsip_init((void *)0);
   dataComplex = vsip_cvcreate_d(L, VSIP_MEM_NONE);
   dataComplexQuotient = vsip_cvcreate_d(L, VSIP_MEM_NONE);

   /* put some complex data in dataComplex */
   for(i = 0; i < L; i++)
       vsip_cvput_d(dataComplex,i,vsip_cmplx_d((double)(i * i),
                    (double)(i + 1)));

   /* define a complex scalar */
   scalarComplex = vsip_cmplx_d(3,4);

   /*divide scalarComplex by dataComplex and print the input and output */
   vsip_csvdiv_d(scalarComplex, dataComplex, dataComplexQuotient);
   for(i=0; i<L; i++)
       printf("(%7.4f + %7.4fi) / (%7.4f + %7.4fi) = (%7.4f + %7.4fi)\n",
              vsip_real_d(scalarComplex),
              vsip_imag_d(scalarComplex),
              vsip_real_d(vsip_cvget_d(dataComplex,i)),
              vsip_imag_d(vsip_cvget_d(dataComplex,i)),
              vsip_real_d(vsip_cvget_d(dataComplexQuotient,i)),
              vsip_imag_d(vsip_cvget_d(dataComplexQuotient,i)));
   vsip_cblockdestroy_d(vsip_cvdestroy_d(dataComplex));
   vsip_cblockdestroy_d(vsip_cvdestroy_d(dataComplexQuotient));
   vsip_finalize((void *)0);
   return 0;
}  /* output */
/* ( 3.0000 +  4.0000i) / ( 0.0000 +  1.0000i) = ( 4.0000 + -3.0000i)
   ( 3.0000 +  4.0000i) / ( 1.0000 +  2.0000i) = ( 2.2000 + -0.4000i)
   ( 3.0000 +  4.0000i) / ( 4.0000 +  3.0000i) = ( 0.9600 +  0.2800i)
   ( 3.0000 +  4.0000i) / ( 9.0000 +  4.0000i) = ( 0.4433 +  0.2474i)
   ( 3.0000 +  4.0000i) / (16.0000 +  5.0000i) = ( 0.2420 +  0.1744i) */
```

See Also

vsip_*ds*div_*p*, and vsip_*ds*sdiv_*p*

vsip_*ds*sdiv_*p* Vector/Matrix Scalar Divide

Computes the quotient, by element, of a vector/matrix and a scalar.

Functionality

$$r_j \leftarrow \frac{a_j}{\beta} \qquad \text{for } j = 0, 1, \ldots, N\text{-}1$$

$$r_{i,j} \leftarrow \frac{a_{i,j}}{\beta} \qquad \text{for } i = 0, 1, \ldots, M\text{-}1;\ \text{for } j = 0, 1, \ldots, N\text{-}1$$

Prototypes

```
void vsip_vsdiv_f(
                      const vsip_vview_f *a,
                      vsip_scalar_f beta,
                      const vsip_vview_f *r);
void vsip_cvrsdiv_f(
                      const vsip_cvview_f *a,
                      vsip_scalar_f beta,
                      const vsip_cvview_f *r);
void vsip_msdiv_f(
                      const vsip_mview_f *a,
                      vsip_scalar_f beta,
                      const vsip_mview_f *r);
void vsip_cmrsdiv_f(
                      const vsip_cmview_f *a,
                      vsip_scalar_f beta,
                      const vsip_cmview_f *r);
```

Arguments

`*a`	View of input vector/matrix
`*beta`	Input scalar
`*r`	View of output vector/matrix

Return Value

None.

Restrictions

Division by zero is not defined and the result is implementation specific.

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.
3. The input and output views must be identical views of the same block (in-place), or must not overlap.

Notes/References

This function was included for those who require increased accuracy when doing a divide. It is recommended that `vsip_dssmul_p` be used if increased accuracy is not required. The increased accuracy of using vector/matrix scalar divide is implementation dependent.

Examples

```
/* example of vector-scalar divide */
#include<stdio.h>
#include "vsip.h"

#define L 5
#define denom (vsip_scalar_d)10

int main()
{ int i;
  /* define some data space */
  vsip_cvview_d* dataComplex;
  vsip_cvview_d* dataComplexQuotient;

  vsip_init((void *)0);
  dataComplex = vsip_cvcreate_d(L, VSIP_MEM_NONE);
  dataComplexQuotient = vsip_cvcreate_d(L, VSIP_MEM_NONE);

  /* put some complex data in dataComplex */
  for(i = 0; i < L; i++)
      vsip_cvput_d(dataComplex,i,
                   vsip_cmplx_d((double)(i * i),(double)(i+1)));

  /*divide dataComplex by some denom and print the input and output */
  vsip_cvrsdiv_d(dataComplex,denom,dataComplexQuotient);
  for(i=0; i<L; i++)
     printf("(%7.4f + %7.4fi) /  %7.4f) = (%7.4f + %7.4fi)\n",
              vsip_real_d(vsip_cvget_d(dataComplex,i)),
              vsip_imag_d(vsip_cvget_d(dataComplex,i)),
              denom,
              vsip_real_d(vsip_cvget_d(dataComplexQuotient,i)),
              vsip_imag_d(vsip_cvget_d(dataComplexQuotient,i)));
   vsip_cblockdestroy_d(vsip_cvdestroy_d(dataComplex));
   vsip_cblockdestroy_d(vsip_cvdestroy_d(dataComplexQuotient));
   vsip_finalize((void *)0);
   return 0;
}
/* output */
/* ( 0.0000 +  1.0000i) /  10.0000) = ( 0.0000 +  0.1000i)
   ( 1.0000 +  2.0000i) /  10.0000) = ( 0.1000 +  0.2000i)
   ( 4.0000 +  3.0000i) /  10.0000) = ( 0.4000 +  0.3000i)
   ( 9.0000 +  4.0000i) /  10.0000) = ( 0.9000 +  0.4000i)
   (16.0000 +  5.0000i) /  10.0000) = ( 1.6000 +  0.5000i) */
```

See Also

The function `vsip_dssmul_p` is recommended for most cases where the multiplying scalar is the inverse of the divisor of `vsip_dssdiv_p`.

`vsip_dssmul_p`, `vsip_dsdiv_p`, and `vsip_dssdiv_p`

vsip_*ds*expoavg_*p* Vector/Matrix Exponential Average

Computes an exponential weighted average, by element, of two vectors/matrices.

Functionality

$c_j \leftarrow \alpha \cdot b_j + (1-\alpha) \cdot c_j$ for $j = 0, 1, \ldots, N-1$

$c_{i,j} \leftarrow \alpha \cdot b_{i,j} + (1-\alpha) \cdot c_{i,j}$ for $i = 0, 1, \ldots, M-1$; for $j = 0, 1, \ldots, N-1$

where α is the weighting factor.

Prototypes

```
void vsip_vexpoavg_f(
                     vsip_scalar_f alpha,
                     const vsip_vview_f *b,
                     const vsip_vview_f *c);
void vsip_mexpoavg_f(
                     vsip_scalar_f alpha,
                     const vsip_mview_f *b,
                     const vsip_mview_f *c);
void vsip_cvexpoavg_f(
                     vsip_scalar_f alpha,
                     const vsip_cvview_f *b,
                     const vsip_cvview_f *c);
void vsip_cmexpoavg_f(
                     vsip_scalar_f alpha,
                     const vsip_cmview_f *b,
                     const vsip_cmview_f *c);
```

Arguments

`alpha`	Scalar weighting factor
`*b`	View of input vector/matrix
`*c`	View of input/output vector/matrix

Return Value

None.

Restrictions

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.
3. The input and output views must be identical views of the same block (in-place), or must not overlap.

Notes/References

If this function is called L times with input vectors $\mathbf{b}_l$ (and recursively with vector $\mathbf{c}$) and the weight is set equal to the inverse of the iteration number l, ($\alpha = 1/l$), then the result will be an average of the vectors $\mathbf{c} = \frac{1}{L}\sum_{l=1}^{L}\mathbf{b}_l$.

Examples

```
/* example of expoavg */
#include<stdio.h>
#include "vsip.h"

#define L 7               /* A length */

int main()
{
    int i = 0, j=0;
    vsip_scalar_d alpha = 0;
    vsip_vview_d* dataB;
    vsip_vview_d* dataC;
    char out[5][7][8];

    vsip_init((void *)0);
    dataB = vsip_vcreate_d(L, VSIP_MEM_NONE);
    dataC = vsip_vcreate_d(L, VSIP_MEM_NONE);

    /* make up some data and average it*/
    vsip_vfill_d(0.0,dataC);
    for (j = 0; j < 5; j++){
      vsip_vramp_d(0.0,(vsip_scalar_d) j ,dataB);
      alpha = (vsip_scalar_d)(1.0 / (double)(j + 1.0));
      vsip_vexpoavg_d(alpha, dataB, dataC);
      for(i = 0; i < L; i++)
          sprintf(out[j][i],"%7.4f", vsip_vget_d(dataB,i));
    }
    /*print it out */
    for(i = 0; i < L; i++){
       printf("(");
       for(j = 0; j< 4; j++)
          printf("%s + ",out[j][i]);
       printf("%s ) / 5.0 = %7.4f \n",out[4][i], vsip_vget_d(dataC,i));
    }
    vsip_valldestroy_d(dataB);
    vsip_valldestroy_d(dataC);
    vsip_finalize((void *)0);
    return 0;
}
/* output */
/* ( 0.0000 +  0.0000 +  0.0000 +  0.0000 +  0.0000 ) / 5.0 =  0.0000
   ( 0.0000 +  1.0000 +  2.0000 +  3.0000 +  4.0000 ) / 5.0 =  2.0000
   ( 0.0000 +  2.0000 +  4.0000 +  6.0000 +  8.0000 ) / 5.0 =  4.0000
   ( 0.0000 +  3.0000 +  6.0000 +  9.0000 + 12.0000 ) / 5.0 =  6.0000
   ( 0.0000 +  4.0000 +  8.0000 + 12.0000 + 16.0000 ) / 5.0 =  8.0000
   ( 0.0000 +  5.0000 + 10.0000 + 15.0000 + 20.0000 ) / 5.0 = 10.0000
   ( 0.0000 +  6.0000 + 12.0000 + 18.0000 + 24.0000 ) / 5.0 = 12.0000 */
```

See Also

vsip_shypot_*p* Vector/Matrix Hypotenuse

Computes the square root of the sum of squares, by element, of two input vectors/matrices.

Functionality

$r_j \leftarrow \sqrt{(a_j)^2 + (b_j)^2}$ for $j = 0, 1, \ldots, N\text{-}1$

$r_{i,j} \leftarrow \sqrt{(a_{i,j})^2 + (b_{i,j})^2}$ for $i = 0, 1, \ldots, M\text{-}1$; for $j = 0, 1, \ldots, N\text{-}1$

Prototypes

```
void vsip_vhypot_f(
                      const vsip_vview_f *a,
                      const vsip_vview_f *b,
                      const vsip_vview_f *r);
void vsip_mhypot_f(
                      const vsip_mview_f *a,
                      const vsip_mview_f *b,
                      const vsip_mview_f *r);
```

Arguments

`*a`	View of input vector/matrix
`*b`	View of input vector/matrix
`*r`	View of output vector/matrix

Return Value

None.

Restrictions

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.
3. The input and output views must be identical views of the same block (in-place), or must not overlap.

Notes/References

The implementation shall ensure that intermediate overflows do not occur.

Examples

```
/* Example of hypot function */
#include<stdio.h>
#include"vsip.h"

#define L 7                    /* length */

int main()
{
    int i;
    vsip_vview_d* dataA;
    vsip_vview_d* dataB;
    vsip_vview_d* dataHypot;
```

```
    vsip_init((void *)0);
    dataA     = vsip_vcreate_d(L, VSIP_MEM_NONE);
    dataB     = vsip_vcreate_d(L, VSIP_MEM_NONE);
    dataHypot = vsip_vcreate_d(L, VSIP_MEM_NONE);

    /* Make up some data to use */
    vsip_vramp_d(1.0, 1.0 , dataA);
    vsip_vramp_d(1.0, -2.0/(double)(L-1), dataB);

    /* Now calculate the hypotenuse of A & B */
    vsip_vhypot_d(dataA,dataB,dataHypot);

    /* and print out the data and the Result */
    for(i=0; i<L; i++){
        printf("hypot(%7.4f,  %7.4f) =>  %7.4f\n",
                  vsip_vget_d(dataA,i),
                  vsip_vget_d(dataB,i),
                  vsip_vget_d(dataHypot,i));
    }
    /* and don't forget to recover memory */
    vsip_blockdestroy_d(vsip_vdestroy_d(dataA));
    vsip_blockdestroy_d(vsip_vdestroy_d(dataB));
    vsip_blockdestroy_d(vsip_vdestroy_d(dataHypot));
    vsip_finalize((void *)0);
    return 0;
}
/* output */
/* hypot( 1.0000,   1.0000) =>   1.4142
   hypot( 2.0000,   0.6667) =>   2.1082
   hypot( 3.0000,   0.3333) =>   3.0185
   hypot( 4.0000,   0.0000) =>   4.0000
   hypot( 5.0000,  -0.3333) =>   5.0111
   hypot( 6.0000,  -0.6667) =>   6.0369
   hypot( 7.0000,  -1.0000) =>   7.0711 */
```

See Also

This is a companion function for `vsip_`***s***`atan2_`***p***.

vsip_csjmul_*p* Vector/Matrix Conjugate Multiply (Elementwise)

Computes the product of a complex vector/matrix with the conjugate of a second complex vector/matrix, by element.

Functionality

$r_j \leftarrow a_j \cdot b_j^*$ for $j = 0, 1, \ldots, N\text{-}1$

$r_{i,j} \leftarrow a_{i,j} \cdot b_{i,j}^*$ for $i = 0, 1, \ldots, M\text{-}1$; for $j = 0, 1, \ldots, N\text{-}1$

Prototypes

```
void vsip_cvjmul_f(
                      const vsip_cvview_f *a,
                      const vsip_cvview_f *b,
                      const vsip_cvview_f *r);
void vsip_cmjmul_f(
                      const vsip_cmview_f *a,
                      const vsip_cmview_f *b,
                      const vsip_cmview_f *r);
```

Arguments

`*a`	View of input vector/matrix
`*b`	View of conjugate multiplier input vector/matrix.
`*r`	View of complex output vector/matrix

Return Value

None.

Restrictions

Results of underflows or overflows are implementation dependent.

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.
3. The input and output views must be identical views of the same block (in-place), or must not overlap.

Notes/References

Examples

```
/* Example of complex conjugate multiply */
#include<stdio.h>
#include "vsip.h"

#define L 7                    /* length */

int main()
{    int i;
    vsip_cvview_d* dataLeft;
    vsip_cvview_d* dataRight;
    vsip_cvview_d* dataJmul;
    vsip_vview_d*  reView;
    vsip_vview_d*  imView;
```

```
    vsip_init((void *)0);
    dataLeft  = vsip_cvcreate_d(L, VSIP_MEM_NONE);
    dataRight = vsip_cvcreate_d(L, VSIP_MEM_NONE);
    dataJmul  = vsip_cvcreate_d(L, VSIP_MEM_NONE);

    /* Make up some data for the left vector */
    reView = vsip_vrealview_d(dataLeft);
    imView = vsip_vimagview_d(dataLeft);
    vsip_vramp_d(1.0, 1.0 , reView);
    vsip_vramp_d(1.0, -2.0/(double)(L-1), imView);

    /* remember to destroy the views before making new ones */
    vsip_vdestroy_d(reView);
    vsip_vdestroy_d(imView);
    reView = vsip_vrealview_d(dataRight);
    imView = vsip_vimagview_d(dataRight);

    /* Make up some data for the right vector */
    vsip_vramp_d(-1.0, -1.0 , reView);
    vsip_vramp_d(2.0, -3.0/(double)(L-1), imView);

    /* Conjugate Multiply the vectors */
    vsip_cvjmul_d(dataLeft, dataRight, dataJmul);

    /*now print out the data and its sum*/
    for(i=0; i<L; i++)
        printf("(%7.3f + %7.3fi) * conj(%7.3f + %7.3fi) =>"
               " (%7.3f + %7.3fi)\n",
               vsip_real_d(vsip_cvget_d(dataLeft,i)),
               vsip_imag_d(vsip_cvget_d(dataLeft,i)),
               vsip_real_d(vsip_cvget_d(dataRight,i)),
               vsip_imag_d(vsip_cvget_d(dataRight,i)),
               vsip_real_d(vsip_cvget_d(dataJmul,i)),
               vsip_imag_d(vsip_cvget_d(dataJmul,i)));
    vsip_vdestroy_d(reView); vsip_vdestroy_d(imView);
    vsip_cblockdestroy_d(vsip_cvdestroy_d(dataLeft));
    vsip_cblockdestroy_d(vsip_cvdestroy_d(dataRight));
    vsip_cblockdestroy_d(vsip_cvdestroy_d(dataJmul));
    vsip_finalize((void *)0);
    return 0;
}
/* output */
/*( 1.000 +   1.000i) * conj( -1.000 +   2.000i) => (  1.000 +  -3.000i)
  ( 2.000 +   0.667i) * conj( -2.000 +   1.500i) => ( -3.000 +  -4.333i)
  ( 3.000 +   0.333i) * conj( -3.000 +   1.000i) => ( -8.667 +  -4.000i)
  ( 4.000 +   0.000i) * conj( -4.000 +   0.500i) => (-16.000 +  -2.000i)
  ( 5.000 +  -0.333i) * conj( -5.000 +   0.000i) => (-25.000 +   1.667i)
  ( 6.000 +  -0.667i) * conj( -6.000 +  -0.500i) => (-35.667 +   7.000i)
  ( 7.000 +  -1.000i) * conj( -7.000 +  -1.000i) => (-48.000 +  14.000i)*/
```

See Also

vsip_*ds*mul_*p*, vsip_*d*s*s*mul_*p*, and vsip_*d*v*d*mmul_*p*

vsip_*ds*mul_*p* Vector/Matrix Multiply (Elementwise)

Computes the product, by element, of two vectors/matrices.

Functionality

$r_j \leftarrow a_j \cdot b_j$ for $j = 0, 1, \ldots, N\text{-}1$

$r_{i,j} \leftarrow a_{i,j} \cdot b_{i,j}$ for $i = 0, 1, \ldots, M\text{-}1$; for $j = 0, 1, \ldots, N\text{-}1$

Prototypes

```
void vsip_vmul_i(
                      const vsip_vview_i *a,
                      const vsip_vview_i *b,
                      const vsip_vview_i *r);
void vsip_vmul_f(
                      const vsip_vview_f *a,
                      const vsip_vview_f *b,
                      const vsip_vview_f *r);
void vsip_rcvmul_f(
                      const vsip_vview_f *a,
                      const vsip_cvview_f *b,
                      const vsip_cvview_f *r);
void vsip_cvmul_f(
                      const vsip_cvview_f *a,
                      const vsip_cvview_f *b,
                      const vsip_cvview_f *r);
void vsip_mmul_i(
                      const vsip_mview_i *a,
                      const vsip_mview_i *b,
                      const vsip_mview_i *r);
void vsip_mmul_f(
                      const vsip_mview_f *a,
                      const vsip_mview_f *b,
                      const vsip_mview_f *r);
void vsip_rcmmul_f(
                      const vsip_mview_f *a,
                      const vsip_cmview_f *b,
                      const vsip_cmview_f *r);
void vsip_cmmul_f(
                      const vsip_cmview_f *a,
                      const vsip_cmview_f *b,
                      const vsip_cmview_f *r);
```

Arguments

`*a`	View of input vector/matrix
`*b`	View of input vector/matrix
`*r`	View of output vector/matrix

Return Value

None.

Restrictions

In the case of a mixed data type, for instance a real vector multiplied by a complex vector, in-place implies the real input may be a real or imaginary view of the output. Input views which encompass both real and imaginary segments of the output, or which do not exactly overlay the real or imaginary view of the output, are not defined for in-place.

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.
3. The input and output views must be identical views of the same block (in-place), or must not overlap.

Notes/References

Examples

```
/* Example of vector multiply */
#include<stdio.h>
#include"vsip.h"

#define L 7                    /* length */

int main()
{
    int i;
    vsip_vview_d*  dataRe;
    vsip_vview_d*  dataIm;
    vsip_cvview_d* cvectorMul;
    vsip_cvview_d* cvectorLeft;
    vsip_cvview_d* cvectorRight;
    vsip_cscalar_d cscalar,cLeft,cRight;

    vsip_init((void *)0);
    dataRe = vsip_vcreate_d(L, VSIP_MEM_NONE);
    dataIm = vsip_vcreate_d(L, VSIP_MEM_NONE);
    cvectorMul = vsip_cvcreate_d(L, VSIP_MEM_NONE);
    cvectorLeft = vsip_cvcreate_d(L, VSIP_MEM_NONE);
    cvectorRight = vsip_cvcreate_d(L, VSIP_MEM_NONE);

    /* Make up some data to use for a vector multiply */
    vsip_vramp_d(1.0, 1.0 , dataRe);
    vsip_vramp_d(1.0, -2.0/(double)(L-1), dataIm);
    vsip_vcmplx_d(dataRe, dataIm, cvectorLeft);

    /* We will just use a simple constant vector for the right */
    cscalar = vsip_cmplx_d(2,1);
```

```
    vsip_cvfill_d(cscalar, cvectorRight);

    /* Now multiply the two vectors */
    vsip_cvmul_d(cvectorLeft,cvectorRight,cvectorMul);

    /* and print out the data and the Result */
    for(i=0; i<L; i++){
        cLeft = vsip_cvget_d(cvectorLeft, i);
        cRight = vsip_cvget_d(cvectorRight, i);
        cscalar = vsip_cvget_d(cvectorMul, i);
        printf("(%7.4f + %7.4fi) * (%7.4f + %7.4fi) = (%7.4f + %7.4fi)\n",
                  vsip_real_d(cLeft),vsip_imag_d(cLeft),
                  vsip_real_d(cRight),vsip_imag_d(cRight),
                  vsip_real_d(cscalar),vsip_imag_d(cscalar));
    }
    /* and don't forget to recover memory */
    vsip_blockdestroy_d(vsip_vdestroy_d(dataRe));
    vsip_blockdestroy_d(vsip_vdestroy_d(dataIm));
    vsip_cblockdestroy_d(vsip_cvdestroy_d(cvectorLeft));
    vsip_cblockdestroy_d(vsip_cvdestroy_d(cvectorRight));
    vsip_cblockdestroy_d(vsip_cvdestroy_d(cvectorMul));
    vsip_finalize((void *)0);
    return 0;
}
```

See Also

`vsip_csjmul_p`, `vsip_dssmul_p`, and `vsip_dvdmmul_p`

vsip_*d*ssmul_*p* Scalar Vector/Matrix Multiply

Computes the product, by element, of a scalar and a vector/matrix.

Functionality

$r_j \leftarrow \alpha \cdot b_j$ for $j = 0, 1, \ldots, N\text{-}1$

$r_{i,j} \leftarrow \alpha \cdot b_{i,j}$ for $i = 0, 1, \ldots, M\text{-}1$; for $j = 0, 1, \ldots, N\text{-}1$

Prototypes

```
void vsip_svmul_f(
                      vsip_scalar_f alpha,
                      const vsip_vview_f *b,
                      const vsip_vview_f *r);
void vsip_rscvmul_f(
                      vsip_scalar_f alpha,
                      const vsip_cvview_f *b,
                      const vsip_cvview_f *r);
void vsip_csvmul_f(
                      vsip_cscalar_f alpha,
                      const vsip_cvview_f *b,
                      const vsip_cvview_f *r);
void vsip_smmul_f(
                      vsip_scalar_f alpha,
                      const vsip_mview_f *b,
                      const vsip_mview_f *r);
void vsip_rscmmul_f(
                      vsip_scalar_f alpha,
                      const vsip_cmview_f *b,
                      const vsip_cmview_f *r);
void vsip_csmmul_f(
                      vsip_cscalar_f alpha,
                      const vsip_cmview_f *b,
                      const vsip_cmview_f *r);
```

Arguments

`alpha`	Input scalar
`*b`	View of input vector
`*r`	View of output vector

Return Value

None.

Restrictions

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.
3. The input and output views must be identical views of the same block (in-place), or must not overlap.

Notes/References

Examples

```
/* example of scalar-vector multiply */
#include<stdio.h>
#include "vsip.h"
#define L 5

int main()
{   int i;
   vsip_cvview_d* dataComplex;
   vsip_cscalar_d scalarComplex;
   vsip_cvview_d* dataComplexProduct;

   vsip_init((void *)0);
   dataComplex = vsip_cvcreate_d(L, VSIP_MEM_NONE);
   dataComplexProduct = vsip_cvcreate_d(L, VSIP_MEM_NONE);
   /* put some complex data in dataComplex */
   for(i = 0; i < L; i++)
     vsip_cvput_d(dataComplex,i,vsip_cmplx_d((double)(i * i),
                  (double)(i+1)));

   /* define a complex scalar */
   scalarComplex = vsip_cmplx_d(3,4);

   /*Multiply scalarComplex by dataComplex
     and print the input and output */
   vsip_csvmul_d(scalarComplex,dataComplex,dataComplexProduct);
   for(i=0; i<L; i++)
       printf("(%7.4f + %7.4fi) * (%7.4f + %7.4fi) = (%7.4f + %7.4fi)\n",
              vsip_real_d(scalarComplex),
              vsip_imag_d(scalarComplex),
              vsip_real_d(vsip_cvget_d(dataComplex,i)),
              vsip_imag_d(vsip_cvget_d(dataComplex,i)),
              vsip_real_d(vsip_cvget_d(dataComplexProduct,i)),
              vsip_imag_d(vsip_cvget_d(dataComplexProduct,i)));

   vsip_cblockdestroy_d(vsip_cvdestroy_d(dataComplex));
   vsip_cblockdestroy_d(vsip_cvdestroy_d(dataComplexProduct));
   vsip_finalize((void *)0);
   return 0;
}  /* output */
/* ( 3.0000 +  4.0000i) * ( 0.0000 +  1.0000i) = (-4.0000 +  3.0000i)
   ( 3.0000 +  4.0000i) * ( 1.0000 +  2.0000i) = (-5.0000 + 10.0000i)
   ( 3.0000 +  4.0000i) * ( 4.0000 +  3.0000i) = ( 0.0000 + 25.0000i)
   ( 3.0000 +  4.0000i) * ( 9.0000 +  4.0000i) = (11.0000 + 48.0000i)
   ( 3.0000 +  4.0000i) * (16.0000 +  5.0000i) = (28.0000 + 79.0000i) */
```

See Also

`vsip_csjmul_p`, `vsip_dsmul_p`, and `vsip_dvdmmul_p`

vsip_*dvd*mmul_*p* Vector-Matrix Multiply (Elementwise)

Computes the product, by element, of a vector and the rows or columns of a matrix.

Functionality

By Rows:

$r_{i,j} \leftarrow a_j \cdot b_{i,j}$ for $i = 0, 1, \ldots, M\text{-}1$; for $j = 0, 1, \ldots, N\text{-}1$

By Columns:

$r_{i,j} \leftarrow a_i \cdot b_{i,j}$ for $i = 0, 1, \ldots, M\text{-}1$; for $j = 0, 1, \ldots, N\text{-}1$

Prototypes

```
void vsip_vmmul_f(
                      const vsip_vview_f *a,
                      const vsip_mview_f *b,
                      vsip_major major,
                      const vsip_mview_f *r);
void vsip_cvmmul_f(
                      const vsip_cvview_f *a,
                      const vsip_cmview_f *b,
                      vsip_major major,
                      const vsip_cmview_f *r);
void vsip_rvcmmul_f(
                      const vsip_vview_f *a,
                      const vsip_cmview_f *b,
                      vsip_major major,
                      const vsip_cmview_f *r);
```

Arguments

`*a` Vector view - by rows: length *N*, by columns: length *M*

`*b` Matrix view - size *M* by *N*

`major` Apply by element to the rows or the columns

```
typedef enum {
   VSIP_ROW = 0;
   VSIP_COL = 1
   } vsip_major;
```

`*r` Result matrix view - size *M* by *N*

Return Value

None.

Restrictions

Errors

The arguments must conform to the following: Assuming an input matrix **B** of size *M* rows by *N* columns then we have the following conditions:

1. The input and output views must be conformant.
2. All view objects must be valid.
3. The *major* argument must be valid.
   `major ∈ {VSIP_ROW, VSIP_COL}`
4. The input and output matrix views must be identical views of the same block (in-place), or must not overlap. The input vector view and output matrix view must not overlap.

Notes/References

Examples

See Also

`vsip_cs`jmul`_p`, `vsip_ds`mul`_p`, and `vsip_d`s`s`mul`_p`

vsip_*ds*sub_*p* Vector/Matrix Subtract

Computes the difference, by element, of two vectors/matrices.

Functionality

$r_j \leftarrow a_j - b_j$ for $j = 0, 1, \ldots, N\text{-}1$

$r_{i,j} \leftarrow a_{i,j} - b_{i,j}$ for $i = 0, 1, \ldots, M\text{-}1$; for $j = 0, 1, \ldots, N\text{-}1$

Prototypes

```
void vsip_vsub_i(
                      const vsip_vview_i *a,
                      const vsip_vview_i *b,
                      const vsip_vview_i *r);
void vsip_vsub_f(
                      const vsip_vview_f *a,
                      const vsip_vview_f *b,
                      const vsip_vview_f *r);
void vsip_rcvsub_f(
                      const vsip_vview_f *a,
                      const vsip_cvview_f *b,
                      const vsip_cvview_f *r);
void vsip_crvsub_f(
                      const vsip_cvview_f *a,
                      const vsip_vview_f *b,
                      const vsip_cvview_f *r);
void vsip_cvsub_f(
                      const vsip_cvview_f *a,
                      const vsip_cvview_f *b,
                      const vsip_cvview_f *r);
void vsip_msub_i(
                      const vsip_mview_i *a,
                      const vsip_mview_i *b,
                      const vsip_mview_i *r);
void vsip_msub_f(
                      const vsip_mview_f *a,
                      const vsip_mview_f *b,
                      const vsip_mview_f *r);
void vsip_rcmsub_f(
                      const vsip_mview_f *a,
                      const vsip_cmview_f *b,
                      const vsip_cmview_f *r);
void vsip_crmsub_f(
                      const vsip_cmview_f *a,
                      const vsip_mview_f *b,
                      const vsip_cmview_f *r);
```

```
void vsip_cmsub_f(
                    const vsip_cmview_f *a,
                    const vsip_cmview_f *b,
                    const vsip_cmview_f *r);
```

Arguments

`*a`	View of input vector/matrix
`*b`	View of input vector/matrix
`*r`	View of output vector/matrix

Return Value

None.

Restrictions

In the case of a mixed data type in-place implies the real input may be a real or imaginary view of the output. Input views which encompass both real and imaginary segments of the output, or which do not exactly overlay the real or imaginary view of the output, are not defined for in-place.

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.
3. The input and output views must be identical views of the same block (in-place), or must not overlap.

Notes/References

Examples

```
/* Example of vector subtract */
#include<stdio.h>
#include "vsip.h"
#define L 7                  /* length */

int main()
{
    int i;
    vsip_vview_d* dataLeft;
    vsip_vview_d* dataRight;
    vsip_vview_d* dataSub;

    vsip_init((void *)0);
    dataLeft  = vsip_vcreate_d(L, VSIP_MEM_NONE);
    dataRight = vsip_vcreate_d(L, VSIP_MEM_NONE);
    dataSub   = vsip_vcreate_d(L, VSIP_MEM_NONE);

    /* Make up some data to find the magnitude of */
    vsip_vramp_d(1.0, 1.0 , dataLeft);
    vsip_vramp_d(1.0, -2.0/(double)(L-1), dataRight);

    /* Add the vectors */
    vsip_vsub_d(dataLeft, dataRight, dataSub);

    /*now print out the data and its sum*/
    for(i=0; i<L; i++)
```

```
        printf("%7.4f = (%7.4f) - (%7.4f) \n",vsip_vget_d(dataSub,i),
             vsip_vget_d(dataLeft,i),vsip_vget_d(dataRight,i));
    /*destroy the vector views and any associated blocks */
    vsip_blockdestroy_d(vsip_vdestroy_d(dataLeft));
    vsip_blockdestroy_d(vsip_vdestroy_d(dataRight));
    vsip_blockdestroy_d(vsip_vdestroy_d(dataSub));
    vsip_finalize((void *)0);
    return 0;
}
/* output */
/* 0.0000 = ( 1.0000) - ( 1.0000)
   1.3333 = ( 2.0000) - ( 0.6667)
   2.6667 = ( 3.0000) - ( 0.3333)
   4.0000 = ( 4.0000) - ( 0.0000)
   5.3333 = ( 5.0000) - (-0.3333)
   6.6667 = ( 6.0000) - (-0.6667)
   8.0000 = ( 7.0000) - (-1.0000) */
```

See Also

vsip_*dsss*sub_*p* Scalar Vector/Matrix Subtract

Computes the difference, by element, of a scalar and a vector/matrix.

Functionality

$r_j \leftarrow \alpha - b_j$ for $j = 0, 1, \ldots, N\text{-}1$

$r_{i,j} \leftarrow \alpha - b_{i,j}$ for $i = 0, 1, \ldots, M\text{-}1$; for $j = 0, 1, \ldots, N\text{-}1$

Prototypes

```
void vsip_svsub_i(
                    vsip_scalar_i alpha,
                    const vsip_vview_i *b,
                    const vsip_vview_i *r);
void vsip_svsub_f(
                    vsip_scalar_f alpha,
                    const vsip_vview_f *b,
                    const vsip_vview_f *r);
void vsip_rscvsub_f(
                    vsip_scalar_f alpha,
                    const vsip_cvview_f *b,
                    const vsip_cvview_f *r);
void vsip_csvsub_f(
                    vsip_cscalar_f alpha,
                    const vsip_cvview_f *b,
                    const vsip_cvview_f *r);
void vsip_smsub_i(
                    vsip_scalar_i alpha,
                    const vsip_mview_i *b,
                    const vsip_mview_i *r);
void vsip_smsub_f(
                    vsip_scalar_f alpha,
                    const vsip_mview_f *b,
                    const vsip_mview_f *r);
void vsip_rscmsub_f(
                    vsip_scalar_f alpha,
                    const vsip_cmview_f *b,
                    const vsip_cmview_f *r);
void vsip_csmsub_f(
                    vsip_cscalar_f alpha,
                    const vsip_cmview_f *b,
                    const vsip_cmview_f *r);
```

Arguments

`*a`	View of input vector/matrix
`*b`	View of input vector/matrix
`*r`	View of output vector/matrix

Return Value

None.

Restrictions

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.
3. The input and output views must be identical views of the same block (in-place), or must not overlap.

Notes/References

To subtract a scalar from a vector just multiply the scalar by minus one and use `vsip_svadd_f`.

Examples

```
/* Example of scalar vector subtract */
#include<stdio.h>
#include "vsip.h"
#define L 7                   /* length */

int main()
{
    int i;
    vsip_scalar_d dataScalar  = 5.5;
    vsip_vview_d* dataVector;
    vsip_vview_d* dataSub;

    vsip_init((void *)0);
    dataVector = vsip_vcreate_d(L, VSIP_MEM_NONE);
    dataSub = vsip_vcreate_d(L, VSIP_MEM_NONE);
    /* Make up some data */
    vsip_vramp_d(1.0, -2.0/(double)(L-1), dataVector);
    /* Subtract the vectors from the scalar*/
    vsip_svsub_d(dataScalar, dataVector, dataSub);
    /*now print out the data and the result*/
    for(i=0; i<L; i++)
        printf("%7.4f = (%7.4f) - (%7.4f) \n", vsip_vget_d(dataSub,i),
             dataScalar, vsip_vget_d(dataVector,i));
    /*destroy the vector views and any associated blocks */
    vsip_blockdestroy_d(vsip_vdestroy_d(dataVector));
    vsip_blockdestroy_d(vsip_vdestroy_d(dataSub));
    vsip_finalize((void *)0);
    return 0;
}
/* output */
/* 4.5000 = ( 5.5000) - ( 1.0000)
   4.8333 = ( 5.5000) - ( 0.6667)
   5.1667 = ( 5.5000) - ( 0.3333)
   5.5000 = ( 5.5000) - ( 0.0000)
   5.8333 = ( 5.5000) - (-0.3333)
   6.1667 = ( 5.5000) - (-0.6667)
   6.5000 = ( 5.5000) - (-1.0000) */
```

See Also

Ternary Operations

This section contains transformations which require three inputs; either three vectors, two vectors and a scalar, or two scalars and a vector.

`vsip_`***dvam_p***	Vector Add and Multiply
`vsip_`***dvma_p***	Vector Multiply and Add
`vsip_`***dvmsa_p***	Vector Multiply, Scalar Add
`vsip_`***dvmsb_p***	Vector Multiply and Subtract
`vsip_`***dvsam_p***	Vector Scalar Add, Vector Multiply
`vsip_`***dvsbm_p***	Vector Subtract and Multiply
`vsip_`***dvsma_p***	Vector Scalar Multiply, Vector Add
`vsip_`***dvsmsa_p***	Vector Scalar Multiply, Scalar Add

vsip_*d*vam_*p* Vector Add and Multiply

Computes the sum of two vectors and product of a third vector, by element.

Functionality

$r_j \leftarrow (a_j + b_j) \cdot c_j$ for $j = 0, 1, \ldots, N\text{-}1$

Prototypes

```
void vsip_vam_f(
                      const vsip_vview_f *a,
                      const vsip_vview_f *b,
                      const vsip_vview_f *c,
                      const vsip_vview_f *r);
void vsip_cvam_f(
                      const vsip_cvview_f *a,
                      const vsip_cvview_f *b,
                      const vsip_cvview_f *c,
                      const vsip_cvview_f *r);
```

Arguments

`*a`	View of input vector
`*b`	View of input vector
`*c`	View of input vector
`*r`	View of output vector

Return Value

None.

Restrictions

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.
3. The input and output views must be identical views of the same block (in-place), or must not overlap.

Notes/References

Examples

```
/* Example of vector add and multiply */
#include<stdio.h>
#include "vsip.h"

#define L 7                    /* length */

int main()
{
    int i;
    vsip_vview_d* dataA;
    vsip_vview_d* dataB;
    vsip_vview_d* dataC;
    vsip_vview_d* dataVam;
```

```
    vsip_init((void *)0);
    dataA   = vsip_vcreate_d(L, VSIP_MEM_NONE);
    dataB   = vsip_vcreate_d(L, VSIP_MEM_NONE);
    dataC   = vsip_vcreate_d(L, VSIP_MEM_NONE);
    dataVam = vsip_vcreate_d(L, VSIP_MEM_NONE);

    /* Make up some data */
    /* First compute a ramp from One to L */
    vsip_vramp_d(1.0, 1.0 , dataA);
    vsip_vramp_d(1.0, .25 , dataB);
    vsip_vramp_d(1.0, -1.5 , dataC);

    /* Add A and B and multiply times C */
    vsip_vam_d(dataA, dataB, dataC, dataVam);

    /*now print out the data and the result */
    for(i=0; i<L; i++)
        printf("(%7.4f + %7.4f) * %7.4f => %7.4f \n",
           vsip_vget_d(dataA, i),
           vsip_vget_d(dataB, i),
           vsip_vget_d(dataC, i),
           vsip_vget_d(dataVam, i));

    /*destroy the vector views and any associated blocks */
    vsip_blockdestroy_d(vsip_vdestroy_d(dataA));
    vsip_blockdestroy_d(vsip_vdestroy_d(dataB));
    vsip_blockdestroy_d(vsip_vdestroy_d(dataC));
    vsip_blockdestroy_d(vsip_vdestroy_d(dataVam));
    vsip_finalize((void *)0);
    return 0;
}
/* output */
/* ( 1.0000 +  1.0000) *  1.0000 =>  2.0000
   ( 2.0000 +  1.2500) * -0.5000 => -1.6250
   ( 3.0000 +  1.5000) * -2.0000 => -9.0000
   ( 4.0000 +  1.7500) * -3.5000 => -20.1250
   ( 5.0000 +  2.0000) * -5.0000 => -35.0000
   ( 6.0000 +  2.2500) * -6.5000 => -53.6250
   ( 7.0000 +  2.5000) * -8.0000 => -76.0000  */
```

See Also

`vsip_`*d*`vma_`*p*, `vsip_`*d*`vmsa_`*p*, `vsip_`*d*`vmsb_`*p*, `vsip_`*d*`vsam_`*p*, `vsip_`*d*`vsbm_`*p*, `vsip_`*d*`vsma_`*p*, and `vsip_`*d*`vsmsa_`*p*

vsip_*d*vma_*p* Vector Multiply and Add

Computes the product of two vectors and sum of a third vector, by element.

Functionality

$r_j \leftarrow \left(a_j \cdot b_j\right) + c_j$ for $j = 0, 1, \ldots, N\text{-}1$

Prototypes

```
void vsip_vma_f(
                      const vsip_vview_f *a,
                      const vsip_vview_f *b,
                      const vsip_vview_f *c,
                      const vsip_vview_f *r);
void vsip_cvma_f(
                      const vsip_cvview_f *a,
                      const vsip_cvview_f *b,
                      const vsip_cvview_f *c,
                      const vsip_cvview_f *r);
```

Arguments

`*a`	View of input vector
`*b`	View of input vector
`*c`	View of input vector
`*r`	View of output vector

Return Value

None.

Restrictions

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.
3. The input and output views must be identical views of the same block (in-place), or must not overlap.

Notes/References

Examples

```
/* Example of vector multiply and add */
#include<stdio.h>
#include "vsip.h"

#define L 7                    /* length */

int main()
{
    int i;
    vsip_vview_d* dataA;
    vsip_vview_d* dataB;
    vsip_vview_d* dataC;
    vsip_vview_d* dataVma;
```

```
    vsip_init((void *)0);
    dataA   = vsip_vcreate_d(L, VSIP_MEM_NONE);
    dataB   = vsip_vcreate_d(L, VSIP_MEM_NONE);
    dataC   = vsip_vcreate_d(L, VSIP_MEM_NONE);
    dataVma = vsip_vcreate_d(L, VSIP_MEM_NONE);
    /* Make up some data */
    /* First compute a ramp from One to L */
    vsip_vramp_d(1.0, 1.0 , dataA);
    vsip_vramp_d(1.0, .25 , dataB);
    vsip_vramp_d(1.0, -1.5 , dataC);

    /* Multiply A and B and Add C */
    vsip_vma_d(dataA, dataB, dataC, dataVma);

    /*now print out the data and the result */
    for(i=0; i<L; i++)
        printf("(%7.4f * %7.4f) + %7.4f => %7.4f \n",
           vsip_vget_d(dataA,i),
           vsip_vget_d(dataB,i),
           vsip_vget_d(dataC,i),
           vsip_vget_d(dataVma,i));

    /*destroy the vector views and any associated blocks */
    vsip_blockdestroy_d(vsip_vdestroy_d(dataA));
    vsip_blockdestroy_d(vsip_vdestroy_d(dataB));
    vsip_blockdestroy_d(vsip_vdestroy_d(dataC));
    vsip_blockdestroy_d(vsip_vdestroy_d(dataVma));
    vsip_finalize((void *)0);
    return 0;
}
/* output */
/* ( 1.0000 *  1.0000) +  1.0000 =>  2.0000
   ( 2.0000 *  1.2500) + -0.5000 =>  2.0000
   ( 3.0000 *  1.5000) + -2.0000 =>  2.5000
   ( 4.0000 *  1.7500) + -3.5000 =>  3.5000
   ( 5.0000 *  2.0000) + -5.0000 =>  5.0000
   ( 6.0000 *  2.2500) + -6.5000 =>  7.0000
   ( 7.0000 *  2.5000) + -8.0000 =>  9.5000  */
```

See Also

`vsip_`*d*`vam_`*p*, `vsip_`*d*`vmsa_`*p*, `vsip_`*d*`vmsb_`*p*, `vsip_`*d*`vsam_`*p*, `vsip_`*d*`vsbm_`*p*, `vsip_`*d*`vsma_`*p*, and `vsip_`*d*`vsmsa_`*p*

vsip_*d*vmsa_*p* Vector Multiply, Scalar Add

Computes the product of two vectors and sum of a scalar, by element.

Functionality

$r_j \leftarrow \left(a_j \cdot b_j\right) + \alpha$ for $j = 0, 1, \ldots, N\text{-}1$

Prototypes

```
void vsip_vmsa_f(
                      const vsip_vview_f *a,
                      const vsip_vview_f *b,
                      vsip_scalar_f alpha,
                      const vsip_vview_f *r);
void vsip_cvmsa_f(
                      const vsip_cvview_f *a,
                      const vsip_cvview_f *b,
                      vsip_cscalar_f alpha,
                      const vsip_cvview_f *r);
```

Arguments

`*a`	View of input vector
`*b`	View of input vector
`alpha`	Input scalar
`*r`	View of output vector

Return Value

None.

Restrictions

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.
3. The input and output views must be identical views of the same block (in-place), or must not overlap.

Notes/References

Examples

```
/* Example of vector-vector multiply and scalar Add */

#include<stdio.h>
#include "vsip.h"

#define L 7                    /* length */

int main()
{
    int i;
    vsip_vview_d* dataA;
    vsip_vview_d* dataB;
    vsip_scalar_d dataC;
    vsip_vview_d* dataVmsa;
```

```
    vsip_init((void *)0);
    dataA    = vsip_vcreate_d(L, VSIP_MEM_NONE);
    dataB    = vsip_vcreate_d(L, VSIP_MEM_NONE);
    dataVmsa = vsip_vcreate_d(L, VSIP_MEM_NONE);
    /* Make up some data*/
    /* First compute some ramps  */
    vsip_vramp_d(1.0, 1.0 , dataA);
    vsip_vramp_d(1.0, .25 , dataB);
    /* and make up a scalar */
    dataC = 4.5;

    /* Multiply A and B and add C */
    vsip_vmsa_d(dataA, dataB, dataC, dataVmsa);

    /*now print out the data and the result */
    for(i=0; i<L; i++)
        printf("(%7.4f * %7.4f) + %7.4f => %7.4f \n",
           vsip_vget_d(dataA,i),
           vsip_vget_d(dataB,i),
           dataC,
           vsip_vget_d(dataVmsa,i));

    /*destroy the vector views and any associated blocks */
    vsip_blockdestroy_d(vsip_vdestroy_d(dataA));
    vsip_blockdestroy_d(vsip_vdestroy_d(dataB));
    vsip_blockdestroy_d(vsip_vdestroy_d(dataVmsa));
    vsip_finalize((void *)0);
    return 0;
}
/* output */
/* ( 1.0000 *  1.0000) +  4.5000 =>  5.5000
   ( 2.0000 *  1.2500) +  4.5000 =>  7.0000
   ( 3.0000 *  1.5000) +  4.5000 =>  9.0000
   ( 4.0000 *  1.7500) +  4.5000 => 11.5000
   ( 5.0000 *  2.0000) +  4.5000 => 14.5000
   ( 6.0000 *  2.2500) +  4.5000 => 18.0000
   ( 7.0000 *  2.5000) +  4.5000 => 22.0000 */
```

See Also

`vsip_`*d*`vam_`*p*, `vsip_`*d*`vma_`*p*, `vsip_`*d*`vmsb_`*p*, `vsip_`*d*`vsam_`*p*, `vsip_`*d*`vsbm_`*p*, `vsip_`*d*`vsma_`*p*, and `vsip_`*d*`vsmsa_`*p*

vsip_*d*vmsb_*p* Vector Multiply and Subtract

Computes the product of two vectors and difference of a third vector, by element.

Functionality

$r_j \leftarrow \left(a_j \cdot b_j\right) - c_j$ for $j = 0, 1, \ldots, N\text{-}1$

Prototypes

```
void vsip_vmsb_f(
                      const vsip_vview_f *a,
                      const vsip_vview_f *b,
                      const vsip_vview_f *c,
                      const vsip_vview_f *r);
void vsip_cvmsb_f(
                      const vsip_cvview_f *a,
                      const vsip_cvview_f *b,
                      const vsip_cvview_f *c,
                      const vsip_cvview_f *r);
```

Arguments

`*a`	View of input vector
`*b`	View of input vector
`*c`	View of input vector
`*r`	View of output vector

Return Value

None.

Restrictions

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.
3. The input and output views must be identical views of the same block (in-place), or must not overlap.

Notes/References

Examples

```
/* Example of vector-vector multiply and subtract */
#include<stdio.h>
#include "vsip.h"

#define L 7                    /* length */

int main()
{
    int i;
    vsip_vview_d* dataA;
    vsip_vview_d* dataB;
    vsip_vview_d* dataC;
    vsip_vview_d* dataVmsb;
```

```
    vsip_init((void *)0);
    dataA    = vsip_vcreate_d(L, VSIP_MEM_NONE);
    dataB    = vsip_vcreate_d(L, VSIP_MEM_NONE);
    dataC    = vsip_vcreate_d(L, VSIP_MEM_NONE);
    dataVmsb = vsip_vcreate_d(L, VSIP_MEM_NONE);

    /* Make up some data */
    /* First compute some ramps  */
    vsip_vramp_d(1.0, 1.0 , dataA);
    vsip_vramp_d(1.0, .25 , dataB);
    vsip_vramp_d(1.0, -.25 , dataC);

    /* Multiply A and B and Subtract C */
    vsip_vmsb_d(dataA, dataB, dataC, dataVmsb);

    /*now print out the data and the result */
    for(i=0; i<L; i++)
        printf("(%7.4f * %7.4f) - %7.4f => %7.4f \n",
           vsip_vget_d(dataA,i),
           vsip_vget_d(dataB,i),
           vsip_vget_d(dataC,i),
           vsip_vget_d(dataVmsb,i));

    /*destroy the vector views and any associated blocks */
    vsip_blockdestroy_d(vsip_vdestroy_d(dataA));
    vsip_blockdestroy_d(vsip_vdestroy_d(dataB));
    vsip_blockdestroy_d(vsip_vdestroy_d(dataC));
    vsip_blockdestroy_d(vsip_vdestroy_d(dataVmsb));
    vsip_finalize((void *)0);
    return 0;
}
/* output */
/* ( 1.0000 *  1.0000) -  1.0000 =>  0.0000
   ( 2.0000 *  1.2500) -  0.7500 =>  1.7500
   ( 3.0000 *  1.5000) -  0.5000 =>  4.0000
   ( 4.0000 *  1.7500) -  0.2500 =>  6.7500
   ( 5.0000 *  2.0000) -  0.0000 => 10.0000
   ( 6.0000 *  2.2500) - -0.2500 => 13.7500
   ( 7.0000 *  2.5000) - -0.5000 => 18.0000  */
```

See Also

`vsip_`*d*`vam_`*p*, `vsip_`*d*`vma_`*p*, `vsip_`*d*`vmsa_`*p*, `vsip_`*d*`vsam_`*p*, `vsip_`*d*`vsbm_`*p*, `vsip_`*d*`vsma_`*p*, and `vsip_`*d*`vsmsa_`*p*

vsip_*d*vsam_*p* Vector Scalar Add, Vector Multiply

Computes the sum of a vector and a scalar, and product with a second vector, by element.

Functionality

$r_j \leftarrow \left(a_j + \beta\right) \cdot c_j$ for $j = 0, 1, \ldots, N\text{-}1$

Prototypes

```
void vsip_vsam_f(
                    const vsip_vview_f *a,
                    vsip_scalar_f beta,
                    const vsip_vview_f *c,
                    const vsip_vview_f *r);
void vsip_cvsam_f(
                    const vsip_cvview_f *a,
                    vsip_cscalar_f beta,
                    const vsip_cvview_f *c,
                    const vsip_cvview_f *r);
```

Arguments

`*a`	View of input vector
`beta`	Input scalar
`*c`	View of input vector
`*r`	View of output vector

Return Value

None.

Restrictions

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.
3. The input and output views must be identical views of the same block (in-place), or must not overlap.

Notes/References

Examples

```
/* Example of vector-scalar add and vector multiply */
#include<stdio.h>
#include "vsip.h"
#define L 7                    /* length  */
int main()
{
    int i;
    vsip_vview_d* dataA;
    vsip_scalar_d dataB;
    vsip_vview_d* dataC;
    vsip_vview_d* dataVsam;

    vsip_init((void *)0);
    dataA    = vsip_vcreate_d(L, VSIP_MEM_NONE);
```

```
    dataC    = vsip_vcreate_d(L, VSIP_MEM_NONE);
    dataVsam = vsip_vcreate_d(L, VSIP_MEM_NONE);
    /* Make up some data*/
    /* First compute some ramps  */
    vsip_vramp_d(1.0, 1.0 , dataA);
    vsip_vramp_d(1.0, .25 , dataC);
    /* and make up a scalar */
    dataB = 4.5;

    /* Add A and B and Multiply C */
    vsip_vsam_d(dataA, dataB, dataC, dataVsam);

    /*now print out the data and the result */
    for(i=0; i<L; i++)
        printf("(%7.4f + %7.4f) * %7.4f => %7.4f \n",
           vsip_vget_d(dataA,i),
           dataB,
           vsip_vget_d(dataC,i),
           vsip_vget_d(dataVsam,i));

    /*destroy the vector views and any associated blocks */
    vsip_blockdestroy_d(vsip_vdestroy_d(dataA));
    vsip_blockdestroy_d(vsip_vdestroy_d(dataC));
    vsip_blockdestroy_d(vsip_vdestroy_d(dataVsam));
    vsip_finalize((void *)0);
    return 0;
}
/* output */
/* ( 1.0000 +  4.5000) *  1.0000 =>  5.5000
   ( 2.0000 +  4.5000) *  1.2500 =>  8.1250
   ( 3.0000 +  4.5000) *  1.5000 => 11.2500
   ( 4.0000 +  4.5000) *  1.7500 => 14.8750
   ( 5.0000 +  4.5000) *  2.0000 => 19.0000
   ( 6.0000 +  4.5000) *  2.2500 => 23.6250
   ( 7.0000 +  4.5000) *  2.5000 => 28.7500 */
```

See Also

`vsip_`*d*`vam_`*p*, `vsip_`*d*`vma_`*p*, `vsip_`*d*`vmsa_`*p*, `vsip_`*d*`vmsb_`*p*, `vsip_`*d*`vsbm_`*p*, `vsip_`*d*`vsma_`*p*, and `vsip_`*d*`vsmsa_`*p*

vsip_*d*vsbm_*p* Vector Subtract and Multiply

Computes the difference of two vectors, and product with a third vector, by element.

Functionality

$r_j \leftarrow \left(a_j - b_j\right) \cdot c_j$ for $j = 0, 1, \ldots, N\text{-}1$

Prototypes

```
void vsip_vsbm_f(
                    const vsip_vview_f *a,
                    const vsip_vview_f *b,
                    const vsip_vview_f *c,
                    const vsip_vview_f *r);
void vsip_cvsbm_f(
                    const vsip_cvview_f *a,
                    const vsip_cvview_f *b,
                    const vsip_cvview_f *c,
                    const vsip_cvview_f *r);
```

Arguments

`*a`	View of input vector
`*b`	View of input vector
`*c`	View of input vector
`*r`	View of output vector

Return Value

None.

Restrictions

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.
3. The input and output views must be identical views of the same block (in-place), or must not overlap.

Notes/References

Examples

```
/* Example of vector subtract and multiply */
#include<stdio.h>
#include "vsip.h"

#define L 7                    /* length */

int main()
{    int i;
    vsip_vview_d* dataA;
    vsip_vview_d* dataB;
    vsip_vview_d* dataC;
    vsip_vview_d* dataVsbm;

    vsip_init((void *)0);
```

```
    dataA    = vsip_vcreate_d(L, VSIP_MEM_NONE);
    dataB    = vsip_vcreate_d(L, VSIP_MEM_NONE);
    dataC    = vsip_vcreate_d(L, VSIP_MEM_NONE);
    dataVsbm = vsip_vcreate_d(L, VSIP_MEM_NONE);

    /* Make up some data*/
    /* First compute a ramp from One to L */
    vsip_vramp_d(1.0, 1.0 , dataA);
    vsip_vramp_d(1.0, .25 , dataB);
    vsip_vramp_d(1.0, -1.5 , dataC);

    /* Subtract A and B and multiply times C */
    vsip_vsbm_d(dataA, dataB, dataC, dataVsbm);

    /*now print out the data and the result */
    for(i=0; i<L; i++)
        printf("(%7.4f - %7.4f) * %7.4f => %7.4f \n",
           vsip_vget_d(dataA,i),
           vsip_vget_d(dataB,i),
           vsip_vget_d(dataC,i),
           vsip_vget_d(dataVsbm,i));

    /*destroy the vector views and any associated blocks */
    vsip_blockdestroy_d(vsip_vdestroy_d(dataA));
    vsip_blockdestroy_d(vsip_vdestroy_d(dataB));
    vsip_blockdestroy_d(vsip_vdestroy_d(dataC));
    vsip_blockdestroy_d(vsip_vdestroy_d(dataVsbm));
    vsip_finalize((void *)0);
    return 0;
}
/* output */
/* ( 1.0000 -  1.0000) *  1.0000 =>  0.0000
   ( 2.0000 -  1.2500) * -0.5000 => -0.3750
   ( 3.0000 -  1.5000) * -2.0000 => -3.0000
   ( 4.0000 -  1.7500) * -3.5000 => -7.8750
   ( 5.0000 -  2.0000) * -5.0000 => -15.0000
   ( 6.0000 -  2.2500) * -6.5000 => -24.3750
   ( 7.0000 -  2.5000) * -8.0000 => -36.0000 */
```

See Also

vsip_*d*vam_*p*, vsip_*d*vma_*p*, vsip_*d*vmsa_*p*, vsip_*d*vmsb_*p*, vsip_*d*vsam_*p*, vsip_*d*vsma_*p*, and vsip_*d*vsmsa_*p*

vsip_*d*vsma_*p* Vector Scalar Multiply, Vector Add

Computes the product of a vector and a scalar, and sum with a second vector, by element.

Functionality

$r_j \leftarrow \left(a_j \cdot \beta\right) + c_j$ for $j = 0, 1, \ldots, N\text{-}1$

Prototypes

```
void vsip_vsma_f(
                    const vsip_vview_f *a,
                    vsip_scalar_f beta,
                    const vsip_vview_f *c,
                    const vsip_vview_f *r);
void vsip_cvsma_f(
                    const vsip_cvview_f *a,
                    vsip_cscalar_f beta,
                    const vsip_cvview_f *c,
                    const vsip_cvview_f *r);
```

Arguments

`*a`	View of input vector
`beta`	Input scalar
`*c`	View of input vector
`*r`	View of output vector

Return Value

None.

Restrictions

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.
3. The input and output views must be identical views of the same block (in-place), or must not overlap.

Notes/References

Examples

```
/* Example of vector-Scalar Multiply and vector Add */
#include<stdio.h>
#include "vsip.h"

#define L 7                    /* length */

int main()
{
    int i;
    vsip_vview_d* dataA;
    vsip_scalar_d dataB;
    vsip_vview_d* dataC;
    vsip_vview_d* dataVsma;
```

```
    vsip_init((void *)0);
    dataA    = vsip_vcreate_d(L, VSIP_MEM_NONE);
    dataC    = vsip_vcreate_d(L, VSIP_MEM_NONE);
    dataVsma = vsip_vcreate_d(L, VSIP_MEM_NONE);

    /* Make up some data*/
    vsip_vramp_d(1.0, 1.0 , dataA);
    vsip_vramp_d(1.0, .25 , dataC);
    dataB = 4.5;

    /* Multiply A and B and add C */
    vsip_vsma_d(dataA, dataB, dataC, dataVsma);

    /*now print out the data and the result */
    for(i=0; i<L; i++)
        printf("(%7.4f * %7.4f) + %7.4f => %7.4f \n",
           vsip_vget_d(dataA,i),
           dataB,
           vsip_vget_d(dataC,i),
           vsip_vget_d(dataVsma,i));

    /*destroy the vector views and any associated blocks */
    vsip_blockdestroy_d(vsip_vdestroy_d(dataA));
    vsip_blockdestroy_d(vsip_vdestroy_d(dataC));
    vsip_blockdestroy_d(vsip_vdestroy_d(dataVsma));
    vsip_finalize((void *)0);
    return 0;
}
/* output */
/* ( 1.0000 *  4.5000) +  1.0000 =>  5.5000
   ( 2.0000 *  4.5000) +  1.2500 => 10.2500
   ( 3.0000 *  4.5000) +  1.5000 => 15.0000
   ( 4.0000 *  4.5000) +  1.7500 => 19.7500
   ( 5.0000 *  4.5000) +  2.0000 => 24.5000
   ( 6.0000 *  4.5000) +  2.2500 => 29.2500
   ( 7.0000 *  4.5000) +  2.5000 => 34.0000 */
```

See Also

`vsip_`*d*`vam_`*p*, `vsip_`*d*`vma_`*p*, `vsip_`*d*`vmsa_`*p*, `vsip_`*d*`vmsb_`*p*, `vsip_`*d*`vsam_`*p*, `vsip_`*d*`vsbm_`*p*, and `vsip_`*d*`vsmsa_`*p*

vsip_*d*vsmsa_*p* Vector Scalar Multiply, Scalar Add

Computes the product of a vector and a scalar, and sum with a second scalar, by element.

Functionality

$r_j \leftarrow \left(a_j \cdot \beta\right) + \gamma$ for $j = 0, 1, \ldots, N\text{-}1$

Prototypes

```
void vsip_vsmsa_f(
                      const vsip_vview_f *a,
                      vsip_scalar_f beta,
                      vsip_scalar_f gamma,
                      const vsip_vview_f *r);
void vsip_cvsmsa_f(
                      const vsip_cvview_f *a,
                      vsip_cscalar_f beta,
                      vsip_cscalar_f gamma,
                      const vsip_cvview_f *r);
```

Arguments

`*a`	View of input vector
`beta`	Input scalar
`gamma`	Input scalar
`*r`	View of output vector

Return Value

None.

Restrictions

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.
3. The input and output views must be identical views of the same block (in-place), or must not overlap.

Notes/References

Examples

```
/* Example of vector-Scalar Multiply and Scalar Add */
#include<stdio.h>
#include "vsip.h"

#define L 7                    /* length */

int main()
{
    int i;
    vsip_vview_d* dataA;
    vsip_scalar_d dataB;
    vsip_scalar_d dataC;
    vsip_vview_d* dataVsmsa;
```

```
    vsip_init((void *)0);
    dataA     = vsip_vcreate_d(L, VSIP_MEM_NONE);
    dataVsmsa = vsip_vcreate_d(L, VSIP_MEM_NONE);

    /* Make up some data*/
    vsip_vramp_d(1.0, 1.0 , dataA);
    dataB = 4.5;
    dataC = -1.8;

    /* Multiply A and B and add C */
    vsip_vsmsa_d(dataA, dataB, dataC, dataVsmsa);

    /*now print out the data and the result */
    for(i=0; i<L; i++)
        printf("(%7.4f * %7.4f) + %7.4f => %7.4f \n",
           vsip_vget_d(dataA,i),
           dataB,
           dataC,
           vsip_vget_d(dataVsmsa,i));

    /*destroy the vector views and any associated blocks */
    vsip_blockdestroy_d(vsip_vdestroy_d(dataA));
    vsip_blockdestroy_d(vsip_vdestroy_d(dataVsmsa));
    vsip_finalize((void *)0);
    return 0;
}
/* output */
/* ( 1.0000 *  4.5000) + -1.8000 =>  2.7000
   ( 2.0000 *  4.5000) + -1.8000 =>  7.2000
   ( 3.0000 *  4.5000) + -1.8000 => 11.7000
   ( 4.0000 *  4.5000) + -1.8000 => 16.2000
   ( 5.0000 *  4.5000) + -1.8000 => 20.7000
   ( 6.0000 *  4.5000) + -1.8000 => 25.2000
   ( 7.0000 *  4.5000) + -1.8000 => 29.7000 */
```

See Also

`vsip_`*d*`vam_`*p*, `vsip_`*d*`vma_`*p*, `vsip_`*d*`vmsa_`*p*, `vsip_`*d*`vmsb_`*p*, `vsip_`*d*`vsam_`*p*, `vsip_`*d*`vsbm_`*p*, and `vsip_`*d*`vsma_`*p*

Logical Operations

The following functions are of two types. The first type compare two vectors/matrices elementwise using a logical test and returns a Boolean *true* or *false* depending on the result of the test. The second type tests a Boolean vector/matrix and produces a Boolean *true* or *false* depending on the state of the Boolean vector.

vsip_***S*alltrue**_bl	Vector/Matrix All True
vsip_***S*anytrue**_bl	Vector/Matrix Any True
vsip_***dS*leq_*p***	Vector/Matrix Logical Equal
vsip_***dsS*leq_*p***	Scalar Vector/Matrix Logical Equal
vsip_***S*lge_*p***	Vector/Matrix Logical Greater Than or Equal
vsip_***sS*lge_*p***	Scalar Vector/Matrix Logical Greater Than or Equal
vsip_***S*lgt_*p***	Vector/Matrix Logical Greater Than
vsip_***sS*lgt_*p***	Scalar Vector/Matrix Logical Greater Than
vsip_***S*lle_*p***	Vector/Matrix Logical Less Than or Equal
vsip_***sS*lle_*p***	Scalar Vector/Matrix Logical Less Than or Equal
vsip_***S*llt_*p***	Vector/Matrix Logical Less Than
vsip_***sS*llt_*p***	Scalar Vector/Matrix Logical Less Than
vsip_***dS*lne_*p***	Vector/Matrix Logical Not Equal
vsip_***dsS*lne_*p***	Scalar Vector/Matrix Logical Not Equal

vsip_salltrue_bl Vector/Matrix All True

Returns *true* if all the elements of a vector/matrix are true.

Functionality

$$\text{all} \leftarrow \bigwedge_{j}^{N-1} a_j$$

$$\text{all} \leftarrow \bigwedge_{j}^{N-1} \bigwedge_{i}^{M-1} a_{i,j}$$

Where $\wedge$ denotes logical "AND."

Prototypes

```
vsip_scalar_bl vsip_valltrue_bl(
                      const vsip_vview_bl *a);
vsip_scalar_bl vsip_malltrue_bl(
                      const vsip_mview_bl *a);
```

Arguments

`*a` View of input vector/matrix

Return Value

Returns *false* if any of the elements are false, otherwise it returns *true*.

Restrictions

Errors

None

Notes/References

Examples

```
/* example of alltrue */
#include<stdio.h>
#include "vsip.h"
#define L  5    /* length */

int main()
{
    int i = 0; int j = 0;
    vsip_vview_d*  dataA;
    vsip_vview_d*  dataB;
    vsip_vview_bl* dataBl;

    vsip_init((void *)0);
    dataA  = vsip_vcreate_d(L, VSIP_MEM_NONE);
    dataB  = vsip_vcreate_d(L, VSIP_MEM_NONE);
    dataBl = vsip_vcreate_bl(L, VSIP_MEM_NONE);
    vsip_vfill_d(0,dataB);

    /* Make up some data and determine if any of it is equal to zero*/
    printf("Any equal to zero?\n");
    for (i=-2; i<4; i++){
       vsip_vramp_d(i, 1 , dataA);
       vsip_vlne_d(dataA,dataB,dataBl);
       for(j=0; j<L; j++)
            printf("%3.0f",vsip_vget_d(dataA,j));
       if(vsip_valltrue_bl(dataBl))
```

```
            printf(" => None zero\n");
        else
            printf(" => Yes\n");
    }
    vsip_blockdestroy_d(vsip_vdestroy_d(dataA));
    vsip_blockdestroy_bl(vsip_vdestroy_bl(dataBl));
    vsip_finalize((void *)0);
    return 0;
}
/* output */
/* Any equal to zero?
   -2 -1  0  1  2 => Yes
   -1  0  1  2  3 => Yes
    0  1  2  3  4 => Yes
    1  2  3  4  5 => None zero
    2  3  4  5  6 => None zero
    3  4  5  6  7 => None zero */
```

See Also

vsip_***S***anytrue_bl.

vsip_sanytrue_bl Vector/Matrix Any True

Returns *true* if one or more elements of a vector/matrix are true.

Functionality

$$\text{any} \leftarrow \bigvee_{j}^{N-1} a_j$$

$$\text{any} \leftarrow \bigvee_{j}^{N-1} \bigvee_{i}^{M-1} a_{i,j}$$

Where $\vee$ denotes logical "OR."

Prototypes

```
vsip_scalar_bl vsip_vanytrue_bl(
                      const vsip_vview_bl *a);
vsip_scalar_bl vsip_manytrue_bl(
                      const vsip_mview_bl *a);
```

Arguments

`*a` View of input vector/matrix

Return Value

Returns a *false* if all the elements are false, otherwise a *true* is returned.

Restrictions

Errors

None

Notes/References

The logical complement of *any true* is *none true*.

Examples

```
/* example of anytrue */
#include<stdio.h>
#include "vsip.h"

#define L  5       /* length */

int main()
{
    int i = 0; int j = 0;
    vsip_vview_d*  dataA;
    vsip_vview_d*  dataB;
    vsip_vview_bl* dataBl;

    vsip_init((void *)0);
    dataA  = vsip_vcreate_d(L, VSIP_MEM_NONE);
    dataB  = vsip_vcreate_d(L, VSIP_MEM_NONE);
    dataBl = vsip_vcreate_bl(L, VSIP_MEM_NONE);
    vsip_vfill_d(0,dataB);

    /* Make up some data and determine if it is greater than zero */
    printf("Any greater than zero\n");
    for (i=-6; i<0; i++){
       vsip_vramp_d(i, 1 , dataA);
       vsip_vlgt_d(dataA,dataB,dataBl);
```

```
        for(j=0; j<L; j++)
             printf("%3.0f",vsip_vget_d(dataA,j));
        if(vsip_vanytrue_bl(dataBl))
            printf(" => Some true\n");
        else
            printf(" => None true\n");
    }
    vsip_blockdestroy_d(vsip_vdestroy_d(dataA));
    vsip_blockdestroy_bl(vsip_vdestroy_bl(dataBl));
    vsip_finalize((void *)0);
    return 0;
}
/* output */
/* Any greater than zero
   -6 -5 -4 -3 -2 => None true
   -5 -4 -3 -2 -1 => None true
   -4 -3 -2 -1  0 => None true
   -3 -2 -1  0  1 => Some true
   -2 -1  0  1  2 => Some true
   -1  0  1  2  3 => Some true */
```

See Also

vsip_***S***alltrue_bl.

vsip_*ds*leq_*p* Vector/Matrix Logical Equal

Computes the boolean comparison of "equal," by element, of two vectors/matrices.

Functionality

for $j = 0, 1, \ldots, N\text{-}1$
  if $a_j = b_j$ then
    $r_j \leftarrow true$;
  else
    $r_j \leftarrow false$;

for $i = 0, 1, \ldots, M\text{-}1$
  for $j = 0, 1, \ldots, N\text{-}1$
    if $a_{i,j} = b_{i,j}$ then
      $r_{i,j} \leftarrow true$;
    else
      $r_{i,j} \leftarrow false$;

Prototypes

```
void vsip_vleq_i(
                      const vsip_vview_i *a,
                      const vsip_vview_i *b,
                      const vsip_vview_bl *r);
void vsip_cvleq_i(
                      const vsip_cvview_i *a,
                      const vsip_cvview_i *b,
                      const vsip_vview_bl *r);
void vsip_vleq_f(
                      const vsip_vview_f *a,
                      const vsip_vview_f *b,
                      const vsip_vview_bl *r);
void vsip_cvleq_f(
                      const vsip_cvview_f *a,
                      const vsip_cvview_f *b,
                      const vsip_vview_bl *r);
void vsip_mleq_i(
                      const vsip_mview_i *a,
                      const vsip_mview_i *b,
                      const vsip_mview_bl *r);
void vsip_cmleq_i(
                      const vsip_cmview_i *a,
                      const vsip_cmview_i *b,
                      const vsip_mview_bl *r);
void vsip_mleq_f(
                      const vsip_mview_f *a,
                      const vsip_mview_f *b,
                      const vsip_mview_bl *r);
void vsip_cmleq_f(
                      const vsip_cmview_f *a,
                      const vsip_cmview_f *b,
                      const vsip_mview_bl *r);
```

Arguments

`*a`	View of input vector/matrix
`*b`	View of input vector/matrix
`*r`	View of Boolean output vector/matrix

Return Value

None.

Restrictions

Since the input and output vectors are of a different precision there is no in-place functionality for this function.

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.

Notes/References

Examples

```
/* example of vleq */
#include<stdio.h>
#include "vsip.h"
#define L  9   /* length */
int main()
{
    vsip_vview_d*  dataA;
    vsip_vview_d*  dataB;
    vsip_vview_bl* dataBl;
    vsip_vview_vi* dataVi;
    vsip_scalar_vi numTrue = 0;
    int i = 0;
    vsip_init((void *)0);
    dataA  = vsip_vcreate_d(L, VSIP_MEM_NONE);
    dataB  = vsip_vcreate_d(L, VSIP_MEM_NONE);
    dataBl = vsip_vcreate_bl(L, VSIP_MEM_NONE);
    dataVi = vsip_vcreate_vi(L, VSIP_MEM_NONE);
    /* Make up some data */
    vsip_vramp_d(-2.0, 1 , dataA);
    vsip_vramp_d(2.0, -1 , dataB);
    printf("index    A       B\n");
    for(i = 0; i<L; i++)
       printf("%3i %7.1f %7.1f \n", i,
              vsip_vget_d(dataA,i),
              vsip_vget_d(dataB,i));
    /* now see if our ramps are equal someplace */
    vsip_vleq_d(dataA,dataB,dataBl);
    /* find the spots where dataA equals dataB */
    if(vsip_vanytrue_bl(dataBl)){
        numTrue = vsip_vindexbool(dataBl,dataVi);
        /* print out the results */
        for(i = 0; i < numTrue; i++)
           printf("A = B at index %3i\n",(int)vsip_vget_vi(dataVi,i));
    }else{
        printf("No true cases\n");
```

```
      }    /* recover allocated memory */
      vsip_blockdestroy_d(vsip_vdestroy_d(dataA));
      vsip_blockdestroy_d(vsip_vdestroy_d(dataB));
      vsip_blockdestroy_bl(vsip_vdestroy_bl(dataBl));
      vsip_blockdestroy_vi(vsip_vdestroy_vi(dataVi));
      vsip_finalize((void *)0);
      return 0;
  }
  /* output */
  /* index    A       B
       0    -2.0     2.0
       1    -1.0     1.0
       2     0.0     0.0
       3     1.0    -1.0
       4     2.0    -2.0
       5     3.0    -3.0
       6     4.0    -4.0
       7     5.0    -5.0
       8     6.0    -6.0
     A = B at index   2  */
```

See Also

vsip_*d*ssleq_*p* Scalar Vector/Matrix Logical Equal

Computes the boolean comparison of "equal," by element, of a scalar constant with a vector/matrix.

Functionality

for $j = 0, 1, \ldots, N\text{-}1$
  if $alpha = b_j$ then
    $r_j \leftarrow true$;
  else
    $r_j \leftarrow false$;

for $i = 0, 1, \ldots, M\text{-}1$
  for $j = 0, 1, \ldots, N\text{-}1$
    if $alpha = b_{i,j}$ then
      $r_{i,j} \leftarrow true$;
    else
      $r_{i,j} \leftarrow false$;

Prototypes

```
void vsip_svleq_i(
                      vsip_scalar_i alpha,
                      const vsip_vview_i *b,
                      const vsip_vview_bl *r);
void vsip_csvleq_i(
                      vsip_cscalar_i alpha,
                      const vsip_cvview_i *b,
                      const vsip_vview_bl *r);
void vsip_svleq_f(
                      vsip_scalar_f alpha,
                      const vsip_vview_f *b,
                      const vsip_vview_bl *r);
void vsip_csvleq_f(
                      vsip_cscalar_f alpha,
                      const vsip_cvview_f *b,
                      const vsip_vview_bl *r);
void vsip_smleq_i(
                      vsip_scalar_i alpha,
                      const vsip_mview_i *b,
                      const vsip_mview_bl *r);
void vsip_csmleq_i(
                      vsip_cscalar_i alpha,
                      const vsip_cmview_i *b,
                      const vsip_mview_bl *r);
void vsip_smleq_f(
                      vsip_scalar_f alpha,
                      const vsip_mview_f *b,
                      const vsip_mview_bl *r);
void vsip_csmleq_f(
                      vsip_cscalar_f alpha,
                      const vsip_cmview_f *b,
                      const vsip_mview_bl *r);
```

Arguments

`alpha`	A scalar constant
`*b`	View of input vector/matrix
`*r`	View of Boolean output vector/matrix

Return Value

None.

Restrictions

Since the input and output vectors are of a different precision there is no in-place functionality for this function.

Errors

The arguments must conform to the following:

1. Input and output views must be the same size.
2. All view objects must be valid.

Notes/References

Examples

See Also

vsip_slge_*p* Vector/Matrix Logical Greater Than or Equal

Computes the boolean comparison of "greater than or equal," by element, of two vectors/matrices.

Functionality

for $j = 0, 1, \ldots, N\text{-}1$
  if $a_j \geq b_j$ then
    $r_j \leftarrow true$;
  else
    $r_j \leftarrow false$;

for $i = 0, 1, \ldots, M\text{-}1$
  for $j = 0, 1, \ldots, N\text{-}1$
    if $a_{i,j} \geq b_{i,j}$ then
      $r_{i,j} \leftarrow true$;
    else
      $r_{i,j} \leftarrow false$;

Prototypes

```
void vsip_vlge_i(
                      const vsip_vview_i *a,
                      const vsip_vview_i *b,
                      const vsip_vview_bl *r);

void vsip_vlge_f(
                      const vsip_vview_f *a,
                      const vsip_vview_f *b,
                      const vsip_vview_bl *r);

void vsip_mlge_i(
                      const vsip_mview_i *a,
                      const vsip_mview_i *b,
                      const vsip_mview_bl *r);

void vsip_mlge_f(
                      const vsip_mview_f *a,
                      const vsip_mview_f *b,
                      const vsip_mview_bl *r);
```

Arguments

`*a`	View of input vector/matrix
`*b`	View of input vector/matrix
`*r`	View of Boolean output vector/matrix

Return Value

None.

Restrictions

Since the input and output vectors are of a different precision there is no in-place functionality for this function.

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.

Notes/References

Examples

For example see example with `vsip_`***ds***`gather_`***p***.

See Also

vsip_sslge_*p* Scalar Vector/Matrix Logical Greater Than or Equal

Computes the boolean comparison of "greater than or equal," by element, of a scalar constant with a vector/matrix.

Functionality

for $j = 0, 1, \ldots, N\text{-}1$
  if $alpha \geq b_j$ then
    $r_j \leftarrow true$;
  else
    $r_j \leftarrow false$;

for $i = 0, 1, \ldots, M\text{-}1$
  for $j = 0, 1, \ldots, N\text{-}1$
    if $alpha \geq b_{i,j}$ then
      $r_{i,j} \leftarrow true$;
    else
      $r_{i,j} \leftarrow false$;

Prototypes

```
void vsip_svlge_i(
                    vsip_scalar_i alpha,
                    const vsip_vview_i *b,
                    const vsip_vview_bl *r);

void vsip_svlge_f(
                    vsip_scalar_f alpha,
                    const vsip_vview_f *b,
                    const vsip_vview_bl *r);

void vsip_smlge_i(
                    vsip_scalar_i alpha,,
                    const vsip_mview_i *b,
                    const vsip_mview_bl *r);

void vsip_smlge_f(
                    vsip_scalar_f alpha,
                    const vsip_mview_f *b,
                    const vsip_mview_bl *r);
```

Arguments

`alpha`	Scalar Constant
`*b`	View of input vector/matrix
`*r`	View of Boolean output vector/matrix

Return Value

None.

Restrictions

Since the input and output vectors are of a different precision there is no in-place functionality for this function.

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.

Notes/References

Examples

See Also

vsip_slgt_*p* Vector/Matrix Logical Greater Than

Computes the boolean comparison of "greater than," by element, of two vectors/matrices.

Functionality

for $j = 0, 1, \ldots, N\text{-}1$
  if $a_j > b_j$ then
    $r_j \leftarrow true$;
  else
    $r_j \leftarrow false$;

for $i = 0, 1, \ldots, M\text{-}1$
  for $j = 0, 1, \ldots, N\text{-}1$
    if $a_{i,j} > b_{i,j}$ then
      $r_{i,j} \leftarrow true$;
    else
      $r_{i,j} \leftarrow false$;

Prototypes

```
void vsip_vlgt_i(
                      const vsip_vview_i *a,
                      const vsip_vview_i *b,
                      const vsip_vview_bl *r);
void vsip_vlgt_f(
                      const vsip_vview_f *a,
                      const vsip_vview_f *b,
                      const vsip_vview_bl *r);
void vsip_mlgt_i(
                      const vsip_mview_i *a,
                      const vsip_mview_i *b,
                      const vsip_mview_bl *r);
void vsip_mlgt_f(
                      const vsip_mview_f *a,
                      const vsip_mview_f *b,
                      const vsip_mview_bl *r);
```

Arguments

`*a`	View of input vector/matrix
`*b`	View of input vector/matrix
`*r`	View of Boolean output vector/matrix

Return Value

None.

Restrictions

Since the input and output vectors are of a different precision there is no in-place functionality for this function.

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.

Notes/References

Examples

For example see example included with `vsip_`***ds***`scatter_`***p***.

See Also

vsip_sslgt_*p* Scalar Vector/Matrix Logical Greater Than

Computes the boolean comparison of "greater than," by element, of a scalar and a vector/matrix.

Functionality

for $j = 0, 1, \ldots, N\text{-}1$
  if $alpha > b_j$ then
    $r_j \leftarrow true$;
  else
    $r_j \leftarrow false$;

for $i = 0, 1, \ldots, M\text{-}1$
  for $j = 0, 1, \ldots, N\text{-}1$
    if $alpha > b_{i,j}$ then
      $r_{i,j} \leftarrow true$;
    else
      $r_{i,j} \leftarrow false$;

Prototypes

```
void vsip_svlgt_i(
                      vsip_scalar_i alpha,
                      const vsip_vview_i *b,
                      const vsip_vview_bl *r);
void vsip_svlgt_f(
                      vsip_scalar_f alpha,
                      const vsip_vview_f *b,
                      const vsip_vview_bl *r);
void vsip_smlgt_i(
                      vsip_scalar_i alpha,
                      const vsip_mview_i *b,
                      const vsip_mview_bl *r);
void vsip_smlgt_f(
                      vsip_scalar_f alpha,
                      const vsip_mview_f *b,
                      const vsip_mview_bl *r);
```

Arguments

`alpha`	Scalar input.
`*b`	View of input vector/matrix
`*r`	View of Boolean output vector/matrix

Return Value

None.

Restrictions

Since the input and output vectors are of a different precision there is no in-place functionality for this function.

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.

Notes/References

Examples

For example see example included with `vsip_`***ds***`scatter_`***p***.

See Also

vsip_slle_*p* Vector/Matrix Logical Less Than or Equal

Computes the boolean comparison of "less than or equal," by element, of two vectors/matrices.

Functionality

for $j = 0, 1, \ldots, N\text{-}1$
  if $a_j \le b_j$ then
    $r_j \leftarrow true$;
  else
    $r_j \leftarrow false$;

for $i = 0, 1, \ldots, M\text{-}1$
  for $j = 0, 1, \ldots, N\text{-}1$
    if $a_{i,j} \le b_{i,j}$ then
      $r_{i,j} \leftarrow true$;
    else
      $r_{i,j} \leftarrow false$;

Prototypes

```
void vsip_vlle_i(
                      const vsip_vview_i *a,
                      const vsip_vview_i *b,
                      const vsip_vview_bl *r);
void vsip_vlle_f(
                      const vsip_vview_f *a,
                      const vsip_vview_f *b,
                      const vsip_vview_bl *r);
void vsip_mlle_i(
                      const vsip_mview_i *a,
                      const vsip_mview_i *b,
                      const vsip_mview_bl *r);
void vsip_mlle_f(
                      const vsip_mview_f *a,
                      const vsip_mview_f *b,
                      const vsip_mview_bl *r);
```

Arguments

`*a`	View of input vector/matrix
`*b`	View of input vector/matrix
`*r`	View of Boolean output vector/matrix

Return Value

None.

Restrictions

Since the input and output vectors are of a different precision there is no in-place functionality for this function.

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.

Notes/References

Examples

```
/* example of Less than or Equal */
```

```
#include <stdio.h>
#include "vsip.h"

#define L  9
#define PI  3.14159265358979323846

int main()
{
    int i;
    vsip_vview_d  *dataCos, *dataSin;
    vsip_vview_bl *dataLle;

    vsip_init((void *)0);
    dataCos = vsip_vcreate_d(L, VSIP_MEM_NONE),
    dataSin = vsip_vcreate_d(L, VSIP_MEM_NONE);
    dataLle = vsip_vcreate_bl(L, VSIP_MEM_NONE);

    /* Make up some data and do the check*/
    vsip_vramp_d(0.0, 2 * PI/(L-1), dataCos);
    vsip_vsin_d(dataCos, dataSin);
    vsip_vcos_d(dataCos, dataCos);
    vsip_vlle_d(dataSin, dataCos, dataLle);

    /* Print the results */
    printf("  Sin                  Cos"
         "                  Sin < or equal Cos?\n");
    for(i=0; i<L; i++){
        printf("%20.17f %20.17f        %5s\n",
             vsip_vget_d(dataSin,i), vsip_vget_d(dataCos,i),
             vsip_vget_bl(dataLle,i) ? "true" : "false");
    }
    vsip_valldestroy_d (dataCos);
    vsip_valldestroy_d (dataSin);
    vsip_valldestroy_bl(dataLle);
    vsip_finalize((void *)0);
    return 0;
}
/* output */
/*  Sin                  Cos                  Sin < or equal Cos?
   0.00000000000000000  1.00000000000000000         true
   0.70710678118654746  0.70710678118654757         true
   1.00000000000000000  0.00000000000000006        false
   0.70710678118654757 -0.70710678118654746        false
   0.00000000000000012 -1.00000000000000000        false
  -0.70710678118654746 -0.70710678118654768        false
  -1.00000000000000000 -0.00000000000000018         true
  -0.70710678118654768  0.70710678118654735         true
  -0.00000000000000024  1.00000000000000000         true */
```

See Also

vsip_sslle_*p* Scalar Vector/Matrix Logical Less Than or Equal

Computes the boolean comparison of "less than or equal," by element, of a scalar and a vector/matrix.

Functionality

for $j = 0, 1, \ldots, N\text{-}1$
  if $alpha \le b_j$ then
    $r_j \leftarrow true$;
  else
    $r_j \leftarrow false$;

for $i = 0, 1, \ldots, M\text{-}1$
  for $j = 0, 1, \ldots, N\text{-}1$
    if $alpha \le b_{i,j}$ then
      $r_{i,j} \leftarrow true$;
    else
      $r_{i,j} \leftarrow false$;

Prototypes

```
void vsip_svlle_i(
                    vsip_scalar_i alpha,
                    const vsip_vview_i *b,
                    const vsip_vview_bl *r);
void vsip_svlle_f(
                    vsip_scalar_f alpha,
                    const vsip_vview_f *b,
                    const vsip_vview_bl *r);
void vsip_smlle_i(
                    vsip_scalar_i alpha,
                    const vsip_mview_i *b,
                    const vsip_mview_bl *r);
void vsip_smlle_f(
                    vsip_scalar_f alpha,
                    const vsip_mview_f *b,
                    const vsip_mview_bl *r);
```

Arguments

`alpha`	Input scalar
`*b`	View of input vector/matrix
`*r`	View of Boolean output vector/matrix

Return Value

None.

Restrictions

Since the input and output vectors are of a different precision there is no in-place functionality for this function.

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.

Notes/References
Examples
See Also

vsip_sllt_*p* Vector/Matrix Logical Less Than

Computes the boolean comparison of "less than," by element, of two vectors/matrices.

Functionality

for $j = 0, 1, \ldots, N\text{-}1$
  if $a_j < b_j$ then
    $r_j \leftarrow true$;
  else
    $r_j \leftarrow false$;

for $i = 0, 1, \ldots, M\text{-}1$
  for $j = 0, 1, \ldots, N\text{-}1$
    if $a_{i,j} < b_{i,j}$ then
      $r_{i,j} \leftarrow true$;
    else
      $r_{i,j} \leftarrow false$;

Prototypes

```
void vsip_vllt_i(
                      const vsip_vview_i *a,
                      const vsip_vview_i *b,
                      const vsip_vview_bl *r);
void vsip_vllt_f(
                      const vsip_vview_f *a,
                      const vsip_vview_f *b,
                      const vsip_vview_bl *r);
void vsip_mllt_i(
                      const vsip_mview_i *a,
                      const vsip_mview_i *b,
                      const vsip_mview_bl *r);
void vsip_mllt_f(
                      const vsip_mview_f *a,
                      const vsip_mview_f *b,
                      const vsip_mview_bl *r);
```

Arguments

`*a`	View of input vector/matrix
`*b`	View of input vector/matrix
`*r`	View of Boolean output vector/matrix

Return Value

None.

Restrictions

Since the input and output vectors are of a different precision there is no in-place functionality for this function.

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.

Notes/References

Examples

```
/* example of less than */
#include <stdio.h>
#include "vsip.h"

#define L  9
#define PI  3.14159265358979323846

int main()
{
    int i;
    vsip_vview_d  *dataCos, *dataSin;
    vsip_vview_bl *dataLlt;

    vsip_init((void *)0);
    dataCos = vsip_vcreate_d(L, VSIP_MEM_NONE);
    dataSin = vsip_vcreate_d(L, VSIP_MEM_NONE);
    dataLlt = vsip_vcreate_bl(L, VSIP_MEM_NONE);

    /* Make up some data and do the check*/
    vsip_vramp_d(0.0, 2 * PI/(L-1), dataCos);
    vsip_vsin_d(dataCos, dataSin);
    vsip_vcos_d(dataCos, dataCos);
    vsip_vllt_d(dataSin, dataCos, dataLlt);

    /* Print the results */
    printf("  Sin                  Cos                    Sin < Cos?\n");
    for(i=0; i<L; i++){
        printf("%20.17f %20.17f        %5s\n",
             vsip_vget_d(dataSin,i), vsip_vget_d(dataCos,i),
             vsip_vget_bl(dataLlt,i) ? "true" : "false");
    }
    vsip_valldestroy_d (dataCos);
    vsip_valldestroy_d (dataSin);
    vsip_valldestroy_bl(dataLlt);
    vsip_finalize((void *)0);
    return 0;
}
/* output */
/*  Sin                  Cos                    Sin < Cos?
   0.00000000000000000  1.00000000000000000         true
   0.70710678118654746  0.70710678118654757         true
   1.00000000000000000  0.00000000000000006        false
   0.70710678118654757 -0.70710678118654746        false
   0.00000000000000012 -1.00000000000000000        false
  -0.70710678118654746 -0.70710678118654768        false
  -1.00000000000000000 -0.00000000000000018         true
  -0.70710678118654768  0.70710678118654735         true
  -0.00000000000000024  1.00000000000000000         true */
```

See Also

vsip_ssllt_*p* Scalar Vector/Matrix Logical Less Than

Computes the boolean comparison of "less than," by element, of a scalar and a vector/matrix.

Functionality

for $j = 0, 1, \ldots, N\text{-}1$
  if $alpha < b_j$ then
    $r_j \leftarrow true$;
  else
    $r_j \leftarrow false$;

for $i = 0, 1, \ldots, M\text{-}1$
  for $j = 0, 1, \ldots, N\text{-}1$
    if $alpha < b_{i,j}$ then
      $r_{i,j} \leftarrow true$;
    else
      $r_{i,j} \leftarrow false$;

Prototypes

```
void vsip_svllt_i(
                      vsip_scalar_i alpha,
                      const vsip_vview_i *b,
                      const vsip_vview_bl *r);
void vsip_svllt_f(
                      vsip_scalar_f alpha,
                      const vsip_vview_f *b,
                      const vsip_vview_bl *r);
void vsip_smllt_i(
                      vsip_scalar_i alpha,
                      const vsip_mview_i *b,
                      const vsip_mview_bl *r);
void vsip_smllt_f(
                      vsip_scalar_f alpha,
                      const vsip_mview_f *b,
                      const vsip_mview_bl *r);
```

Arguments

`alpha`	Input Scalar
`*b`	View of input vector/matrix
`*r`	View of Boolean output vector/matrix

Return Value

None.

Restrictions

Since the input and output vectors are of a different precision there is no in-place functionality for this function.

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.

Notes/References

Examples
See Also

vsip_*ds*lne_*p* Vector/Matrix Logical Not Equal

Computes the boolean comparison of "not equal," by element, of two vectors/matrices.

Functionality

for $j = 0, 1, \ldots, N\text{-}1$
  if $a_j \neq b_j$ then
    $r_j \leftarrow true$;
  else
    $r_j \leftarrow false$;

for $i = 0, 1, \ldots, M\text{-}1$
  for $j = 0, 1, \ldots, N\text{-}1$
    if $a_{i,j} \neq b_{i,j}$ then
      $r_{i,j} \leftarrow true$;
    else
      $r_{i,j} \leftarrow false$;

Prototypes

```
void vsip_vlne_i(
                    const vsip_vview_i *a,
                    const vsip_vview_i *b,
                    const vsip_vview_bl *r);
void vsip_cvlne_i(
                    const vsip_cvview_i *a,
                    const vsip_cvview_i *b,
                    const vsip_vview_bl *r);
void vsip_vlne_f(
                    const vsip_vview_f *a,
                    const vsip_vview_f *b,
                    const vsip_vview_bl *r);
void vsip_cvlne_f(
                    const vsip_cvview_f *a,
                    const vsip_cvview_f *b,
                    const vsip_vview_bl *r);
void vsip_mlne_i(
                    const vsip_mview_i *a,
                    const vsip_mview_i *b,
                    const vsip_mview_bl *r);
void vsip_cmlne_i(
                    const vsip_cmview_i *a,
                    const vsip_cmview_i *b,
                    const vsip_mview_bl *r);
void vsip_mlne_f(
                    const vsip_mview_f *a,
                    const vsip_mview_f *b,
                    const vsip_mview_bl *r);
void vsip_cmlne_f(
                    const vsip_cmview_f *a,
                    const vsip_cmview_f *b,
                    const vsip_mview_bl *r);
```

Arguments

`*a`	View of input vector/matrix
`*b`	View of input vector/matrix
`*r`	View of Boolean output vector/matrix

Return Value

None.

Restrictions

Since the input and output vectors are of a different precision there is no in-place functionality for this function.

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.

Notes/References

Examples

For example of vector logical not equal see example included with `vsip_`***s***`alltrue_bl`.

See Also

vsip_*ds*lne_*p* Scalar Vector/Matrix Logical Not Equal

Computes the boolean comparison of "not equal," by element, of a scalar and a vector/matrix.

Functionality

for $j = 0, 1, \ldots, N\text{-}1$
  if $alpha \neq b_j$ then
    $r_j \leftarrow true$;
  else
    $r_j \leftarrow false$;

for $i = 0, 1, \ldots, M\text{-}1$
  for $j = 0, 1, \ldots, N\text{-}1$
    if $alpha \neq b_{i,j}$ then
      $r_{i,j} \leftarrow true$;
    else
      $r_{i,j} \leftarrow false$;

Prototypes

```
void vsip_svlne_i(
                    vsip_scalar_i alpha,
                    const vsip_vview_i *b,
                    const vsip_vview_bl *r);
void vsip_csvlne_i(
                    vsip_scalar_i alpha,
                    const vsip_cvview_i *b,
                    const vsip_vview_bl *r);
void vsip_svlne_f(
                    vsip_scalar_f alpha,
                    const vsip_vview_f *b,
                    const vsip_vview_bl *r);
void vsip_csvlne_f(
                    vsip_cscalar_f alpha,
                    const vsip_cvview_f *b,
                    const vsip_vview_bl *r);
void vsip_smlne_i(
                    vsip_scalar_i alpha,
                    const vsip_mview_i *b,
                    const vsip_mview_bl *r);
void vsip_scmlne_i(
                    vsip_scalar_i alpha,
                    const vsip_cmview_i *b,
                    const vsip_mview_bl *r);
void vsip_smlne_f(
                    vsip_scalar_f alpha,
                    const vsip_mview_f *b,
                    const vsip_mview_bl *r);
void vsip_scmlne_f(
                    vsip_cscalar_f alpha,
                    const vsip_cmview_f *b,
                    const vsip_mview_bl *r);
```

Selection Operations

The selection operations include functions which select an element or elements from an input vector/matrix or a pair of input vectors/matrices based on some logical comparison between two input vectors/matrices or a comparison between the elements of a single input vector/matrix and some selection criterion.

vsip_***Sclip_p***	Vector/Matrix Clip
vsip_***Svfirst_p***	Vector Find First Vector Index
vsip_***Sinvclip_p***	Vector/Matrix Inverted Clip
vsip_***Sindexbool***	Vector/Matrix Index a Boolean
vsip_***Smax_p***	Vector/Matrix Maximum
vsip_***Smaxmg_p***	Vector/Matrix Maximum Magnitude
vsip_***Scmaxmgsq_p***	Vector/Matrix Complex Max Magnitude Squared
vsip_***Scmaxmgsqval_p***	Vector/Matrix Complex Max Mag Squared Value
vsip_***Smaxmgval_p***	Vector/Matrix Maximum Magnitude Value
vsip_***Smaxval_p***	Vector/Matrix Maximum Value
vsip_***Smin_p***	Vector/Matrix Minimum
vsip_***Sminmg_p***	Vector/Matrix Minimum Magnitude
vsip_***Scminmgsq_p***	Vector/Matrix Complex Min Magnitude Squared
vsip_***Scminmgsqval_p***	Vector/Matrix Complex Min Mag Squared Value
vsip_***Sminmgval_p***	Vector/Matrix Minimum Magnitude Value
vsip_***Sminval_p***	Vector/Matrix Minimum Value

vsip_sclip_*p* Vector/Matrix Clip

Computes the generalized double clip, by element, of two vectors/matrices.

Functionality

For a vector/matrix **a**, filter each element according to the following rules, in order, producing the output vector/matrix **r**. Note that once a rule is met for an element the following rules are ignored.

$$r_j \leftarrow \begin{cases} 1: c_1; & \text{if } a_j \le t_1 \\ 2: a_j; & \text{if } a_j < t_2 \\ 3: c_2; & \text{otherwise} \end{cases} \quad \text{for } j = 0, 1, \cdots, N-1$$

or

$$r_{i,j} \leftarrow \begin{cases} 1: c_1; & \text{if } a_{i,j} \le t_1 \\ 2: a_{i,j}; & \text{if } a_{i,j} < t_2 \\ 3: c_2; & \text{otherwise} \end{cases} \quad \begin{matrix} \text{for } i = 0, 1, \cdots, M-1 \\ \text{for } j = 0, 1, \cdots, N-1 \end{matrix}$$

Prototypes

```
void vsip_vclip_f(
                    const vsip_vview_f *a,
                    vsip_scalar_f t1,
                    vsip_scalar_f t2,
                    vsip_scalar_f c1,
                    vsip_scalar_f c2,
                    const vsip_vview_f *r);
void vsip_vclip_i(
                    const vsip_vview_i *a,
                    vsip_scalar_i t1,
                    vsip_scalar_i t2,
                    vsip_scalar_i c1,
                    vsip_scalar_i c2,
                    const vsip_vview_i *r);
void vsip_mclip_f(
                    const vsip_mview_f *a,
                    vsip_scalar_f t1,
                    vsip_scalar_f t2,
                    vsip_scalar_f c1,
                    vsip_scalar_f c2,
                    const vsip_mview_f *r);
void vsip_mclip_i(
                    const vsip_mview_i *a,
                    vsip_scalar_i t1,
                    vsip_scalar_i t2,
                    vsip_scalar_i c1,
                    vsip_scalar_i c2,
                    const vsip_mview_i *r);
```

Arguments

`*a`	View of input vector/matrix
`t1`	Lower threshold
`t2`	Upper threshold
`c1`	Lower threshold clip value
`c2`	Upper threshold clip value
`*r`	View of output vector/matrix

Return Value

None.

Restrictions

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.
3. The input and output views must be identical views of the same block (in-place), or must not overlap.

Notes/References

The clipping rules are evaluated (in order) sequentially. Once a rule is met, the following rules are ignored.

The variables t_1 and t_2 are unrestricted; it is not an error if $t_2 < t_1$.

Examples

Single Sided Limiter

```
vsip_vview_sp *x, *y;
vsip_scalar_sp alpha;
…
vsip_vclip_sp(x, -VSIP_MAX_SP, alpha, -VSIP_MAX_SP,
alpha, y);
```

Double Sided Symmetric Limiter

```
vsip_vview_i16 *x, *y;
vsip_scalar_i16 alpha;
…
vsip_vclip_i16(x,- alpha, alpha, -alpha, alpha, y);
```

Bi-level Thresholder

```
vsip_vview_uc *x, *y;
vsip_scalar_uc alpha;
…
vsip_vclip_i16(x, alpha, alpha, 0, 1, y);
```

Clip off Hi & Low

```
vsip_vview_uc *x, *y;
vsip_scalar_uc alpha, beta;
…
vsip_vclip_i16(x, alpha, beta, 0, 0, y);
```

```
/* example of clip, double sided symmetric limiter */
#include<stdio.h>
#include "vsip.h"
#define L  9
#define PI  3.14159265359
int main()
{
    vsip_vview_d  *dataIn, *dataClip;
    vsip_init((void *)0);
    dataIn   = vsip_vcreate_d(L, VSIP_MEM_NONE);
    dataClip = vsip_vcreate_d(L, VSIP_MEM_NONE);
    /* make some data */
    vsip_vramp_d(0.0, (2 * PI)/(L - 1.0), dataIn);
    vsip_vcos_d(dataIn,dataIn);
    vsip_vclip_d(dataIn,-.8,.8,-.8,.8,dataClip);
    printf("clip Cosine between -.8 and .8\n   in   =>    out\n ");
    {int i; for(i=0; i<L; i++) printf("%7.4f => %7.4f\n",
              vsip_vget_d(dataIn,i),vsip_vget_d(dataClip,i));}
    vsip_finalize((void *)0);
    return 0;
} /* output clip Cosine between -.8 and .8
     in   =>    out
   1.0000 =>  0.8000
   0.7071 =>  0.7071
  -0.0000 => -0.0000
  -0.7071 => -0.7071
  -1.0000 => -0.8000
  -0.7071 => -0.7071
   0.0000 =>  0.0000
   0.7071 =>  0.7071
   1.0000 =>  0.8000  */
```

See Also

vsip_***s***invclip_***p***

vsip_*d*vfirst_*p* Find First Vector Index

Returns the index of the first element of a pair of vector view objects for which a user-specified binary scalar function, applied by element, returns *true*.

Functionality

Given a starting index *j*, a pair of vectors ***x*** and ***y***, and a user-specified binary function $f(x_j, y_j)$ returns:

Vector Index Return Value	When
j	$j \geq length$ where *length* is the length in elements of the vector view objects **x** and **y**
length	$f(x_j, y_j)$ is *false* for all vector indices *j*, such that $0 \leq j < length$ where *length* is the length in elements of the vector view objects **x** and **y**
First vector index *j* for which $f(x_j, y_j)$ is *true*	otherwise

Where $f(x_j, y_j)$ is a user-specified binary function that takes two scalar elements as arguments and returns a boolean value.

Prototypes

```
vsip_index vsip_vfirst_f(
                     vsip_index j,
                     vsip_bool (*f)(vsip_scalar_f, vsip_scalar_f),
                     const vsip_vview_f *x,
                     const vsip_vview_f *y);
vsip_index vsip_vfirst_i(
                     vsip_index j,
                     vsip_bool (*f)(vsip_scalar_i, vsip_scalar_i),
                     const vsip_vview_i *x,
                     const vsip_vview_i *y);
vsip_index vsip_vfirst_bl(
                     vsip_index j,
                     vsip_bool (*f)(vsip_scalar_bl, vsip_scalar_bl),
                     const vsip_vview_bl *x,
                     const vsip_vview_bl *y);
vsip_index vsip_vfirst_vi(
                     vsip_index j,
                     vsip_bool (*f)(vsip_index, vsip_index),
                     const vsip_vview_vi *x,
                     const vsip_vview_vi *y);
vsip_index vsip_vfirst_mi(
                     vsip_index j,
                     vsip_bool (*f)(vsip_scalar_mi, vsip_scalar_mi),
                     const vsip_vview_mi *x,
                     const vsip_vview_mi *y);
vsip_index vsip_vfirst_ti(
                     vsip_index j,
                     vsip_bool (*f)(vsip_scalar_ti, vsip_scalar_ti),
                     const vsip_vview_ti *x,
                     const vsip_vview_ti *y);
```

Arguments

`j`	User specified staring index of search
`*f()`	User specified binary function of two scalars, returning a boolean
`*x`	Vector view object of **x** operand
`*y`	Vector view object of **y** operand

Return Value

This function returns a vector index value of:

1. j, if j is greater than the length of the vector view arguments
2. The *length*, if $f(x_k, y_k)$ is *false* for all k, $j \le k < length$
3. The first vector index $k \ge j$, for which $f(x_k, y_k)$ is not *false*.

Restrictions

Errors

The arguments must conform to the following:

1. The vector views must all be the same size.
2. All view objects must be valid.
3. The pointer to the user-specified function must be valid – non-*null.*

Notes/References

There are no complex versions of this function. This is a consequence of supporting the implementation of complex blocks with split storage, which is not compatible with a `vsip_cscalar_`*p* data type.

Examples

See Also

vsip_sinvclip_*p* Vector/Matrix Inverted Clip

Computes the generalized inverted double clip, by element, of two vectors/matrices.

Functionality

For vector/matrix **a**, filter each element according to the following rules producing an output vector/matrix **r**. Note that once a rule is met for an element the following rules are ignored.

$$r_j \leftarrow \begin{cases} 1: a_j; & \text{if } a_j < \mathrm{t}_1 \\ 2: c_1; & \text{if } a_j < \mathrm{t}_2 \\ 3: c_2; & \text{if } a_j \le \mathrm{t}_3 \\ 4: a_j; & \text{otherwise} \end{cases} \quad \text{for } j = 0, 1, \cdots, N-1$$

or

$$r_{i,j} \leftarrow \begin{cases} 1: a_{i,j}; & \text{if } a_{i,j} < \mathrm{t}_1 \\ 2: c_1; & \text{if } a_{i,j} < \mathrm{t}_2 \\ 3: c_2; & \text{if } a_{i,j} \le \mathrm{t}_3 \\ 4: a_{i,j}; & \text{otherwise} \end{cases} \quad \begin{array}{l} \text{for } i = 0, 1, \cdots, M-1 \\ \text{for } j = 0, 1, \cdots, N-1 \end{array}$$

Prototypes

```
void vsip_vinvclip_f(
                      const vsip_vview_f *a,
                      vsip_scalar_f t1,
                      vsip_scalar_f t2,
                      vsip_scalar_f t3,
                      vsip_scalar_f c1,
                      vsip_scalar_f c2,
                      const vsip_vview_f *r);
void vsip_vinvclip_i(
                      const vsip_vview_i *a,
                      vsip_scalar_i t1,
                      vsip_scalar_i t2,
                      vsip_scalar_i t3,
                      vsip_scalar_i c1,
                      vsip_scalar_i c2,
                      const vsip_vview_i *r);
void vsip_minvclip_f(
                      const vsip_mview_f *a,
                      vsip_scalar_f t1,
                      vsip_scalar_f t2,
                      vsip_scalar_f t3,
                      vsip_scalar_f c1,
                      vsip_scalar_f c2,
                      const vsip_mview_f *r);
```

```
void vsip_minvclip_i(
                      const vsip_mview_i *a,
                      vsip_scalar_i t1,
                      vsip_scalar_i t2,
                      vsip_scalar_i t3,
                      vsip_scalar_i c1,
                      vsip_scalar_i c2,
                      const vsip_mview_i *r);
```

Arguments

`*a`	View of input vector/matrix
`t1`	Lower threshold
`t2`	Mid threshold
`t3`	Upper threshold
`c1`	Lower threshold clip value
`c2`	Upper threshold clip value
`*r`	View of output vector/matrix

Return Value

None.

Restrictions

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.
3. The input and output views must be identical views of the same block (in-place), or must not overlap.

Notes/References

The clipping rules are evaluated (in order) sequentially. Once a rule is met, the following rules are ignored.

The variables t_1, t_2, and t_3 are unrestricted; it is not an error if the relationship $t_1 \le t_2 \le t_3$ does not hold.

Examples

Clip out small values

```
vsip_vview_i16 *x, *y;
vsip_scalar_i16 alpha;
…
vsip_vinvclip_i16(x,-alpha, 0, alpha, -alpha, alpha,
y);
```

See Also

`vsip_sclip_p`

vsip_sindexbool Vector/Matrix Index a Boolean

Computes an index vector of the indices of the *non-false* elements of the boolean vector/matrix, and returns the number of *non-false* elements.

Functionality

Returns an index vector of the indices of the *non-false* elements of the boolean vector **b**, or boolean matrix **B**. The index vector is ordered; lower indices appear before higher indices. For a matrix, elements appear in the index vector in accord with the major ordering of the matrix. If no *non-false* elements are found, the *index* vector is unmodified, otherwise the *length* of the vector view is set equal to the number of *non-false* elements.

The Return value is the number of *non-false* elements.

Prototypes

```
vsip_length vsip_vindexbool (
                      const vsip_vview_bl *b,
                      vsip_vview_vi *index);
vsip_length vsip_mindexbool (
                      const vsip_mview_bl *B,
                      vsip_vview_mi *index);
```

Arguments

`*b (*B)`	View of input boolean vector **b** (matrix **B**)
`*index`	View of output vector/matrix index vector **index**.

Return Value

The Return value is the number of *non-false* elements.

Restrictions

The length of the return index vector is dependent on the number of non-false values in the boolean object. The user must make sure that the index vector's length attribute is greater than or equal to the maximum number of non-false elements expected. If the index vector is re-used for multiple calls, its length may change after each call; therefore, the user should reset the length to the maximum value.

No in-place operations are allowed.

Errors

The arguments must conform to the following:

1. All view objects must be valid.
2. The index vector must be of length greater than or equal to the number of *non-false* boolean elements.

Notes/References

VSIPL does not support zero length vectors. It is important to test the return value for zero to handle the case of no *non-false* elements.

Examples

For example of `vsip_`***s***`indexbool_`***p*** see `vsip_`***ds***`gather_`***p*** example.

See Also

`vsip_`***ds***`gather_`***p***, and `vsip_`***ds***`scatter_`***p***

vsip_smax_*p* Vector/Matrix Maximum

Computes the maximum, by element, of two vectors/matrices.

Functionality

$r_j \leftarrow \max\{a_j, b_j\}$ for $j = 0, 1, \ldots, N\text{-}1$

$r_{i,j} \leftarrow \max\{a_{i,j}, b_{i,j}\}$ for $i = 0, 1, \ldots, M\text{-}1$; for $j = 0, 1, \ldots, N\text{-}1$

Prototypes

```
void vsip_vmax_f(
                      const vsip_vview_f *a,
                      const vsip_vview_f *b,
                      const vsip_vview_f *r);
void vsip_mmax_f(
                      const vsip_mview_f *a,
                      const vsip_mview_f *b,
                      const vsip_mview_f *r);
```

Arguments

`*a`	View of input vector/matrix
`*b`	View of input vector/matrix
`*r`	View of output vector/matrix

Return Value

None.

Restrictions

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.
3. The input and output views must be identical views of the same block (in-place), or must not overlap.

Notes/References

Examples

```
/* example of max and min */
#include<stdio.h>
#include "vsip.h"

#define L  9
#define PI  3.14159265359

int main()
{
    int i = 0;
    vsip_vview_d*  dataA;
    vsip_vview_d*  dataB;
    vsip_vview_d*  dataMax;
    vsip_vview_d*  dataMin;
    vsip_vview_d*  dataRamp;
```

```
    vsip_init((void *)0);
    dataA    = vsip_vcreate_d(L, VSIP_MEM_NONE);
    dataB    = vsip_vcreate_d(L, VSIP_MEM_NONE);
    dataMax  = vsip_vcreate_d(L, VSIP_MEM_NONE);
    dataMin  = vsip_vcreate_d(L, VSIP_MEM_NONE);
    dataRamp = vsip_vcreate_d(L, VSIP_MEM_NONE);

    /* Make up some data */
    vsip_vramp_d(0.0, (2 * PI)/((double)(L-1)), dataRamp);
    vsip_vsin_d(dataRamp, dataA);
    vsip_vcos_d(dataRamp, dataB);

    /* find the Max and Min of dataA and dataB */
    vsip_vmax_d(dataA,dataB,dataMax);
    vsip_vmin_d(dataA,dataB,dataMin);

    /* print out the results */
    printf("   A       B       Max     Min\n");
    for(i = 0; i < L; i++)
        printf("%7.4f %7.4f %7.4f %7.4f\n",
            vsip_vget_d(dataA,i), vsip_vget_d(dataB,i),
            vsip_vget_d(dataMax,i), vsip_vget_d(dataMin,i));
    /* recover allocated memory */
    vsip_blockdestroy_d(vsip_vdestroy_d(dataA));
    vsip_blockdestroy_d(vsip_vdestroy_d(dataB));
    vsip_blockdestroy_d(vsip_vdestroy_d(dataMax));
    vsip_blockdestroy_d(vsip_vdestroy_d(dataRamp));
    vsip_finalize((void *)0);
    return 0;
}
/* output */
/*   A       B       Max     Min
   0.0000  1.0000  1.0000  0.0000
   0.7071  0.7071  0.7071  0.7071
   1.0000 -0.0000  1.0000 -0.0000
   0.7071 -0.7071  0.7071 -0.7071
  -0.0000 -1.0000 -0.0000 -1.0000
  -0.7071 -0.7071 -0.7071 -0.7071
  -1.0000  0.0000  0.0000 -1.0000
  -0.7071  0.7071  0.7071 -0.7071 */
```

See Also

vsip_*s*maxmg_*p*, vsip_*s*cmaxmgsq_*p*, vsip_*s*cmaxmgsqval_*p*, vsip_*s*maxmgval_*p*, vsip_*s*maxval_*p*, vsip_*s*min_*p*, vsip_*s*minmg_*p*, vsip_*s*cminmgsq_*p*, vsip_*s*cminmgsqval_*p*, vsip_*s*minmgval_*p*, and vsip_*s*minval_*p*

vsip_smaxmg_*p* Vector/Matrix Maximum Magnitude

Computes the maximum magnitude (absolute value), by element, of two vectors/matrices.

Functionality

$r_j \leftarrow \max\{|a_j|,|b_j|\}$ for $j = 0, 1, \ldots, N\text{-}1$

$r_{i,j} \leftarrow \max\{|a_{i,j}|,|b_{i,j}|\}$ for $i = 0, 1, \ldots, M\text{-}1$; for $j = 0, 1, \ldots, N\text{-}1$

Prototypes

```
void vsip_vmaxmg_f(
                      const vsip_vview_f *a
                      const vsip_vview_f *b,
                      const vsip_vview_f *r);
void vsip_mmaxmg_f(
                      const vsip_mview_f *a
                      const vsip_mview_f *b,
                      const vsip_mview_f *r);
```

Arguments

`*a`	View of input vector/matrix
`*b`	View of input vector/matrix
`*r`	View of output vector/matrix

Return Value

None.

Restrictions

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.
3. The input and output views must be identical views of the same block (in-place), or must not overlap.

Notes/References

Examples

```
/* example of maxmg and minmg */
#include<stdio.h>
#include "vsip.h"

#define L  9
#define PI  3.14159265359

int main()
{   int i = 0;
    vsip_vview_d*  dataA;
    vsip_vview_d*  dataB;
    vsip_vview_d*  dataMax;
    vsip_vview_d*  dataMin;
    vsip_vview_d*  dataRamp;
```

```
    vsip_init((void *)0);
    dataA    = vsip_vcreate_d(L, VSIP_MEM_NONE);
    dataB    = vsip_vcreate_d(L, VSIP_MEM_NONE);
    dataMax  = vsip_vcreate_d(L, VSIP_MEM_NONE);
    dataMin  = vsip_vcreate_d(L, VSIP_MEM_NONE);
    dataRamp = vsip_vcreate_d(L, VSIP_MEM_NONE);

    /* Make up some data */
    vsip_vramp_d(0.0, (2 * PI)/((double)(L-1)), dataRamp);
    vsip_vsin_d(dataRamp, dataA);
    vsip_vcos_d(dataRamp, dataB);

    /* find the Maximum Magnitde dataA or dataB*/
    vsip_vmaxmg_d(dataA,dataB,dataMax);
    vsip_vminmg_d(dataA,dataB,dataMin);

    /* print out the results */
    printf("   A       B     Max Mag    Min Mag\n");
    for(i = 0; i < L; i++)
        printf("%7.4f %7.4f %7.4f    %7.4f\n",
            vsip_vget_d(dataA,i), vsip_vget_d(dataB,i),
            vsip_vget_d(dataMax,i), vsip_vget_d(dataMin,i));

    /* recover allocated memory */
    vsip_blockdestroy_d(vsip_vdestroy_d(dataA));
    vsip_blockdestroy_d(vsip_vdestroy_d(dataB));
    vsip_blockdestroy_d(vsip_vdestroy_d(dataMax));
    vsip_blockdestroy_d(vsip_vdestroy_d(dataMin));
    vsip_blockdestroy_d(vsip_vdestroy_d(dataRamp));
    vsip_finalize((void *)0);
    return 0;
}
/* output */
/*   A       B     Max Mag    Min Mag
   0.0000  1.0000  1.0000     0.0000
   0.7071  0.7071  0.7071     0.7071
   1.0000 -0.0000  1.0000     0.0000
   0.7071 -0.7071  0.7071     0.7071
  -0.0000 -1.0000  1.0000     0.0000
  -0.7071 -0.7071  0.7071     0.7071
  -1.0000  0.0000  1.0000     0.0000
  -0.7071  0.7071  0.7071     0.7071
   0.0000  1.0000  1.0000     0.0000 */
```

See Also

For complex data use `vsip_`***`scmaxmgsq`***`_`***`p`***.

`vsip_`*s*`max_`*p*, `vsip_`*s*`cmaxmgsq_`*p*, `vsip_`*s*`cmaxmgsqval_`*p*, `vsip_`*s*`maxmgval_`*p*, `vsip_`*s*`maxval_`*p*, `vsip_`*s*`min_`*p*, `vsip_`*s*`minmg_`*p*, `vsip_`*s*`cminmgsq_`*p*, `vsip_`*s*`cminmgsqval_`*p*, `vsip_`*s*`minmgval_`*p*, and `vsip_`*s*`minval_`*p*

vsip_scmaxmgsq_*p* Vector/Matrix Complex Max Magnitude Squared

Computes the maximum magnitude squared, by element, of two complex vectors/matrices.

Functionality

$$r_j \leftarrow \max\left\{ |a_j|^2, |b_j|^2 \right\} \qquad \text{for } j = 0, 1, \ldots, N\text{-}1$$

$$r_{i,j} \leftarrow \max\left\{ |a_{i,j}|^2, |b_{i,j}|^2 \right\} \qquad \text{for } i = 0, 1, \ldots, M\text{-}1; \text{ for } j = 0, 1, \ldots, N\text{-}1$$

Prototypes

```
void vsip_vcmaxmgsq_f(
                      const vsip_cvview_f *a
                      const vsip_cvview_f *b,
                      const vsip_vview_f *r);
void vsip_mcmaxmgsq_f(
                      const vsip_cmview_f *a
                      const vsip_cmview_f *b,
                      const vsip_mview_f *c);
```

Arguments

`*a`	View of input vector/matrix
`*b`	View of input vector/matrix
`*r`	View of output vector/matrix

Return Value

None.

Restrictions

In-place operation for this function means the output vector is either a real view, or an imaginary view, of one of the input vectors. No in-place operation is defined for an output vector which contains both real and imaginary components of an input vector, or which does not exactly overlap a real view or an imaginary view of one of the input vectors.

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.
3. The input and output views must be identical views of the same block (in-place), or must not overlap.

Notes/References

Examples

```
/* example of cmaxmgsq and cminmgsq */
#include<stdio.h>
#include "vsip.h"
#define L  9
#define PI  3.14159265359
int main()
{   /* Make up some data space */
    vsip_cvview_d*  cdataA;
    vsip_cvview_d*  cdataB;
    vsip_vview_d*   dataReA;
```

```
    vsip_vview_d*   dataImA;
    vsip_vview_d*   dataReB;
    vsip_vview_d*   dataImB;
    vsip_vview_d*   dataMaxmgsq;
    vsip_vview_d*   dataMinmgsq;
    int i = 0;

    vsip_init((void *)0);
    cdataA      = vsip_cvcreate_d(L, VSIP_MEM_NONE);
    cdataB      = vsip_cvcreate_d(L, VSIP_MEM_NONE);
    dataReA     = vsip_vrealview_d(cdataA);
    dataImA     = vsip_vimagview_d(cdataA);
    dataReB     = vsip_vrealview_d(cdataB);
    dataImB     = vsip_vimagview_d(cdataB);
    dataMaxmgsq = vsip_vcreate_d(L, VSIP_MEM_NONE);
    dataMinmgsq = vsip_vcreate_d(L, VSIP_MEM_NONE);
    vsip_vramp_d(0.001, (2 * PI)/((double)(L-1)), dataImB);
    vsip_vsin_d(dataImB, dataReA); vsip_vcos_d(dataImB, dataReB);
    vsip_vsqrt_d(dataImB, dataImA); vsip_vrsqrt_d(dataImB, dataImB);

    /* find the Maximum Magnitude Sq. of cdataA and cdataB*/
    vsip_vcmaxmgsq_d(cdataA,cdataB,dataMaxmgsq);
    vsip_vcminmgsq_d(cdataA,cdataB,dataMinmgsq);

    /* print out the input */
    printf("         A               B          Max Mag Sq  Min Mag Sq\n");
    for(i = 0; i < L; i++)
        printf("(%7.4f, %7.4f) (%7.4f, %7.4f) => %7.4f   %7.4f\n",
            vsip_vget_d(dataReA,i), vsip_vget_d(dataImA,i),
            vsip_vget_d(dataReB,i), vsip_vget_d(dataImB,i),
            vsip_vget_d(dataMaxmgsq,i),vsip_vget_d(dataMinmgsq,i));
    vsip_vdestroy_d(dataReA); vsip_vdestroy_d(dataImA);
    vsip_vdestroy_d(dataReB); vsip_vdestroy_d(dataImB);
    vsip_cblockdestroy_d(vsip_cvdestroy_d(cdataA));
    vsip_cblockdestroy_d(vsip_cvdestroy_d(cdataB));
    vsip_blockdestroy_d(vsip_vdestroy_d(dataMaxmgsq));
    vsip_blockdestroy_d(vsip_vdestroy_d(dataMinmgsq));
    vsip_finalize((void *)0);
    return 0;
}/*         A                  B           Max Mag Sq  Min Mag Sq
   ( 0.0010,  0.0316) ( 1.0000, 31.6228) => 1001.0000    0.0010
   ( 0.7078,  0.8868) ( 0.7064,  1.1277) =>  1.7706    1.2874
   ( 1.0000,  1.2537) (-0.0010,  0.7976) =>  2.5718    0.6362
   ( 0.7064,  1.5353) (-0.7078,  0.6513) =>  2.8562    0.9252
   (-0.0010,  1.7727) (-1.0000,  0.5641) =>  3.1426    1.3182
   (-0.7078,  1.9819) (-0.7064,  0.5046) =>  4.4290    0.7536
   (-1.0000,  2.1710) ( 0.0010,  0.4606) =>  5.7134    0.2122
   (-0.7064,  2.3449) ( 0.7078,  0.4264) =>  5.9978    0.6829
   ( 0.0010,  2.5068) ( 1.0000,  0.3989) =>  6.2842    1.1591 */
```

See Also

For real data use `vsip_`***s***`maxmg_`***p***.

`vsip_`***s***`max_`***p***, `vsip_`***s***`maxmg_`***p***, `vsip_`***s***`cmaxmgsqval_`***p***, `vsip_`***s***`maxmgval_`***p***, `vsip_`***s***`maxval_`***p***, `vsip_`***s***`min_`***p***, `vsip_`***s***`minmg_`***p***, `vsip_`***s***`cminmgsq_`***p***, `vsip_`***s***`cminmgsqval_`***p***, `vsip_`***s***`minmgval_`***p***, and `vsip_`***s***`minval_`***p***

vsip_scmaxmgsqval_*p* Vector/Matrix Complex Max Mag Squared Value

Returns the index and value of the maximum magnitude squared of the elements of a complex vector/matrix. The index is returned by reference as one of the arguments.

Functionality

$\max \leftarrow |a_0|^2$; index $\leftarrow 0$;
for $j = 1, 2, \ldots, N\text{-}1$
if $|a_j|^2 < \max$ then
$\max \leftarrow |a_j|^2$; index $\leftarrow j$;

$\max \leftarrow |a_{0,0}|^2$; index $\leftarrow (0, 0)$;
for $i = 0, 1, \ldots, M\text{-}1$
for $j = 0, 1, \ldots, N\text{-}1$
if $|a_{i,j}|^2 < \max$ then
$\max \leftarrow |a_{i,j}|^2$; index $\leftarrow (i, j)$;

Where: $|a|^2 \equiv (\mathrm{Re}(a))^2 + (\mathrm{Im}(a))^2$

Prototypes

```
vsip_scalar_f vsip_vcmaxmgsqval_f(
                      const vsip_cvview_f *a,
                      vsip_scalar_vi *index);
vsip_scalar_f vsip_mcmaxmgsqval_f(
                      const vsip_cmview_f *a,
                      vsip_scalar_mi *index);
```

Arguments

`*a`	View of input vector/matrix
`*index`	Pointer to *index*, if *null* the index is not returned

Return Value

Returns the maximum magnitude squared value of the elements. The index is returned using the *index* pointer, if non-*null*.

Restrictions

Errors

The arguments must conform to the following:

1. All view objects must be valid.

Notes/References

If the vector/matrix has more than one element with identical maximum magnitude squared values, the index of the first maximum magnitude squared is returned in the index.

Examples

```
/* Example of min and max mgsqval */
#include<stdio.h>
#include "vsip.h"

#define PI 3.1415926535
#define L 7                    /* length */

int main()
{   int i;
    vsip_vview_d*  dataMagsq;
    vsip_cvview_d* dataCmplx;
    vsip_vview_d*  dataRe;
    vsip_vview_d*  dataIm;
```

```
    vsip_cscalar_d cscalar;
    vsip_scalar_d  mgsqval;
    vsip_scalar_vi index = 0;

    vsip_init((void *)0);
    dataMagsq = vsip_vcreate_d(L, VSIP_MEM_NONE);
    dataCmplx = vsip_cvcreate_d(L, VSIP_MEM_NONE);
    dataRe    = vsip_vrealview_d(dataCmplx);
    dataIm    = vsip_vimagview_d(dataCmplx);

    /* Make up some data to find the magnitude of */
    vsip_vramp_d(0.0, (2.0 * PI / (double) (L - 1)), dataRe);
    vsip_vsin_d(dataRe,dataRe);
    vsip_vramp_d(0.0, (3.0 * PI / (double) (L - 1)), dataIm);
    vsip_vcos_d(dataIm,dataIm);
    vsip_vdestroy_d(dataRe); /*don't need these views any more*/
    vsip_vdestroy_d(dataIm);

    /* Find the Magnitude */
    vsip_vcmagsq_d(dataCmplx,dataMagsq);

    /*now print out dataCmplex an its magnitude squared*/
    printf("  complex vector   => Mag Squared\n");
    for(i=0; i<L; i++){
        cscalar = vsip_cvget_d(dataCmplx, (vsip_scalar_vi) i);
        printf("(%7.4f, %7.4f) => %7.4f\n",
             vsip_real_d(cscalar), vsip_imag_d(cscalar),
             vsip_vget_d(dataMagsq, i));
    }
    /* now find the maximum and minimum value and its index */
    mgsqval = vsip_vcmaxmgsqval_d(dataCmplx, &index);
    printf("Max Mag Squared  of %7.4f at index %i\n", mgsqval,(int) index);
    mgsqval = vsip_vcminmgsqval_d(dataCmplx, &index);
    printf("Min Mag Squared  of %7.4f at index %i\n",
                     mgsqval,(int) index);

    /*destroy the vector views and any associated blocks */
    vsip_blockdestroy_d(vsip_vdestroy_d(dataMagsq));
    vsip_cblockdestroy_d(vsip_cvdestroy_d(dataCmplx));
    vsip_finalize((void *)0);
    return 0;
}
/*  complex vector   => Mag Squared
   ( 0.0000,  1.0000) =>  1.0000
   ( 0.8660,  0.0000) =>  0.7500
   ( 0.8660, -1.0000) =>  1.7500
   ( 0.0000, -0.0000) =>  0.0000
   (-0.8660,  1.0000) =>  1.7500
   (-0.8660,  0.0000) =>  0.7500
   (-0.0000, -1.0000) =>  1.0000
   Max Mag Squared  of  1.7500 at index 2
   Min Mag Squared  of  0.0000 at index 3 */
```

See Also

`vsip_`*s*`max_`*p*, `vsip_`*s*`maxmg_`*p*, `vsip_`*s*`cmaxmgsq_`*p*, `vsip_`*s*`maxmgval_`*p*, `vsip_`*s*`maxval_`*p*, `vsip_`*s*`min_`*p*, `vsip_`*s*`minmg_`*p*, `vsip_`*s*`cminmgsq_`*p*, `vsip_`*s*`cminmgsqval_`*p*, `vsip_`*s*`minmgval_`*p*, and `vsip_`*s*`minval_`*p*

vsip_smaxmgval_*p* Vector/Matrix Maximum Magnitude Value

Returns the index and value of the maximum absolute value of the elements of a vector/matrix. The index is returned by reference as one of the arguments.

Functionality

max ← $|a_0|$; index ← 0;
for j = 1, 2, …, N-1
if $|a_j|$ < max then
max ← $|a_j|$; index ← j;

max ← $|a_{0,0}|$; index ← (0, 0);
for i = 0, 1, …, M-1
for j = 0, 1, …, N-1
if $|a_{i,j}|$ < max then
max ← $|a_{i,j}|$; index ← (i, j);

Prototypes

```
vsip_scalar_f vsip_vmaxmgval_f(
                      const vsip_vview_f *a,
                      vsip_scalar_vi *index);
vsip_scalar_f vsip_mmaxmgval_f(
                      const vsip_mview_f *a,
                      vsip_scalar_mi *index);
```

Arguments

`*a`	View of input vector/matrix
`*index`	Pointer to *index*, if *null* the index is not returned

Return Value

Returns the maximum absolute value of the elements. The index is returned using the *index* pointer, if non-*null*.

Restrictions

Errors

The arguments must conform to the following:

1. All view objects must be valid.

Notes/References

If the vector/matrix has more than one element with identical maximum absolute value values, the index of the first maximum absolute value is returned in the index.

Examples

```
/* Example of max and min mgval*/
#include<stdio.h>
#include "vsip.h"
#define PI 3.1415926535
#define L 7                    /* length */

int main()
{
    int i;
    vsip_vview_d*  data;
    vsip_scalar_d  mgval;
    vsip_scalar_vi index = 0;

    vsip_init((void *)0);
    data = vsip_vcreate_d(L, VSIP_MEM_NONE);
```

```
    /* Make up some data to find the magnitude of */
    vsip_vramp_d(0.0, (2.0 * PI / (double) (L - 1)), data);
    vsip_vsin_d(data,data);
    printf(" data \n");
    for(i=0; i<L; i++)
        printf("%7.4f\n", vsip_vget_d(data, i));
    /* now find the max and min magnitude value and their index */
    mgval = vsip_vmaxmgval_d(data, &index);
    printf("Max Mag of %7.4f at index %i\n",
                      mgval,(int) index);
    mgval = vsip_vminmgval_d(data, &index);
    printf("Min Mag of %7.4f at index %i\n",
                      mgval,(int) index);

    /*destroy the vector views and any associated blocks */
    vsip_blockdestroy_d(vsip_vdestroy_d(data));
    vsip_finalize((void *)0);
    return 0;
}
/* output */
/* 0.0000
   0.8660
   0.8660
   0.0000
  -0.8660
  -0.8660
  -0.0000
  Max Mag of  0.8660 at index 5
  Min Mag of  0.0000 at index 0 */
```

See Also

vsip_*s*max_*p*, vsip_*s*maxmg_*p*, vsip_*s*cmaxmgsq_*p*, vsip_*s*cmaxmgsqval_*p*, vsip_*s*maxval_*p*, vsip_*s*min_*p*, vsip_*s*minmg_*p*, vsip_*s*cminmgsq_*p*, vsip_*s*cminmgsqval_*p*, vsip_*s*minmgval_*p*, and vsip_*s*minval_*p*

vsip_smaxval_p Vector/Matrix Maximum Value

Returns the index and value of the maximum value of the elements of a vector/matrix. The index is returned by reference as one of the arguments.

Functionality

max ← a_0 ; index ← 0;
for j = 1, 2, …, N-1
if a_j < max then
max ← a_j ; index ← j;

max ← $a_{0,0}$; index ← (0, 0);
for i = 0, 1, …, M-1
for j = 0, 1, …, N-1
if $a_{i,j}$ < max then
max ← $a_{i,j}$; index ← (i, j);

Prototypes

```
vsip_scalar_f vsip_vmaxval_f(
                      const vsip_vview_f *a,
                      vsip_scalar_vi *index);
vsip_scalar_f vsip_mmaxval_f(
                      const vsip_mview_f *a,
                      vsip_scalar_mi *index);
```

Arguments

`*a`	View of input vector/matrix
`*index`	Pointer to index, if null the index is not returned

Return Value

Returns the maximum value of the elements. The index is returned using the *index* pointer, if non-*null*.

Restrictions

Errors

The arguments must conform to the following:

1. All view objects must be valid.

Notes/References

If the vector/matrix has more than one element with identical maximum values the index of the first maximum is returned in the index.

Examples

```
/* Example of  maxval and minval */
#include<stdio.h>
#include "vsip.h"

#define PI 3.1415926535
#define L 9                    /* length */

int main()
{
    int i;
    vsip_vview_d*  data;
    vsip_scalar_vi index  = 0;
    vsip_scalar_d  maxval = 0, minval = 0;

    vsip_init((void *)0);
```

```
    data = vsip_vcreate_d(L, VSIP_MEM_NONE);

    /* Make up some data */
    vsip_vramp_d(0.0, (2.0 * PI / (double) (L - 1)), data);
    vsip_vsin_d(data, data);

    /*now print out data */
    printf("Input Vector  \n");
    for(i=0; i<L; i++){
        printf("%7.4f \n", vsip_vget_d(data, i));
    }
    /* Now find the maximum and minimum value and their indices */
    maxval = vsip_vmaxval_d(data, &index);
    printf("Max Value of %7.4f at index %i\n",
                     maxval,(int) index);
    minval = vsip_vminval_d(data, &index);
    printf("Min Value of %7.4f at index %i\n",
                     minval,(int) index);

    /* Destroy the vector views and any associated blocks */
    vsip_blockdestroy_d(vsip_vdestroy_d(data));
    vsip_finalize((void *)0);
    return 0;
}
/*output*/
/* Input Vector
   0.0000
   0.7071
   1.0000
   0.7071
   0.0000
  -0.7071
  -1.0000
  -0.7071
  -0.0000
  Max Value of  1.0000 at index 2
  Min Value of -1.0000 at index 6 */
```

See Also

vsip_*s*max_*p*, vsip_*s*maxmg_*p*, vsip_*s*cmaxmgsq_*p*, vsip_*s*cmaxmgsqval_*p*, vsip_*s*maxmgval_*p*, vsip_*s*min_*p*, vsip_*s*minmg_*p*, vsip_*s*cminmgsq_*p*, vsip_*s*cminmgsqval_*p*, vsip_*s*minmgval_*p*, and vsip_*s*minval_*p*

vsip_smin_*p* Vector/Matrix Minimum

Computes the minimum, by element, of two vectors/matrices.

Functionality

$r_j \leftarrow \min\{a_j, b_j\}$ for $j = 0, 1, \ldots, N\text{-}1$

$r_{i,j} \leftarrow \min\{a_{i,j}, b_{i,j}\}$ for $i = 0, 1, \ldots, M\text{-}1$; for $j = 0, 1, \ldots, N\text{-}1$

Prototypes

```
void vsip_vmin_f(
                    const vsip_vview_f *a,
                    const vsip_vview_f *b,
                    const vsip_vview_f *r);
void vsip_mmin_f(
                    const vsip_mview_f *a,
                    const vsip_mview_f *b,
                    const vsip_mview_f *r);
```

Arguments

`*a`	View of input vector/matrix
`*b`	View of input vector/matrix
`*r`	View of output vector/matrix

Return Value

None.

Restrictions

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.
3. The input and output views must be identical views of the same block (in-place), or must not overlap.

Notes/References

Examples

For example of vsip_*s*min_*p* see example with vsip_*s*max_*p*.

See Also

`vsip_`*s*`max_`*p*, `vsip_`*s*`maxmg_`*p*, `vsip_`*s*`cmaxmgsq_`*p*, `vsip_`*s*`cmaxmgsqval_`*p*, `vsip_`*s*`maxmgval_`*p*, `vsip_`*s*`maxval_`*p*, `vsip_`*s*`minmg_`*p*, `vsip_`*s*`cminmgsq_`*p*, `vsip_`*s*`cminmgsqval_`*p*, `vsip_`*s*`minmgval_`*p*, and `vsip_`*s*`minval_`*p*

vsip_sminmg_*p* Vector/Matrix Minimum Magnitude

Computes the minimum magnitude (absolute value), by element, of two vectors/matrices.

Functionality

$r_j \leftarrow \min\{|a_j|,|b_j|\}$ for $j = 0, 1, \ldots, N\text{-}1$

$r_{i,j} \leftarrow \min\{|a_{i,j}|,|b_{i,j}|\}$ for $i = 0, 1, \ldots, M\text{-}1$; for $j = 0, 1, \ldots, N\text{-}1$

Prototypes

```
void vsip_vminmg_f(
                    const vsip_vview_f *a,
                    const vsip_vview_f *b,
                    const vsip_vview_f *r);
void vsip_mminmg_f(
                    const vsip_mview_f *a,
                    const vsip_mview_f *b,
                    const vsip_mview_f *r);
```

Arguments

`*a`	View of input vector/matrix
`*b`	View of input vector/matrix
`*r`	View of output vector/matrix

Return Value

None.

Restrictions

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.
3. The input and output views must be identical views of the same block (in-place), or must not overlap.

Notes/References

Examples

For example of `vsip_sminmg_p` see example with `vsip_smaxmg_p`.

See Also

For complex data use `vsip_scminmgsq_p`.

`vsip_smax_p`, `vsip_smaxmg_p`, `vsip_scmaxmgsq_p`, `vsip_scmaxmgsqval_p`, `vsip_smaxmgval_p`, `vsip_smaxval_p`, `vsip_smin_p`, `vsip_scminmgsq_p`, `vsip_scminmgsqval_p`, `vsip_sminmgval_p`, and `vsip_sminval_p`

vsip_scminmgsq_*p* Vector/Matrix Complex Min Magnitude Squared

Computes the minimum magnitude squared, by element, of two complex vectors/matrices.

Functionality

$$r_j \leftarrow \min\left\{ |a_j|^2, |b_j|^2 \right\} \qquad \text{for } j = 0, 1, \ldots, N\text{-}1$$

$$r_{i,j} \leftarrow \min\left\{ |a_{i,j}|^2, |b_{i,j}|^2 \right\} \qquad \text{for } i = 0, 1, \ldots, M\text{-}1; \text{ for } j = 0, 1, \ldots, N\text{-}1$$

Prototypes

```
void vsip_vcminmgsq_f(
                      const vsip_cvview_f *a
                      const vsip_cvview_f *b,
                      const vsip_vview_f *r);
void vsip_mcminmgsq_f(
                      const vsip_cmview_f *a
                      const vsip_cmview_f *b,
                      const vsip_mview_f *r);
```

Arguments

`*a`	View of input vector/matrix
`*b`	View of input vector/matrix
`*r`	View of output vector/matrix

Return Value

None.

Restrictions

In-place operation for this function means the output vector is either a real view, or an imaginary view, of one of the input vectors. No in-place operation is defined for an output vector which contains both real and imaginary components of an input vector, or which does not exactly overlap a real view or an imaginary view of one of the input vectors.

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.
3. The input and output views must be identical views of the same block (in-place), or must not overlap.

Notes/References

Examples

For example of `vsip_scminmgsq_p` see example of `vsip_scmaxmgsq_p`.

See Also

For real data use `vsip_sminmg_p`.

`vsip_smax_p`, `vsip_smaxmg_p`, `vsip_scmaxmgsq_p`, `vsip_scmaxmgsqval_p`, `vsip_smaxmgval_p`, `vsip_smaxval_p`, `vsip_smin_p`, `vsip_sminmg_p`, `vsip_scminmgsqval_p`, `vsip_sminmgval_p`, and `vsip_sminval_p`

vsip_scminmgsqval_p Vector/Matrix Complex Min Mag Squared Value

Returns the index and value of the minimum magnitude squared of the elements of a complex vector/matrix. The index is returned by reference as one of the arguments.

Functionality

$\min \leftarrow |a_0|^2$; index $\leftarrow 0$;
for $j = 1, 2, \ldots, N\text{-}1$
if $|a_j|^2 < \min$ then
$\min \leftarrow |a_j|^2$; index $\leftarrow j$;

$\min \leftarrow |a_{0,0}|^2$; index $\leftarrow (0, 0)$;
for $i = 0, 1, \ldots, M\text{-}1$
for $j = 0, 1, \ldots, N\text{-}1$
if $|a_{i,j}|^2 < \min$ then
$\min \leftarrow |a_{i,j}|^2$; index $\leftarrow (i, j)$;

Where: $|a|^2 \equiv (\mathrm{Re}(a))^2 + (\mathrm{Im}(a))^2$

Prototypes

```
vsip_scalar_f vsip_vcminmgsqval_f(
                    const vsip_cvview_f *a,
                    vsip_scalar_vi *index);
vsip_scalar_f vsip_mcminmgsqval_f(
                    const vsip_cmview_f *a,
                    vsip_scalar_mi *index);
```

Arguments

`*a`	View of input vector/matrix
`*index`	Pointer to *index*, if *null* the index is not returned

Return Value

Returns the minimum magnitude squared value of the elements. The index is returned using the *index* pointer, if non-*null*.

Restrictions

Errors

The arguments must conform to the following:

1. All view objects must be valid.

Notes/References

If the vector/matrix has more than one element with identical maximum magnitude squared values, the index of the first minimum magnitude squared is returned in the index.

Examples

For example of `vsip_scminmgsqval_p` see `vsip_scmaxmgsqval_p`.

See Also

`vsip_smax_p`, `vsip_smaxmg_p`, `vsip_scmaxmgsq_p`, `vsip_scmaxmgsqval_p`, `vsip_smaxmgval_p`, `vsip_smaxval_p`, `vsip_smin_p`, `vsip_sminmg_p`, `vsip_scminmgsq_p`, `vsip_sminmgval_p`, and `vsip_sminval_p`

vsip_sminmgval_p Vector/Matrix Minimum Magnitude Value

Returns the index and value of the minimum absolute value of the elements of a vector/matrix. The index is returned by reference as one of the arguments.

Functionality

min ← $|a_0|$; index ← 0;
for j = 1, 2, …, N-1
if $|a_j|$ < min then
min ←$|a_j|$; index ← j;

min ← $|a_{0,0}|$; index ← (0, 0);
for i = 0, 1, …, M-1
for j = 0, 1, …, N-1
if $|a_{i,j}|$ < min then
min ←$|a_{i,j}|$; index ← (i, j);

Prototypes

```
vsip_scalar_f vsip_vminmgval_f(
                      const vsip_vview_f *a,
                      vsip_scalar_vi *index);
vsip_scalar_f vsip_mminmgval_f(
                      const vsip_mview_f *a,
                      vsip_scalar_mi *index);
```

Arguments

`*a`	View of input vector/matrix
`*index`	Pointer to *index*, if *null* the index is not returned

Return Value

Returns the minimum absolute value of the elements. The index is returned using the *index* pointer, if non-*null*.

Restrictions

Errors

The arguments must conform to the following:

1. All view objects must be valid.

Notes/References

If the vector/matrix has more than one element with identical maximum absolute value values, the index of the first minimum absolute value is returned in the index.

Examples

For example of `vsip_sminmgval_p` see example of `vsip_smaxmgval_p`.

See Also

`vsip_smax_p`, `vsip_smaxmg_p`, `vsip_scmaxmgsq_p`, `vsip_scmaxmgsqval_p`, `vsip_smaxmgval_p`, `vsip_smaxval_p`, `vsip_smin_p`, `vsip_sminmg_p`, `vsip_scminmgsq_p`, `vsip_scminmgsqval_p`, and `vsip_sminval_p`

vsip_sminval_p Vector/Matrix Minimum Value

Returns the index and value of the minimum value of the elements of a vector/matrix. The index is returned by reference as one of the arguments.

Functionality

min ← a_0; index ← 0;
for j = 1, 2, …, N-1
if a_j < min then
min ← a_j ; index ← j;

min ← $a_{0,0}$; index ← (0, 0);
for i = 0, 1, …, M-1
for j = 0, 1, …, N-1
if $a_{i,j}$ < min then
min ← $a_{i,j}$; index ← (i, j);

Prototypes

```
vsip_scalar_f vsip_vminval_f(
                      const vsip_vview_f *a,
                      vsip_scalar_vi *index);
vsip_scalar_f vsip_mminval_f(
                      const vsip_mview_f *a,
                      vsip_scalar_mi *index);
```

Arguments

`*a`	View of input vector/matrix
`*index`	Pointer to index, if null the index is not returned

Return Value

Returns the maximum value of the elements. The index is returned using the *index* pointer, if non-*null*.

Restrictions

Errors

The arguments must conform to the following:

1. All view objects must be valid.

Notes/References

If the vector/matrix has more than one element with identical minimum values the index of the first minimum is returned in the index.

Examples

For example of `vsip_sminval_p` see example of `vsip_smaxval_p`.

See Also

`vsip_smax_p`, `vsip_smaxmg_p`, `vsip_scmaxmgsq_p`, `vsip_scmaxmgsqval_p`, `vsip_smaxmgval_p`, `vsip_smaxval_p`, `vsip_smin_p`, `vsip_sminmg_p`, `vsip_scminmgsq_p`, `vsip_scminmgsqval_p`, and `vsip_sminmgval_p`

Bitwise and Boolean Logical Operators

These functions correspond to by element application of bitwise logical operators to vectors/matrices of integers or logical operations to Boolean vectors/matrices.

`vsip_`***`Sand`***`_`*`p`	Vector/Matrix AND
`vsip_`***`Snot`***`_`*`p`	Vector/Matrix NOT
`vsip_`***`Sor`***`_`*`p`	Vector/Matrix OR
`vsip_`***`Sxor`***`_`*`p`	Vector/Matrix Exclusive OR

vsip_sand_*p* Vector/Matrix AND

Computes the "AND", by element, of two vectors/matrices.

Functionality

$r_j \leftarrow a_j \wedge b_j$ for $j = 0, 1, \ldots, N\text{-}1$

$r_{i,j} \leftarrow a_{i,j} \wedge b_{i,j}$ for $i = 0, 1, \ldots, M\text{-}1$; for $j = 0, 1, \ldots, N\text{-}1$

Prototypes

```
void vsip_vand_i(
                      const vsip_vview_i *a,
                      const vsip_vview_i *b,
                      const vsip_vview_i *r);
void vsip_mand_i(
                      const vsip_mview_i *a,
                      const vsip_mview_i *b,
                      const vsip_mview_i *r);
void vsip_vand_bl(
                      const vsip_vview_bl *a,
                      const vsip_vview_bl *b,
                      const vsip_vview_bl *r);
void vsip_mand_bl(
                      const vsip_mview_bl *a,
                      const vsip_mview_bl *b,
                      const vsip_mview_bl *r);
```

Arguments

`*a`	View of input vector/matrix
`*b`	View of input vector/matrix
`*r`	View of output vector/matrix

Return Value

None.

Restrictions

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.
3. The input and output views must be identical views of the same block (in-place), or must not overlap.

Notes/References

For integers, the "AND" is bitwise, for booleans, it is logical.

Examples

```
/* example of boolean AND */
#include<stdio.h>
#include "vsip.h"
#define L  4

int main()
{
    int i;
    vsip_vview_bl*  blA;
    vsip_vview_bl*  blB;
    vsip_vview_bl*  andBl;
    vsip_scalar_bl  vsip_false = 0;
    vsip_scalar_bl  vsip_true  = !vsip_false;

    vsip_init((void *)0);
    blA   = vsip_vcreate_bl(L, VSIP_MEM_NONE);
    blB   = vsip_vcreate_bl(L, VSIP_MEM_NONE);
    andBl = vsip_vcreate_bl(L, VSIP_MEM_NONE);

    /* Make up some data */
    vsip_vput_bl(blA,0,vsip_false);
    vsip_vput_bl(blB,0,vsip_false);
    vsip_vput_bl(blA,1,vsip_false);
    vsip_vput_bl(blB,1,vsip_true);
    vsip_vput_bl(blA,2,vsip_true);
    vsip_vput_bl(blB,2,vsip_false);
    vsip_vput_bl(blA,3,vsip_true);
    vsip_vput_bl(blB,3,vsip_true);

    /* do a boolean AND of A with B */
    vsip_vand_bl(blA,blB,andBl);

    /* print the results */
    printf("  A    B   =>  A and B\n");
    for(i = 0; i<L; i++){
       printf("%5s %5s     %5s \n",
             vsip_vget_bl(blA,i)   ? "True ":"False",
             vsip_vget_bl(blB,i)   ? "True ":"False",
             vsip_vget_bl(andBl,i) ? "True ":"False");
    }
    /* recover allocated memory */
    vsip_blockdestroy_bl(vsip_vdestroy_bl(blA));
    vsip_blockdestroy_bl(vsip_vdestroy_bl(blB));
    vsip_blockdestroy_bl(vsip_vdestroy_bl(andBl));
    vsip_finalize((void *)0);
    return 0;
}
/* output */
/*  A    B   =>  A and B
  False False     False
  False True      False
  True  False     False
  True  True      True   */
```

See Also

`vsip_`***s***`not_`***p***, `vsip_`***s***`or_`***p***, and `vsip_`***s***`xor_`***p***

vsip_snot_*p* Vector/Matrix NOT

Computes the NOT (one's complement), by element, of a vector/matrix.

Functionality

$r_j \leftarrow \neg a_j$ for $j = 0, 1, \ldots, N\text{-}1$

$r_{i,j} \leftarrow \neg a_{i,j}$ for $i = 0, 1, \ldots, M\text{-}1$; for $j = 0, 1, \ldots, N\text{-}1$

Prototypes

```
void vsip_vnot_i(
                      const vsip_vview_i *a,
                      const vsip_vview_i *r);
void vsip_mnot_i(
                      const vsip_mview_i *a,
                      const vsip_mview_i *r);
void vsip_vnot_bl(
                      const vsip_vview_bl *a,
                      const vsip_vview_bl *r);
void vsip_mnot_bl(
                      const vsip_mview_bl *a,
                      const vsip_mview_bl *r);
```

Arguments

`*a`	View of input vector/matrix
`*r`	View of input vector/matrix

Return Value

None.

Restrictions

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.
3. The input and output views must be identical views of the same block (in-place), or must not overlap.

Notes/References

For integers, the “NOT” is bitwise, for booleans, it is logical.

Examples

```
/* example of boolean NOT */
#include<stdio.h>
#include "vsip.h"

#define L  4

int main()
{
    int i;
    vsip_vview_bl* blA;
    vsip_vview_bl* blB;
    vsip_vview_bl* xorBl;
```

```
    vsip_vview_bl* notBl;
    vsip_scalar_bl vsip_false = 0;
    vsip_scalar_bl vsip_true  = !vsip_false;

    vsip_init((void *)0);
    blA   = vsip_vcreate_bl(L, VSIP_MEM_NONE);
    blB   = vsip_vcreate_bl(L, VSIP_MEM_NONE);
    xorBl = vsip_vcreate_bl(L, VSIP_MEM_NONE);
    notBl = vsip_vcreate_bl(L, VSIP_MEM_NONE);

    /* Make up some data */
    vsip_vput_bl(blA,0,vsip_false);
    vsip_vput_bl(blB,0,vsip_false);
    vsip_vput_bl(blA,1,vsip_false);
    vsip_vput_bl(blB,1,vsip_true);
    vsip_vput_bl(blA,2,vsip_true);
    vsip_vput_bl(blB,2,vsip_false);
    vsip_vput_bl(blA,3,vsip_true);
    vsip_vput_bl(blB,3,vsip_true);

    /* do a boolean XOR of A with B */
    vsip_vxor_bl(blA,blB,xorBl);

    /* do a boolean not of xorBl);*/
    vsip_vnot_bl(xorBl,notBl);

   /* print the results */
    printf("  A    B   =>  A xor B  => not(A xor B)\n");
    for(i = 0; i<L; i++){
       printf("%5s %5s     %5s       %5s\n",
             vsip_vget_bl(blA,i)   ? "True ":"False",
             vsip_vget_bl(blB,i)   ? "True ":"False",
             vsip_vget_bl(xorBl,i) ? "True ":"False",
             vsip_vget_bl(notBl,i) ? "True ":"False");
    }
    /* recover allocated memory */
    vsip_blockdestroy_bl(vsip_vdestroy_bl(blA));
    vsip_blockdestroy_bl(vsip_vdestroy_bl(blB));
    vsip_blockdestroy_bl(vsip_vdestroy_bl(xorBl));
    vsip_finalize((void *)0);
    return 0;
}   /* output */
/*  A    B   =>  A xor B  => not(A xor B)
  False False     False       True
  False True      True        False
  True  False     True        False
  True  True      False       True  */
```

See Also

vsip_*s*and_*p*, vsip_*s*or_*p*, and vsip_*s*xor_*p*

vsip_sor_*p* Vector/Matrix OR

Computes the "OR", by element, of two vectors/matrices.

Functionality

$r_j \leftarrow a_j \vee b_j$ for $j = 0, 1, \ldots, N\text{-}1$

$r_{i,j} \leftarrow a_{i,j} \vee b_{i,j}$ for $i = 0, 1, \ldots, M\text{-}1$; for $j = 0, 1, \ldots, N\text{-}1$

Prototypes

```
void vsip_vor_i(
                      const vsip_vview_i *a,
                      const vsip_vview_i *b,
                      const vsip_vview_i *r);
void vsip_mor_i(
                      const vsip_mview_i *a,
                      const vsip_mview_i *b,
                      const vsip_mview_i *r);
void vsip_vor_bl(
                      const vsip_vview_bl *a,
                      const vsip_vview_bl *b,
                      const vsip_vview_bl *r);
void vsip_mor_bl(
                      const vsip_mview_bl *a,
                      const vsip_mview_bl *b,
                      const vsip_mview_bl *r);
```

Arguments

`*a`	View of input vector/matrix
`*b`	View of input vector/matrix
`*r`	View of output vector/matrix

Return Value

None.

Restrictions

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.
3. The input and output views must be identical views of the same block (in-place), or must not overlap.

Notes/References

For integers, the "OR" is bitwise, for booleans, it is logical.

Examples

```
/* example of boolean OR */
#include<stdio.h>
#include "vsip.h"

#define L  4
```

```
int main()
{
    int i;
    vsip_vview_bl* blA;
    vsip_vview_bl* blB;
    vsip_vview_bl* orBl;
    vsip_scalar_bl vsip_false = 0;
    vsip_scalar_bl vsip_true  = !vsip_false;

    vsip_init((void *)0);
    blA  = vsip_vcreate_bl(L, VSIP_MEM_NONE);
    blB  = vsip_vcreate_bl(L, VSIP_MEM_NONE);
    orBl = vsip_vcreate_bl(L, VSIP_MEM_NONE);

    /* Make up some data */
    vsip_vput_bl(blA,0,vsip_false);
    vsip_vput_bl(blB,0,vsip_false);
    vsip_vput_bl(blA,1,vsip_false);
    vsip_vput_bl(blB,1,vsip_true);
    vsip_vput_bl(blA,2,vsip_true);
    vsip_vput_bl(blB,2,vsip_false);
    vsip_vput_bl(blA,3,vsip_true);
    vsip_vput_bl(blB,3,vsip_true);

    /* do a boolean OR of A with B */
    vsip_vor_bl(blA,blB,orBl);

    /* print the results */
    printf("  A    B   =>  A or B\n");
    for(i = 0; i<L; i++){
       printf("%5s %5s     %5s \n",
             vsip_vget_bl(blA,i)   ? "True ":"False",
             vsip_vget_bl(blB,i)   ? "True ":"False",
             vsip_vget_bl(orBl,i) ? "True ":"False");
    }    /* recover allocated memory */
    vsip_blockdestroy_bl(vsip_vdestroy_bl(blA));
    vsip_blockdestroy_bl(vsip_vdestroy_bl(blB));
    vsip_blockdestroy_bl(vsip_vdestroy_bl(orBl));
    vsip_finalize((void *)0);
    return 0;
}
/* output */
/*  A    B   =>  A or B
  False False     False
  False True      True
  True  False     True
  True  True      True  */
```

See Also

`vsip_`***s***`and_`***p***, `vsip_`***s***`not_`***p***, and `vsip_`***s***`xor_`***p***

vsip_*sxor*_*p* Vector/Matrix Exclusive OR

Computes the "XOR", by element, of two vectors/matrices.

Functionality

$r_j \leftarrow a_j \oplus b_j$ for $j = 0, 1, \ldots, N\text{-}1$

$r_{i,j} \leftarrow a_{i,j} \oplus b_{i,j}$ for $i = 0, 1, \ldots, M\text{-}1$; for $j = 0, 1, \ldots, N\text{-}1$

Prototypes

```
void vsip_vxor_i(
                      const vsip_vview_i *a,
                      const vsip_vview_i *b,
                      const vsip_vview_i *r);
void vsip_mxor_i(
                      const vsip_mview_i *a,
                      const vsip_mview_i *b,
                      const vsip_mview_i *r);
void vsip_vxor_bl(
                      const vsip_vview_bl *a,
                      const vsip_vview_bl *b,
                      const vsip_vview_bl *r);
void vsip_mxor_bl(
                      const vsip_mview_bl *a,
                      const vsip_mview_bl *b,
                      const vsip_mview_bl *r);
```

Arguments

`*a`	View of input vector/matrix
`*b`	View of input vector/matrix
`*r`	View of output vector/matrix

Return Value

None.

Restrictions

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.
3. The input and output views must be identical views of the same block (in-place), or must not overlap.

Notes/References

For integers, the "XOR" is bitwise, for booleans, it is logical.

Examples

```
/* example of boolean XOR */
#include<stdio.h>
#include "vsip.h"

#define L  4

int main()
{    int i;
    /* Make up some Boolean data space */
    vsip_vview_bl* blA;
    vsip_vview_bl* blB;
    vsip_vview_bl* xorBl;
    vsip_scalar_bl vsip_false = 0;
    vsip_scalar_bl vsip_true  = !vsip_false;

    vsip_init((void *)0);
    blA   = vsip_vcreate_bl(L, VSIP_MEM_NONE);
    blB   = vsip_vcreate_bl(L, VSIP_MEM_NONE);
    xorBl = vsip_vcreate_bl(L, VSIP_MEM_NONE);

    /* Make up some data */
    vsip_vput_bl(blA,0,vsip_false);
    vsip_vput_bl(blB,0,vsip_false);
    vsip_vput_bl(blA,1,vsip_false);
    vsip_vput_bl(blB,1,vsip_true);
    vsip_vput_bl(blA,2,vsip_true);
    vsip_vput_bl(blB,2,vsip_false);
    vsip_vput_bl(blA,3,vsip_true);
    vsip_vput_bl(blB,3,vsip_true);

    /* do a boolean XOR of A with B */
    vsip_vxor_bl(blA,blB,xorBl);

    /* print the results */
    printf("  A    B   =>  A xor B\n");
    for(i = 0; i<L; i++){
       printf("%5s %5s     %5s \n",
             vsip_vget_bl(blA,i)   ? "True ":"False",
             vsip_vget_bl(blB,i)   ? "True ":"False",
             vsip_vget_bl(xorBl,i) ? "True ":"False");
    }
    /* recover allocated memory */
    vsip_blockdestroy_bl(vsip_vdestroy_bl(blA));
    vsip_blockdestroy_bl(vsip_vdestroy_bl(blB));
    vsip_blockdestroy_bl(vsip_vdestroy_bl(xorBl));
    vsip_finalize((void *)0);
    return 0;
}
/* output */
/*  A    B   =>  A xor B
  False False     False
  False True      True
  True  False     True
  True  True      False  */
```

See Also

`vsip_`*s*`and_`*p*, `vsip_`*s*`not_`*p*, and `vsip_`*s*`or_`*p*

Element Generation and Copy

The following functions generate elements to fill a vector/matrix view based upon scalar parameters. These include random numbers, scalar fills, and ramps.

vsip_*ds*copy_*p*_*p*	Vector/Matrix/Tensor Copy
vsip_*ds*copyto_user_*p*	Vector/Matrix Copy To User Memory
vsip_*ds*copyfrom_user_*p*	Vector/Matrix Copy From User Memory
vsip_*ds*fill_*p*	Vector/Matrix Fill
vsip_vramp_*p*	Vector Ramp

vsip_*ds*copy_*p*_*p* Vector/Matrix/Tensor Copy

Copy the source vector/matrix/tensor to the destination vector/matrix/tensor performing any necessary type conversion of the standard ANSI C scalar types.

Functionality

Copies the source data of the source vector/matrix/tensor view object to the destination data of the destination vector/matrix/tensor view object performing any necessary type conversion of the standard ANSI C scalar types.

Prototypes

```
void vsip_vcopy_s_t(
                      const vsip_vview_s *x,
                      const vsip_vview_t *y);
void vsip_vcopy_vi_vi(
                      const vsip_vview_vi *x,
                      const vsip_vview_vi *y);
void vsip_vcopy_vi_i(
                      const vsip_vview_vi *x,
                      const vsip_vview_i *y);
void vsip_vcopy_i_vi (
                      const vsip_vview_i *x,
                      const vsip_vview_vi *y);
void vsip_vcopy_mi_mi(
                      const vsip_vview_mi *x,
                      const vsip_vview_mi *y);
void vsip_vcopy_ti_ti(
                      const vsip_vview_ti *x,
                      const vsip_vview_ti *y);
void vsip_vcopy_bl_t(
                      const vsip_vview_bl *x,
                      const vsip_vview_t *y);
void vsip_vcopy_s_bl (
                      const vsip_vview_s *x,
                      const vsip_vview_bl *y);
void vsip_cvcopy_s_t(
                      const vsip_cvview_s *x,
                      const vsip_cvview_t *y);
void vsip_mcopy_s_t(
                      const vsip_mview_s *X,
                      const vsip_mview_t *Y);
void vsip_mcopy_bl_t(
                      const vsip_mview_bl *X,
                      const vsip_mview_t *Y);
void vsip_mcopy_s_bl (
                      const vsip_mview_s *X,
                      const vsip_mview_bl *Y);
```

```
void vsip_cmcopy_s_t(
                      const vsip_cmview_s *X,
                      const vsip_cmview_t *Y);
void vsip_tcopy_s_t(
                      const vsip_tview_s *X,
                      const vsip_tview_t *Y);
void vsip_tcopy_bl_t(
                      const vsip_tview_bl *X,
                      const vsip_tview_t *Y);
void vsip_tcopy_s_bl (
                      const vsip_tview_s *X,
                      const vsip_tview_bl *Y);
void vsip_ctcopy_s_t(
                      const vsip_ctview_s *X,
                      const vsip_ctview_t *Y);
```

Where _s, and _t can be any combination of:

	_s, _t	ANSI C Type
_i	_c	signed char
	_si	short int
	_i	int
	_li	long int
	_ll	long long int (non-ANSI)
	_uc	unsigned char
	_us	unsigned short int
	_ui	unsigned int
	_ul	unsigned long int
	_ull	unsigned long long int (non-ANSI)
_f	_f	float
	_d	double
	_ld	long double

	VSIPL Type
_vi	vsip_scalar_vi or vsip_index
_mi	vsip_scalar_mi
_ti	vsip_scalar_ti
_bl	vsip_scalar_bl or vsip_bool

Arguments

*x Source vector/matrix/tensor view object.

*y Destination vector/matrix/tensor view object.

Return Value

None.

Restrictions

If the source and destination overlap, the result is undefined.

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.
3. The input and output views must be identical views of the same block (in-place), if and only if the source and destination data types are of the same size; otherwise they must not overlap.

Notes/References

Booleans as a source map *false* and *true* into 0 and 1 (or 0.0 and 1.0) respectively. Booleans as a destination map 0 (or 0.0) to *false* and everything else to *true*.

There are many possible different prototypes of this function. Most implementations will only support a small subset. Profiles may define required copy functionality. An example of the *core profile* required copies is below.

Core Profile Required Copy functions

Vector	*_f*	*_i*	_bl	_vi	_mi	_ti
_f	✓	✓	✓			
_i	✓	✓		✓		
_bl	✓		✓			
_vi		✓		✓		
_mi					✓	
_ti						

Matrix	*_f*	*_i*	_bl
_f	✓		
_i			
_bl			

The "Core Profile" requires 11 `vsip_vcopy` + 1 `vsip_mcopy` + 1 `vsip_cvcopy` + 1 `vsip_cmcopy`. There must be one base float *_f* type and one base integer *_i* type in a profile. (Typically the base types are `float` and `int` or `double` and `int`.) The base types must be fully supported in terms of the indicated table.

If an implementation supports more than one float or integer type, it must support copy functions from/to that type to/from the base type as indicated by the table. For example, if the base types are `float` and `int`, and the library also supports `long int` and `unsigned char`, then the following additional copy routines are required to support the copy to/from the base `int` type:

`vsip_vcopy_i_li` `vsip_vcopy_li_i` `vsip_vcopy_i_uc` `vsip_vcopy_uc_i`

Examples

Prototype Examples

Convert unsigned char to signed int.

```
void vsip_vcopy_uc_i(const vsip_vview_uc *x, const vsip_vview_i *y);
```

Convert complex user-specified unsigned 12 bit integers to complex single precision floating point.

```
void vsip_cvcopy_i12_f(const vsip_cvview_i12 *x, const vsip_cvview_f *y);
```

See Also

vsip_*ds*copyto_user_*p* Vector/Matrix Copy To User Memory

Copy data referenced by a VSIPL view to user allocated memory.

Functionality

The functionality is demonstrated using pseudo C code and well-defined VSIPL functionality. The method described indicates functionality and not the actual method used by an implementation.

VSIPL considers *user* data arrays to always have unit stride. Copy to user memory implies a major direction for matrices. The major argument in the prototypes indicates the major and minor stride in the user memory, and the direction of copy from the view. User memory has a unit stride in the major direction, and (for matrices) a stride the length of the major dimension in the minor direction.

In the notation below the view object we copy from is `a`, and the memory we copy into is `R` and possibly `I`. The memory pointer `I` is used if the output data are split complex.

The value N is the vector length (`vsip_dvgetlength_p(a)`) for vectors, and for matrices N is the row length (`vsip_dmgetrowlength_p(a)`). The value M is the column length for matrices (`vsip_dmgetcollength_p(a)`). The vector index is `k` and the matrix index is `(j,k)`. Using the above notation we have the following functionality for the copy to user memory.

For real vectors:

```
for(k=0; k<N; k++){
   *(R+k) = vsip_vget_p(a,k);
}
```

For complex vectors:

```
/* interleaved complex */
for(k=0; k<N; k++){
   *(R+2*k) = vsip_real_p(vsip_cvget_p(a,k));
   *(R+1+2*k) = vsip_imag_p(vsip_cvget_p(a,k));
}

/* split complex */
for(k=0; k<N; k++){
   *(R+k) = vsip_real_p(vsip_cvget_p(a,k));
   *(I+k) = vsip_imag_p(vsip_cvget_p(a,k));
}
```

For real matrices copied as column major:

```
for(k=0; k<N; k++)
   for(j=0; j<M; j++){
      *(R+j+k*M) = vsip_mget_p(a,j,k);
}
```

For real matrices copied as row major:

```
for(j=0; j<M; j++)
   for(k=0; k<N; k++){
      *(R+k+j*N) = vsip_mget_p(a,j,k);
}
```

For complex matrices copied as column major:

```
/* interleaved complex */
for(k=0; k<N; k++)
```

```
   for(j=0; j<M; j++){
      *(R+2*(j+k*M)) = vsip_real_p(vsip_cmget_p(a,j,k));
      *(R+1+2*(j+k*M)) = vsip_image_p(vsip_cmget_p(a,j,k));
}

/* split complex */for(k=0; k<N; k++)
    for(j=0; j<M; j++){
      *(R+j+k*M) = vsip_real_p(vsip_cmget_p(a,j,k));
      *(I+j+k*M) = vsip_image_p(vsip_cmget_p(a,j,k))
}
```

For complex matrices copied as row major:

```
/* interleaved complex */
for(j=0; j<M; j++)
   for(k=0; k<N; k++){
      *(R+2*(k+j*N)) = vsip_real_p(vsip_cmget_p(a,j,k));
      *(R+1+2*(k+j*N)) = vsip_image_p(vsip_cmget_p(a,j,k));
}

/* split complex */
for(j=0; j<M; j++)
   for(k=0; k<N; k++){
      *(R+k+j*N) = vsip_real_p(vsip_cmget_p(a,j,k));
      *(I+k+j*N) = vsip_image_p(vsip_cmget_p(a,j,k));
}
```

Prototypes

```
void vsip_vcopyto_user_p(
                      const vsip_vview_p *a,
                      vsip_scalar_p* const R);
void vsip_mcopyto_user_p(
                      const vsip_mview_p *a,
                      vsip_major major,
                      vsip_scalar_p* const R);
void vsip_cvcopyto_user_p(
                      const vsip_cvview_p *a,
                      vsip_scalar_p* const R
                      vsip_scalar_p* const I);
void vsip_cmcopyto_user_p(
                      const vsip_cmview_p *a,
                      vsip_major major,
                      vsip_scalar_p* const R,
                      vsip_scalar_p* const I);
```

Arguments

`*a`	View of input vector/matrix
`major`	The direction of copy, either by rows or by columns, for matrix copies.

`*R` Pointer to beginning of output memory. For real views, or complex views where `I` is not a *null* pointer then `R` should point to allocated memory of size [`ELEMENTS*sizeof(vsip_scalar_p)`]. For complex if `I` is `NULL` then `R` should point to user memory of size [`2*ELEMENTS*sizeof(vsip_scalar_p)`] and the output of the copy will be interleaved complex.

`*I` Pointer to output of imaginary part of complex data or NULL. If I is not a null pointer then it should point to allocated memory of size[`ELEMENTS*sizeof(vsip_scalar_p)`] and the output of the copy will be split complex.

Return Value

None

Restrictions

The arguments must conform to the following:

1. All view objects must be valid.

Errors

Notes/References

It is the responsibility of the user to ensure that sufficient memory is available to receive the data copied from the input view.

The major and minor stride of the view object is not relevant to the functionality. For this function the major argument indicates the direction of copy from matrices and the major and minor stride direction of user memory.

Examples

See Also

`vsip_dscopyfrom_user_p`.

vsip_*ds*copyfrom_user_*p* Vector/Matrix Copy From User Memory

Copy data from user allocated memory to a VSIPL view.

Functionality

The functionality is demonstrated using pseudo C code and well-defined VSIPL functionality. The method described indicates functionality and not the actual method used by an implementation.

VSIPL considers user data arrays to always have unit stride. Copy from user memory implies a major direction for matrices. The major argument indicates the major and minor stride in the user memory, and the direction of copy into the view. User memory has a unit stride in the major direction, and (for matrices) a stride the length of the major dimension in the minor direction.

In the notation below the view object we copy to is `a`, and the memory pointer we copy from is `R` and possibly `I`. The memory pointer `I` is used if the input data are split complex.

The value `N` is the vector length (`vsip_dvgetlength_p(a)`) for vectors, and for matrices `N` is the row length (`vsip_dmgetrowlength_p(a)`). The value `M` is the column length for matrices (`vsip_dmgetcollength_p(a)`). The vector index is `k` and the matrix index is (j,k).

Using the above notation we have the following functionality for the copy to a VSIPL view.

For real vectors:

```
for(k=0; k<N; k++){
   vsip_scalar_p re = *(R+k);
   vsip_vput_p(a, k, re );
}
```

For complex vectors:

```
/* interleaved complex */
for(k=0; k<N; k++){
   vsip_scalar_p re = *(R+2*k), im = *(R+2*k+1);
   vsip_cvput_p(a,k,vsip_cmplx_p(re,im));
}

/* split complex */
for(k=0; k<N; k++){
   vsip_scalar_p re = *(R+k), im = *(I+K);
   vsip_cvput_p(a,k,vsip_cmplx_p(re,im));
}
```

For real matrices copied as column major:

```
for(k=0; k<N; k++)
   for(j=0; j<M; j++){
      vsip_scalar_p re = *(R+j+k*M);
      vsip_mput_p(a,j,k,re);
}
```

For real matrices copied as row major:

```
for(j=0; j<M; j++)
   for(k=0; k<N; k++){
      vsip_scalar_p re = *(R+k+j*N);
      vsip_mput_p(a,j,k,re);
}
```

For complex matrices copied as column major:

```
/* interleaved complex */
for(k=0; k<N; k++)
   for(j=0; j<M; j++){
       vsip_scalar_p re = *(R+2*(j+k*M)),
                     im = *(R+1+2*(j+k*M));
       vsip_cmput_p(a,j,k,vsip_cmplx_p(re,im));
}

/* split complex */
for(k=0; k<N; k++)
   for(j=0; j<M; j++){
       vsip_scalar_p re = *(R+j+k*M), im = *(I+j+k*M);
       vsip_cmput_p(a,j,k,vsip_cmplx_p(re,im));
}
```

For complex matrices copied as row major:

```
/* interleaved complex */
for(j=0; j<M; j++)
   for(k=0; k<N; k++){
       vsip_scalar_p re = *(R+2*(k+j*N)),
                     im = *(R+1+2*(k+j*N));
       vsip_cmput_p(a,j,k,vsip_cmplx_p(re,im));
}

/* split complex */
for(k=0; k<N; k++)
   for(j=0; j<M; j++){
       vsip_scalar_p re = *(R+k+j*N), im = *(I+k+j*N);
       vsip_cmput_p(a,j,k,vsip_cmplx_p(re,im));
}
```

Prototypes

```
void vsip_vcopyfrom_user_p(
                     vsip_scalar_p* const R,
                     const vsip_vview_p *a);
void vsip_mcopyfrom_user_p(
                     vsip_scalar_p* const R,
                     vsip_major major,
                     const vsip_mview_p *a);
void vsip_cvcopyfrom_user_p(
                     vsip_scalar_p* const R,
                     vsip_scalar_p* const I,
                     const vsip_cvview_p *a);
void vsip_cmcopyfrom_user_p(
                     vsip_scalar_p* const R,
                     vsip_scalar_p* const I,
                     vsip_major major,
                     const vsip_cmview_p *a);
```

Arguments

Note that `ELEMENTS` is the number of elements of data referenced by the view. For vectors this is the vector length and for matrices this is the row length times the column length.

`*R`	Pointer to input memory holding data. For real views, or complex views where `I` is not a null pointer then `R` should point to allocated memory of size [`ELEMENTS*sizeof(vsip_scalar_`*p*`)`]. For complex if `I` is `NULL` then `R` should point to allocated memory of size [`2*ELEMENTS*sizeof(vsip_scalar_`*p*`)`] and the input data area assumed to be interleaved complex.
`*I`	Pointer to input memory holding the imaginary part of the complex data or `NULL`. If `I` is not a null pointer then it should point to allocated memory of size [`ELEMENTS*sizeof(vsip_scalar_`*p*`)`] and the input data are assumed to be split complex.
`major`	The direction of copy, either by rows or by columns, for matrix copies.
`*a`	View of output vector/matrix.

Return Value

None.

Restrictions

Errors

The arguments must conform to the following.

1. All view objects must be valid.

Notes/References

The major and minor stride of the view object is not relevant to the functionality. For this function the `major` argument indicates the direction of copy from matrices and the major and minor stride direction of user memory.

Examples

See Also

`vsip_`*ds*`copyfrom_user_`*p*.

vsip_*ds*fill_*p* Vector/Matrix Fill

Fill a vector/matrix with a constant value.

Functionality

$r_j \leftarrow \alpha$ for $j = 0, 1, \ldots, N\text{-}1$

$r_{i,j} \leftarrow \alpha$ for $i = 0, 1, \ldots, M\text{-}1$; for $j = 0, 1, \ldots, N\text{-}1$

Prototypes

```
void vsip_vfill_i(
                    vsip_scalar_i alpha,
                    const vsip_vview_i *r);
void vsip_vfill_f(
                    vsip_scalar_f alpha,
                    const vsip_vview_f *r);
void vsip_cvfill_f(
                    vsip_cscalar_f alpha,
                    const vsip_cvview_f *r);
void vsip_mfill_i(
                    vsip_scalar_i alpha,
                    const vsip_mview_i *r);
void vsip_mfill_f(
                    vsip_scalar_f alpha,
                    const vsip_mview_f *r);
void vsip_cmfill_f(
                    vsip_cscalar_f alpha,
                    const vsip_cmview_f *r);
```

Arguments

`alpha`	Scalar fill value
`*r`	View of output vector/matrix

Return Value

None.

Restrictions

Errors

The arguments must conform to the following:

1. All view objects must be valid.

Notes/References

Examples

There are numerous instances of `vsip_dsfill_p` in other examples. See, the example included with `vsip_dsexpoavg_p`.

See Also

`vsip_vramp_p`

vsip_vramp_*p* Vector Ramp

Computes a vector ramp by starting at an initial value and incrementing each successive element by the ramp step size.

Functionality

$r_k \leftarrow \alpha \quad + k \cdot \beta$ for $k = 0, 1, \ldots, N\text{-}1$

Where:

α is the initial value of the ramp

β is the ramp step size

Prototypes

```
void vsip_vramp_i(
                      vsip_scalar_i alpha,
                      vsip_scalar_i beta,
                      const vsip_vview_i *r);
void vsip_vramp_f(
                      vsip_scalar_f alpha
                      vsip_scalar_f beta,
                      const vsip_vview_f *r);
```

Arguments

`alpha`	Initial value of vector ramp
`beta`	Ramp increment (decrement if negative)
`*r`	View of output vector

Return Value

None.

Restrictions

Errors

The arguments must conform to the following:

1. All view objects must be valid.

Notes/References

Examples

There are numerous examples of ramp in other examples. See for instance the example included with `vsip_shypot_p`.

See Also

`vsip_dsfill_p`

Manipulation Operations

These functions operate by copying vectors/matrices, or parts of vectors/matrices, from one vector/matrix location to another. In addition, functions whose primary job is to convert to or from a complex data type from a real or polar data type are included here.

vsip_*s*cmplx_*p*	Vector/Matrix Complex
vsip_*ds*gather_*p*	Vector/Matrix Gather
vsip_*d*tgather_*p*	Tensor Gather
vsip_*s*imag_*p*	Vector/Matrix Imaginary
vsip_*s*polar_*p*	Vector/Matrix Polar
vsip_*s*real_*p*	Vector/Matrix Real
vsip_*s*rect_*p*	Vector/Matrix Rectangular
vsip_*ds*scatter_*p*	Vector/Matrix Scatter
vsip_*d*tscatter_*p*	Tensor Scatter
vsip_*ds*swap_*p*	Vector/Matrix Swap

vsip_scmplx_*p* Vector/Matrix Complex

Form a complex vector/matrix from two real vectors/matrices.

Functionality

$\mathrm{Re}(r_j) \leftarrow a_j$; $\mathrm{Im}(r_j) \leftarrow b_j$ for $j = 0, 1, \ldots, N\text{-}1$

$\mathrm{Re}(r_{i,j}) \leftarrow a_{i,j}$; $\mathrm{Im}(r_{i,j}) \leftarrow b_{i,j}$ for $i = 0, 1, \ldots, M\text{-}1$; for $j = 0, 1, \ldots, N\text{-}1$

Prototypes

```
void vsip_vcmplx_f(
                    const vsip_vview_f *a,
                    const vsip_vview_f *b,
                    const vsip_cvview_f *r);
void vsip_mcmplx_f(
                    const vsip_mview_f *a,
                    const vsip_mview_f *b,
                    const vsip_cmview_f *r);
```

Arguments

`*a`	View of input vector/matrix which contains the real part
`*b`	View of input vector/matrix which contains the imaginary part
`*r`	View of output complex vector/matrix

Return Value

None.

Restrictions

In-place operation for this function means the input vectors (one or both) are either a real view, or an imaginary view, of the output vector. No in-place operation is defined for an input vector which contains both real and imaginary components of the output vector, or which do not exactly overlap a real view or an imaginary view of the output vector.

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.
3. The input and output views must be identical views of the same block (in-place), or must not overlap.

Notes/References

Examples

For example of complex see example included with `vsip_cvjdot_p`.

See Also

`vsip_simag_p`, `vsip_spolar_p`, `vsip_sreal_p`, and `vsip_srect_p`

vsip_*ds*gather_*p* Vector/Matrix Gather

The gather operation selects elements of a source vector/matrix using indices supplied by an index vector. The selected elements are placed sequentially in an output vector so that the output vector and the index vector are indexed the same.

Functionality

$y_j \leftarrow x_{\text{index}_j}$ for $j = 0, 1, \ldots, N\text{-}1$

Where N is the index vector length.

Note that an index vector for a vector contains scalar elements suitable for indexing a vector. An index vector for a matrix contains elements consisting of pairs of scalars (row index and column index) suitable for indexing a matrix. The output of a gather is always a vector.

Prototypes

```
void vsip_vgather_i (
                     const vsip_vview_i *x,
                     const vsip_vview_vi *index,
                     vsip_vview_i *y);
void vsip_mgather_i (
                     const vsip_mview_i *x,
                     const vsip_vview_mi *index,
                     vsip_vview_i *y);
void vsip_vgather_mi (
                     const vsip_vview_mi *x,
                     const vsip_vview_vi *index,
                     vsip_vview_mi *y);
void vsip_vgather_f (
                     const vsip_vview_f *x,
                     const vsip_vview_vi *index,
                     vsip_vview_f *y);
void vsip_mgather_f (
                     const vsip_mview_f *x,
                     const vsip_vview_mi *index,
                     vsip_vview_f *y);
void vsip_cvgather_f (
                     const vsip_cvview_f *x,
                     const vsip_vview_vi *index,
                     vsip_cvview_f *y);
void vsip_cmgather_f (
                     const vsip_cmview_f *x,
                     const vsip_vview_mi *index,
                     vsip_cvview_f *y);
```

Arguments

`*x`	View of input source vector/matrix
`*index`	View of input vector/matrix index vector
`*y`	View of output destination vector/matrix

Return Value

None.

Restrictions

The length of the destination vector must be (set to) the same size as the index vector.

Errors

The arguments must conform to the following:

1. The index and output vectors views must be the same length.
2. All view objects must be valid.
3. Index values in the index vector must be valid indexes into the source vector.

Notes/References

The destination vector must be the same size as the index vector. If these are not predetermined they should be checked and set at runtime.

Examples

```
/* example of gather */
#include<stdio.h>
#include "vsip.h"

#define L 10 /* A length */
#define PI 3.141592653589793

int main()
{
    vsip_vview_d*  dataA;
    vsip_vview_d*  dataB;
    vsip_vview_vi* Index;
    vsip_vview_bl* dataBl;
    int i;
    vsip_length N;

    vsip_init((void *)0);
    dataA  = vsip_vcreate_d(L, VSIP_MEM_NONE);
    dataB  = vsip_vcreate_d(L, VSIP_MEM_NONE);
    Index  = vsip_vcreate_vi(L, VSIP_MEM_NONE);
    dataBl = vsip_vcreate_bl(L, VSIP_MEM_NONE);

    /* Make up some data */
    vsip_vramp_d(0,2 * PI/(L-1),dataA);
    vsip_vcos_d(dataA,dataB);

    /* Find out where dataB is greater than zero */
    vsip_vfill_d(0,dataA);
    vsip_vlge_d(dataB,dataA,dataBl);

    /* Find the index where dataB is greater than zero */
    if((N = vsip_vindexbool(dataBl,Index))){
        /* make a vector of those points where dataB is greater than zero*/
        vsip_vgather_d(dataB,Index,vsip_vputlength_d(dataA,N));

        /*print out the results */
        printf("Index   Value\n");
        for(i=0; i<N; i++)
           printf("%li      %6.3f\n", vsip_vget_vi(Index,i),
```

```
                vsip_vget_d(dataA,i));
        } else {
            printf("Zero Length Index");
        }
         vsip_blockdestroy_d(vsip_vdestroy_d(dataA));
         vsip_blockdestroy_d(vsip_vdestroy_d(dataB));
         vsip_blockdestroy_vi(vsip_vdestroy_vi(Index));
         vsip_blockdestroy_bl(vsip_vdestroy_bl(dataBl));
         vsip_finalize((void *)0);
         return 0;
}
/* output */
/* Index   Value
   0       1.000
   1       0.766
   2       0.174
   7       0.174
   8       0.766
   9       1.000 */
```

See Also

The function `vsip_`***s***`indexbool` may be used to produce index vectors from boolean results.

The function `vsip_`***s***`scatter_`***p*** is an inverse function of gather.

vsip_*d*tgather_*p* Tensor Gather

Gather tensor elements into a vector.

Functionality

Selects values from tensor ***U*** to gather (place) into vector ***v***, using indices from vector ***t***, such that:

$v_i \leftarrow u_{t_i}$

Prototypes

```
void vsip_tgather_f (
                     const vsip_tview_f *U,
                     const vsip_vview_ti *t,
                     const vsip_vview_f *v);

void vsip_ctgather_f (
                     const vsip_ctview_f *U,
                     const vsip_vview_ti *t,
                     const vsip_cvview_f *v);

void vsip_tgather_i (
                     const vsip_tview_i *U,
                     const vsip_vview_ti *t,
                     const vsip_vview_i *v);

void vsip_ctgather_i (
                     const vsip_ctview_i *U,
                     const vsip_vview_ti *t,
                     const vsip_cvview_i *v);

void vsip_tgather_bl (
                     const vsip_tview_bl *U,
                     const vsip_vview_ti *t,
                     const vsip_vview_bl *v);
```

Arguments

`*T`	Input – Tensor view of source
`*U`	Input - View of index vector
`*v`	Output – Vector view of destination

Return Value

The length of the destination vector must be (set to) the same size as the index vector.

Restrictions

Errors

The following cause a VSIPL runtime error in *development mode;* in *production* mode the results will be implementation dependent.

1. The index input vector and the output vector must have identical lengths.
2. Arguments passed to the function must be defined and must not be *null*.
3. Index values in the index vector must be valid indexes into the source tensor.

Notes/References

Examples

See Also

vsip_simag_*p* Vector/Matrix Imaginary

Extract the imaginary part of a complex vector/matrix.

Functionality

$r_j \leftarrow \mathrm{Im}(a_j)$ for $j = 0, 1, \ldots, N\text{-}1$

$r_{i,j} \leftarrow \mathrm{Im}(a_{i,j})$ for $i = 0, 1, \ldots, M\text{-}1$; for $j = 0, 1, \ldots, N\text{-}1$

Prototypes

```
void vsip_vimag_f(
                      const vsip_cvview_f *a,
                      const vsip_vview_f *r);
void vsip_mimag_f(
                      const vsip_cmview_f *a,
                      const vsip_mview_f *r);
```

Arguments

`*a`	View of complex input vector/matrix
`*r`	View of real output vector(matrix)

Return Value

None.

Restrictions

If done in-place the output is placed in a real or imaginary view of the input. No in-place functionality is defined which places the output in a view which encompasses both real and imaginary space in the input vector. The output vector for in-place must exactly overlap the data space of the real view or the imaginary view of the input, and must not be disjoint.

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.
3. The input and output views must be identical views of the same block (in-place), or must not overlap.

Notes/References

Frequently it would be preferable to use the support function `vsip_imagview_p` instead of `vsip_imag_p`. The difference is whether a copy of the imaginary portion of the vector is made, or just a view of the imaginary portion is returned.

Examples

```
/* Example of imag and real */
#include<stdio.h>
#include "vsip.h"

#define PI 3.1415926535
#define L 7                    /* length */

int main()
{
    int i;
    vsip_cvview_d* dataEuler;
```

```
    vsip_vview_d  *data, *real, *imag;

    vsip_init((void *)0);
    dataEuler = vsip_cvcreate_d(L, VSIP_MEM_NONE);
    data      = vsip_vcreate_d(L, VSIP_MEM_NONE);
    real      = vsip_vcreate_d(L, VSIP_MEM_NONE);
    imag      = vsip_vcreate_d(L, VSIP_MEM_NONE);

    /* Make up some data */
    /* compute a ramp from zero to 2pi */
    vsip_vramp_d(0.0, (2.0 * PI / (double) (L - 1)), data);
    vsip_veuler_d(data, dataEuler);

    /* find the real and imaginary parts */
    vsip_vreal_d(dataEuler, real);
    vsip_vimag_d(dataEuler, imag);

    /* print the results */
    printf("      Complex          Real     Imaginary\n");
    for(i=0; i<L; i++){
        printf("(%7.4f, %7.4f) => %7.4f;  %7.4f\n",
                           vsip_real_d(vsip_cvget_d(dataEuler,i)),
                           vsip_imag_d(vsip_cvget_d(dataEuler,i)),
                           vsip_vget_d(real,i),
                           vsip_vget_d(imag,i));
    }

    /*destroy the vector views and any associated blocks */
    vsip_blockdestroy_d(vsip_vdestroy_d(data));
    vsip_blockdestroy_d(vsip_vdestroy_d(real));
    vsip_blockdestroy_d(vsip_vdestroy_d(imag));
    vsip_cblockdestroy_d(vsip_cvdestroy_d(dataEuler));
    vsip_finalize((void *)0);
    return 0;
}
/* output */
/*      Complex          Real     Imaginary
   ( 1.0000,  0.0000) =>  1.0000;   0.0000
   ( 0.5000,  0.8660) =>  0.5000;   0.8660
   (-0.5000,  0.8660) => -0.5000;   0.8660
   (-1.0000,  0.0000) => -1.0000;   0.0000
   (-0.5000, -0.8660) => -0.5000;  -0.8660
   ( 0.5000, -0.8660) =>  0.5000;  -0.8660
   ( 1.0000, -0.0000) =>  1.0000;  -0.0000  */
```

See Also

vsip_***s***imagview_***f***, vsip_***s***cmplx_***p***, vsip_***s***imag_***p***, vsip_***s***polar_***p***, vsip_***s***real_***p***, and vsip_***s***rect_***p***

vsip_spolar_*p* Vector/Matrix Polar

Convert a complex vector/matrix from rectangular to polar form. The polar data consists of a real vector/matrix containing the radius and a corresponding real vector/matrix containing the argument (angle) of the complex input data.

Functionality

$r_j \leftarrow |a_j|$; $\phi_j \leftarrow \arg(a_j)$ for $j = 0, 1, \ldots, N\text{-}1$

$r_{i,j} \leftarrow |a_{i,j}|$; $\phi_{i,j} \leftarrow \arg(a_{i,j})$ for $i = 0, 1, \ldots, M\text{-}1$; for $j = 0, 1, \ldots, N\text{-}1$

Prototypes

```
void vsip_vpolar_f(
                    const vsip_cvview_f *a,
                    const vsip_vview_f *r,
                    const vsip_vview_f *phi);
void vsip_mpolar_f(
                    const vsip_cmview_f *a,
                    const vsip_mview_f *r,
                    const vsip_mview_f *phi);
```

Arguments

`*a`	View of input rectangular form vector/matrix
`*r`	View of output radius (magnitude) vector/matrix
`*phi`	View of output angle (argument), ϕ, vector/matrix

Return Value

Restrictions

In-place operation for this function requires that the radius and argument output vectors be placed in a real or imaginary view of the input vector. No in-place functionality is defined where an output view contains both real and imaginary data space. The in-place real or imaginary view must exactly overlap the input data space and must not be disjoint.

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.
3. The input and output views must be identical views of the same block (in-place), or must not overlap.

Notes/References

For in-place there is no requirement on which view which output vector is placed in. So the radius vector could go in either the real or imaginary view, and the argument vector would go in the view not used by the radius vector.

In VSIPL, complex numbers are always in rectangular (Cartesian) format. The polar form is represented by two real vectors/matrices.

Examples

```
/* Example of polar */
#include<stdio.h>
#include "vsip.h"
```

```
#define PI 3.1415926535
#define L 7                    /* length */

int main()
{
    int i;
    vsip_cvview_d* dataEuler;
    vsip_vview_d  *data, *radius, *arg;

    vsip_init((void *)0);
    dataEuler = vsip_cvcreate_d(L, VSIP_MEM_NONE);
    data      = vsip_vcreate_d(L, VSIP_MEM_NONE);
    radius    = vsip_vcreate_d(L, VSIP_MEM_NONE);
    arg       = vsip_vcreate_d(L, VSIP_MEM_NONE);

    /* Make up some data */
    /* compute a ramp from zero to 2pi */
    vsip_vramp_d(0.0, (2.0 * PI / (double) (L - 1)), data);
    vsip_veuler_d(data,dataEuler);

    /* find the radius and argument */
    vsip_vpolar_d(dataEuler, radius, arg);

    /* print the results */
    printf("       rect            radius    argument\n");
    for(i=0; i<L; i++){
        printf("(%7.4f, %7.4f) => %7.4f;  %7.4f\n",
                           vsip_real_d(vsip_cvget_d(dataEuler,i)),
                           vsip_imag_d(vsip_cvget_d(dataEuler,i)),
                           vsip_vget_d(radius,i),
                           vsip_vget_d(arg,i));
    }
    /*destroy the vector views and any associated blocks */
    vsip_blockdestroy_d(vsip_vdestroy_d(data));
    vsip_blockdestroy_d(vsip_vdestroy_d(radius));
    vsip_blockdestroy_d(vsip_vdestroy_d(arg));
    vsip_cblockdestroy_d(vsip_cvdestroy_d(dataEuler));
    vsip_finalize((void *)0);
    return 0;
}
/* output */
/*        rect            radius    argument
   ( 1.0000,  0.0000) =>  1.0000;   0.0000
   ( 0.5000,  0.8660) =>  1.0000;   1.0472
   (-0.5000,  0.8660) =>  1.0000;   2.0944
   (-1.0000,  0.0000) =>  1.0000;   3.1416
   (-0.5000, -0.8660) =>  1.0000;  -2.0944
   ( 0.5000, -0.8660) =>  1.0000;  -1.0472
   ( 1.0000, -0.0000) =>  1.0000;  -0.0000 */
```

See Also

vsip_***s***cmplx_***p***, vsip_***s***imag_***p***, vsip_***s***real_***p***, and vsip_***s***rect_***p***

vsip_sreal_p Vector/Matrix Real

Extract the real part of a complex vector/matrix.

Functionality

$r_j \leftarrow \mathrm{Re}(a_j)$ for $j = 0, 1, \ldots, N\text{-}1$

$r_{i,j} \leftarrow \mathrm{Re}(a_{i,j})$ for $i = 0, 1, \ldots, M\text{-}1$; for $j = 0, 1, \ldots, N\text{-}1$

Prototypes

```
void vsip_vreal_f(
                    const vsip_cvview_f *a
                    const vsip_vview_f *r);
void vsip_mreal_f(
                    const vsip_cmview_f *a
                    const vsip_mview_f *r);
```

Arguments

`*a`	View of complex input vector/matrix
`*r`	View of real output vector/matrix

Return Value

None.

Restrictions

If done in-place the output is placed in a real or imaginary view of the input. No in-place functionality is defined which places the output in a view which encompasses both real and imaginary space in the input vector. The output vector for in-place must exactly overlap the data space of the real view or the imaginary view of the input, and must not be disjoint.

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.
3. The input and output views must be identical views of the same block (in-place), or must not overlap.

Notes/References

The support function `vsip_srealview_p` will often be preferable to `vsip_sreal_d`. The difference is whether a copy of the real portion of the vector is made, or just a view of the real portion is returned.

Examples

For example of `vsip_sreal_p` see example under `vsip_simag_p`

See Also

`vsip_srealview_p`, `vsip_scmplx_p`, `vsip_simag_p`, `vsip_spolar_p`, and `vsip_srect_p`

vsip_srect_*p* Vector/Matrix Rectangular

Convert a pair of real vectors/matrices from complex polar to complex rectangular form.

Functionality

$Y_k \leftarrow r_k \cdot (\cos \phi_k + j \cdot \sin \phi_k)$ for $k = 0, 1, \ldots, N\text{-}1$

$y_{k,l} \leftarrow r_{k,l} \cdot (\cos \phi_{k,l} + j \cdot \sin \phi_{k,l})$ for $k = 0, 1, \ldots, M\text{-}1$; for $l = 0, 1, \ldots, N\text{-}1$

Prototypes

```
void vsip_vrect_f(
                      const vsip_vview_f *r,
                      const vsip_vview_f  *phi,
                      const vsip_cvview_f *y);
void vsip_mrect_f(
                      const vsip_mview_f *r,
                      const vsip_mview_f  *phi,
                      const vsip_cmview_f *y);
```

Arguments

`*r`	View of input radius (magnitude) vector/matrix
`*phi`	View of input angle (argument), ϕ, vector/matrix
`*y`	View of output rectangular form vector/matrix

Return Value

None.

Restrictions

In-place operation for this function requires that the radius and argument input vectors be in a real or imaginary view of the output vector. No in-place functionality is defined where an input view contains both real and imaginary data space of the output view. For in-place the data in the views must exactly overlap and not be disjoint.

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.
3. The input and output views must be identical views of the same block (in-place), or must not overlap.

Notes/References

For in-place either the real or imaginary view of the output can hold the radius data and the other view holds the argument data.

In VSIPL, complex numbers are always in rectangular (Cartesian) format. The polar form is represented by two real vectors/matrices.

Examples

```
/* Example of rect */
#include<stdio.h>
#include "vsip.h"

#define PI 3.1415926535
#define L 7                    /* length */
```

```
int main()
{
    int i;
    vsip_cvview_d *dataEuler, *dataRect;
    vsip_vview_d  *data, *radius, *arg;

    vsip_init((void *)0);
    dataEuler = vsip_cvcreate_d(L, VSIP_MEM_NONE);
    dataRect  = vsip_cvcreate_d(L, VSIP_MEM_NONE);
    data      = vsip_vcreate_d(L, VSIP_MEM_NONE);
    radius    = vsip_vrealview_d(dataEuler);
    arg       = vsip_vimagview_d(dataEuler);

    /* Make up some data */
    /* compute a ramp from zero to 2pi */
    vsip_vramp_d(0.0, (2.0 * PI / (double) (L - 1)), data);
    vsip_veuler_d(data,dataEuler);

    /* find the complex assuming real view of Euler is Radius
       and the imaginary view is the Argument */
    vsip_vrect_d(radius, arg, dataRect);

    /* print the results */
    printf("       rect            radius    argument\n");
    for(i=0; i<L; i++){
        printf("(%7.4f, %7.4f) <= %7.4f; %7.4f\n",
                           vsip_real_d(vsip_cvget_d(dataRect,i)),
                           vsip_imag_d(vsip_cvget_d(dataRect,i)),
                          vsip_vget_d(radius,i),
                           vsip_vget_d(arg,i));
    }
    /*destroy the vector views and any associated blocks */
    vsip_blockdestroy_d(vsip_vdestroy_d(data));
    vsip_blockdestroy_d(vsip_vdestroy_d(radius));
    vsip_blockdestroy_d(vsip_vdestroy_d(arg));
    vsip_cblockdestroy_d(vsip_cvdestroy_d(dataEuler));
    vsip_cblockdestroy_d(vsip_cvdestroy_d(dataRect));
    vsip_finalize((void *)0);
    return 0;
}
/* output */
/*       rect            radius    argument
   ( 1.0000,  0.0000) <=  1.0000;  0.0000
   ( 0.3239,  0.3809) <=  0.5000;  0.8660
   (-0.3239, -0.3809) <= -0.5000;  0.8660
   (-1.0000, -0.0000) <= -1.0000;  0.0000
   (-0.3239,  0.3809) <= -0.5000; -0.8660
   ( 0.3239, -0.3809) <=  0.5000; -0.8660
   ( 1.0000, -0.0000) <=  1.0000; -0.0000 */
```

See Also

vsip_***s***cmplx_***p***, vsip_***s***imag_***p***, vsip_***s***polar_***p***, and vsip_***s***real_***p***

vsip_*ds*scatter_*p* Vector/Matrix Scatter

The scatter operation sequentially utilizes elements of a source vector and an index vector. The element of the vector index (matrix index) is used to select a storage location in the output vector/matrix to store the element from the source vector.

Functionality

$y_{\text{index}_j} \leftarrow x_j$ for $j = 0, 1, \ldots, N\text{-}1$

Where N is the index vector length

Note that an index vector for a vector contains scalar elements suitable for indexing a vector. An index vector for a matrix contains elements consisting of pairs of scalars (row index and column index) suitable for indexing a matrix. The input of a scatter is always a vector.

Prototypes

```
void vsip_vscatter_i (
                      const vsip_vview_i *x,
                      const vsip_vview_i *y,
                      const vsip_vview_vi *index);
void vsip_mscatter_i (
                      const vsip_vview_i *x,
                      const vsip_mview_i *y,
                      const vsip_vview_mi *index);
void vsip_vscatter_mi(
                      const vsip_vview_mi *x,
                      const vsip_vview_mi *y,
                      const vsip_vview_vi *index);
void vsip_vscatter_f (
                      const vsip_vview_f *x,
                      const vsip_vview_f *y,
                      const vsip_vview_vi *index);
void vsip_mscatter_f (
                      const vsip_vview_f *x,
                      const vsip_mview_f *y,
                      const vsip_vview_mi *index);
void vsip_cvscatter_f (
                      const vsip_vview_f *x,
                      const vsip_vview_f *y,
                      const vsip_vview_vi *index);
void vsip_cmscatter_f (
                      const vsip_cvview_f *x,
                      const vsip_cmview_f *y,
                      const vsip_vview_mi *index);
```

Arguments

`*x`	View of input source vector/matrix
`*y`	View of output destination vector/matrix
`*index`	View of input vector/matrix index vector

Return Value

None.

Restrictions

If the index vector contains duplicate entries, the value stored in the destination will be from the source vector, but which value is not defined.

There is no in-place functionality for this function.

Errors

The arguments must conform to the following:

1. The index and input vectors must have identical lengths.
2. All view objects must be valid.
3. Index values in the index vector must be valid indexes into the output.

Notes/References

The attributes of the destination vector/matrix are not modified. Values in the destination not indexed are not modified.

Examples

```
/* example of scatter */
/* Use gather and scatter to clip at zero a cosine */
#include<stdio.h>
#include "vsip.h"

#define L 10                    /* A length/
#define PI 3.141592653589793

int main()
{
    vsip_vview_d*   dataA;
    vsip_vview_d*   dataB;
    vsip_vview_vi*  Index;
    vsip_vview_bl*  dataBl;
    int i;
    vsip_length N;

    vsip_init((void *)0);
    dataA = vsip_vcreate_d(L, VSIP_MEM_NONE);
    dataB = vsip_vcreate_d(L, VSIP_MEM_NONE);
    Index = vsip_vcreate_vi(L, VSIP_MEM_NONE);
    dataBl= vsip_vcreate_bl(L, VSIP_MEM_NONE);

    /* make up some data */
    vsip_vramp_d(0,2 * PI/(L-1),dataA);
    vsip_vcos_d(dataA,dataB);

    /* find out where dataB is greater than zero */
    vsip_vfill_d(0,dataA);
    vsip_vlgt_d(dataB,dataA,dataBl);

    /* find the index where dataB is greater than zero */
    if((N = vsip_vindexbool(dataBl,Index))){
        /* make a vector of those points where dataB is greater than zero*/
        vsip_vgather_d(dataB,Index,vsip_vputlength_d(dataA,N));
        /*print out the results */
```

```
         printf("Index   Value\n");
         for(i=0; i<N; i++)
            printf("%li      %6.3f\n",
              vsip_vget_vi(Index,i),
              vsip_vget_d(dataA,i));
      }
      else{
         printf("Zero Length Index");
         exit(0);
      }
      vsip_vfill_d(0,dataB);
      vsip_vscatter_d(dataA,dataB,Index);
      for(i=0; i<L; i++)
         printf("%6.3f\n",vsip_vget_d(dataB,i));

      /*recover the data space*/
      vsip_blockdestroy_d(vsip_vdestroy_d(dataA));
      vsip_blockdestroy_d(vsip_vdestroy_d(dataB));
      vsip_blockdestroy_vi(vsip_vdestroy_vi(Index));
      vsip_blockdestroy_bl(vsip_vdestroy_bl(dataBl));
      vsip_finalize((void *)0);
      return 0;
}
/* output */
/* Index   Value
   0       1.000
   1       0.766
   2       0.174
   7       0.174
   8       0.766
   9       1.000
    1.000
    0.766
    0.174
    0.000
    0.000
    0.000
    0.000
    0.174
    0.766
    1.000 */
```

See Also

The function `vsip_`***s***`indexbool` may be used to produce index vectors from boolean results.

The function `vsip_`***ds***`gather_`***p*** is an inverse function of scatter only if the index vector contains no duplicate entries.

vsip_*d*tscatter_*p* Tensor Scatter

Scatter the elements of a vector into a tensor.

Functionality

Selects locations in tensor ***V*** to scatter (place) the values from vector ***u*** using tensor indices from vector ***t*** such that:

$v_{t_i} \leftarrow u_i$

Prototypes

```
void vsip_tscatter_f (
                      const vsip_vview_f *u,
                      const vsip_vview_ti *t,
                      const vsip_tview_f *V);
void vsip_ctscatter_f (
                      const vsip_cvview_f *u,
                      const vsip_cvview_ti *t,
                      const vsip_ctview_f *V);
void vsip_tscatter_i (
                      const vsip_vview_i *u,
                      const vsip_vview_ti *t,
                      const vsip_tview_i *V);
void vsip_ctscatter_i (
                      const vsip_cvview_i *u,
                      const vsip_cvview_ti *t,
                      const vsip_ctview_i *V);
void vsip_tscatter_bl (
                      const vsip_vview_bl *u,
                      const vsip_vview_ti *t,
                      const vsip_tview_bl *V);
```

Arguments

`*u`	Input – Tensor view of source
`*t`	Input - View of index vector
`*V`	Output – Tensor view of destination

Return Value

None.

Restrictions

If the index vector contains duplicate entries the value stored in the destination will be from the source vector but which value is not defined.

There is no in-place functionality for this function.

Errors

The following cause a VSIPL runtime error in *development mode;* in *production* mode the results will be implementation dependent.

1. The index vector and input tensor must have identical lengths.
2. Arguments passed to the function must be defined and must not be *null*.
3. Index values in the index vector must be valid indexes into the output tensor view.

Notes/References

Examples

See Also

vsip_*ds*swap_*p* Vector/Matrix Swap

Swap elements between two vectors/matrices.

Functionality

for $j = 1, 2, \ldots, N\text{-}1$

$\tau \leftarrow a_j$

$b_j \leftarrow a_j$

$a_j \leftarrow \tau$

for $i = 0, 1, \ldots, M\text{-}1$

for $j = 0, 1, \ldots, N\text{-}1$

$\tau \leftarrow a_{i,j}$

$b_{i,j} \leftarrow a_{i,j}$

$a_{i,j} \leftarrow \tau$

Prototypes

```
void vsip_vswap_i(
                      const vsip_vview_i *a,
                      const vsip_vview_i *b);
void vsip_vswap_f(
                      const vsip_vview_f *a,
                      const vsip_vview_f *b);
void vsip_cvswap_f(
                      const vsip_cvview_f *a,
                      const vsip_cvview_f *b);
void vsip_mswap_i(
                      const vsip_mview_i *a,
                      const vsip_mview_i *b);
void vsip_mswap_f(
                      const vsip_mview_f *a,
                      const vsip_mview_f *b);
void vsip_cmswap_f(
                      const vsip_cmview_f *a,
                      const vsip_cmview_f *b);
```

Arguments

`*a`	View of input/output vector/matrix
`*b`	View of input/output vector/matrix

Return Value

None.

Restrictions

This function may not be done in-place.

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.
3. The input and output views must be identical views of the same block (in-place), or must not overlap.

Notes/References

Examples

```
/* example of swap */
#include<stdio.h>
#include "vsip.h"

#define L 7 /* A length */
#define PI 3.141592653589793

int main()
{
    vsip_vview_d*   dataA;
    vsip_vview_d*   dataB;
    int i;

    vsip_init((void *)0);
    dataA = vsip_vcreate_d(L, VSIP_MEM_NONE);
    dataB = vsip_vcreate_d(L, VSIP_MEM_NONE);

    vsip_vramp_d(0,2 * PI/(L-1),dataA);
    vsip_vcos_d(dataA,dataB);
    printf("  A       B   \n");
    for(i=0; i<L; i++)
       printf("%6.3f  %6.3f\n",vsip_vget_d(dataA,i),
            vsip_vget_d(dataB,i));
     printf("      Swap\n  A       B   \n");
    vsip_vswap_d(dataA,dataB);
    for(i=0; i<L; i++)
      printf("%6.3f  %6.3f\n",vsip_vget_d(dataA,i),vsip_vget_d(dataB,i));

    vsip_blockdestroy_d(vsip_vdestroy_d(dataA));
    vsip_blockdestroy_d(vsip_vdestroy_d(dataB));
    vsip_finalize((void *)0);
    return 0;
}
/* A       B
   0.000   1.000
   1.047   0.500
   2.094  -0.500
   3.142  -1.000
   4.189  -0.500
   5.236   0.500
   6.283   1.000
        Swap
    A       B
   1.000   0.000
   0.500   1.047
  -0.500   2.094
  -1.000   3.142
  -0.500   4.189
   0.500   5.236
   1.000   6.283 */
```

See Also

User-Specified By Element Functions

This section describes a set of functions that allows the user to specify a function to be applied by element to a set of vector/matrix/tensors view objects, and simple by element "get" and "put."

`vsip_`***`S`***`binary_`***`p`***	User-Specified Binary Function
`vsip_`***`S`***`bool_`***`p`***	User-Specified Boolean Binary Function
`vsip_`***`S`***`mary_`***`p`***	User-Specified M-ary Vector Function
`vsip_`***`S`***`nary_`***`p`***	User-Specified Stream Function
`vsip_`***`S`***`serialmary_`***`p`***	User-Specified Serial M-ary Function
`vsip_`***`S`***`unary_`***`p`***	User-Specified Unary Function

vsip_sbinary_p User-Specified Binary Function

Computes a user-specified binary scalar function, by element, of two vectors/matrices/tensors.

Functionality

Computes the binary vector/matrix/tensor function

$z_j \leftarrow f(x_j, y_j)$

$z_{i,j} \leftarrow f(x_{i,j}, y_{i,j})$

$z_{h,i,j} \leftarrow f(x_{h,i,j}, y_{h,i,j})$

element by element. The exact order of computation is undefined. The user specifies a binary function of two scalars that returns a scalar result.

Prototypes

```
void vsip_vbinary_f(
                      vsip_scalar_f (*f)(vsip_scalar_f, vsip_scalar_f),
                      const vsip_vview_f *x,
                      const vsip_vview_f *y,
                      const vsip_vview_f *z);
void vsip_vbinary_i(
                      vsip_scalar_i (*f)(vsip_scalar_i, vsip_scalar_i),
                      const vsip_vview_i *x,
                      const vsip_vview_i *y,
                      const vsip_vview_i *z);
void vsip_vbinary_vi(
                      vsip_scalar_vi (*f)(vsip_scalar_vi, vsip_scalar_vi),
                      const vsip_vview_vi *x,
                      const vsip_vview_vi *y,
                      const vsip_vview_vi *z);
void vsip_vbinary_mi(
                      vsip_scalar_mi (*f)(vsip_scalar_mi, vsip_scalar_mi),
                      const vsip_vview_mi *x,
                      const vsip_vview_mi *y,
                      const vsip_vview_mi *z);
void vsip_vbinary_ti(
                      vsip_scalar_ti (*f)(vsip_scalar_ti, vsip_scalar_ti),
                      const vsip_vview_ti *x,
                      const vsip_vview_ti *y,
                      const vsip_vview_ti *z);
void vsip_mbinary_f(
                      vsip_scalar_f (*f)(vsip_scalar_f, vsip_scalar_f),
                      const vsip_mview_f *x,
                      const vsip_mview_f *y,
                      const vsip_mview_f *z);
void vsip_mbinary_i(
                      vsip_scalar_i (*f)(vsip_scalar_i, vsip_scalar_i),
                      const vsip_mview_i *x,
                      const vsip_mview_i *y,
                      const vsip_mview_i *z);
```

```
void vsip_mbinary_bl(
                     vsip_scalar_bl (*f)(vsip_scalar_bl, vsip_scalar_bl),
                     const vsip_mview_bl *x,
                     const vsip_mview_bl *y,
                     const vsip_mview_bl *z);
void vsip_tbinary_f (
                     vsip_scalar_f (*f)(vsip_scalar_f, vsip_scalar_f),
                     const vsip_tview_f *x,
                     const vsip_tview_f *y,
                     const vsip_tview_f *z);
void vsip_tbinary_i (
                     vsip_scalar_i (*f)(vsip_scalar_i, vsip_scalar_i),
                     const vsip_tview_i *x,
                     const vsip_tview_i *y,
                     const vsip_tview_i *z);
void vsip_tbinary_bl(
                     vsip_scalar_bl (*f)(vsip_scalar_bl, vsip_scalar_bl),
                     const vsip_tview_bl *x,
                     const vsip_tview_bl *y,
                     const vsip_tview_bl *z);
```

Arguments

`*f()`	User specified binary function of two scalars
`*x`	Vector/matrix/tensor view object of source1 operand
`*y`	Vector/matrix/tensor view object of source2 operand
`*z`	Vector/matrix/tensor view object of result

Return Value

None.

Restrictions

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.
3. The input and output views must be identical views of the same block (in-place), or must not overlap.
4. The pointer to the user-specified function must be valid – non-*null*.

Notes/References

There are no complex versions of this function. This is a consequence of supporting the implementation of complex blocks with split storage, which is not compatible with a `vsip_cscalar_p` data type.

Examples

See Also

`vsip_sbool_p`, `vsip_smary_p`, `vsip_snary_p`, `vsip_sserialmary_p`, and `vsip_sunary_p`

vsip_sbool_p User-Specified Boolean Binary Function

Computes a user-specified boolean binary scalar function, by element, of two vectors/matrices/tensors.

Functionality

Computes the binary vector/matrix/tensor function

$z_j \leftarrow f(x_j, y_j)$

$z_{i,j} \leftarrow f(x_{i,j}, y_{i,j})$

$z_{h,i,j} \leftarrow f(x_{h,i,j}, y_{h,i,j})$

element by element. The exact order of computation is undefined. The user specifies a binary function of two scalars that returns a boolean result.

Prototypes

```
void vsip_vbool_f(
                      vsip_bool (*f)(vsip_scalar_f, vsip_scalar_f),
                      const vsip_vview_f *x,
                      const vsip_vview_f *y,
                      const vsip_vview_bl *z);
void vsip_vbool_i(
                      vsip_bool (*f)(vsip_scalar_i, vsip_scalar_i),
                      const vsip_vview_i *x,
                      const vsip_vview_i *y,
                      const vsip_vview_bl *z);
void vsip_vbool_vi(
                      vsip_bool (*f)(vsip_scalar_vi, vsip_scalar_vi),
                      const vsip_vview_vi *x,
                      const vsip_vview_vi *y,
                      const vsip_vview_bl *z);
void vsip_vbool_mi(
                      vsip_bool (*f)(vsip_scalar_mi, vsip_scalar_mi),
                      const vsip_vview_mi *x,
                      const vsip_vview_mi *y,
                      const vsip_vview_bl *z);
void vsip_vbool_ti(
                      vsip_bool (*f)(vsip_scalar_ti, vsip_scalar_ti),
                      const vsip_vview_ti *x,
                      const vsip_vview_ti *y,
                      const vsip_vview_bl *z);
void vsip_mbool_f(
                      vsip_bool (*f)(vsip_scalar_f, vsip_scalar_f),
                      const vsip_mview_f *x,
                      const vsip_mview_f *y,
                      const vsip_mview_bl *z);
void vsip_mbool_i(
                      vsip_bool (*f)(vsip_scalar_i, vsip_scalar_i),
                      const vsip_mview_i *x,
                      const vsip_mview_i *y,
                      const vsip_mview_bl *z);
```

```
void vsip_tbool_f (
                    vsip_bool (*f)(vsip_scalar_f, vsip_scalar_f),
                    const vsip_tview_f *x,
                    const vsip_tview_f *y,
                    const vsip_tview_bl *z);
void vsip_tbool_i (
                    vsip_bool (*f)(vsip_scalar_i, vsip_scalar_i),
                    const vsip_tview_i *x,
                    const vsip_tview_i *y,
                    const vsip_tview_bl *z);
```

Arguments

`*f()`	User specified binary function of two scalars, returning a boolean
`*x`	Vector/matrix/tensor view object of source1 operand
`*y`	Vector/matrix/tensor view object of source2 operand
`*z`	Boolean Vector/matrix/tensor view object of result

Return Value

None.

Restrictions

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.
3. The input and output views must be identical views of the same block (in-place), or must not overlap.
4. The pointer to the user-specified function must be valid – non-*null*.

Notes/References

There are no complex versions of this function. This is a consequence of supporting the implementation of complex blocks with split storage, which is not compatible with a `vsip_cscalar_p` data type.

Examples

See Also

`vsip_sbinary_p`, `vsip_smary_p`, `vsip_snary_p`, `vsip_sserialmary_p`, and `vsip_sunary_p`

vsip_smary_*p* User-Specified M-ary Function

Computes a user-specified specified m-ary scalar function, by element, of *m* vectors/matrices/tensors. The order of evaluation is not specified.

Functionality

This function applies the scalar function

Vector:

`void (*f)( vsip_scalar_p *p[],`	Pointer to array of *m*+1 pointers to scalars
`vsip_length m,`	*m*
`vsip_scalar_vi index)`	index *i*

Matrix:

`void (*f)( vsip_scalar_p *p[],`	Pointer to array of *m*+1 pointers to scalars
`vsip_length m,`	*m*
`vsip_scalar_mi index)`	index (*i, j*)

Tensor:

`void (*f)( vsip_scalar_p *p[],`	Pointer to array of *m*+1 pointers to scalars
`vsip_length m,`	*m*
`vsip_scalar_ti index)`	index (*h, i, j*)

by element to the elements of an *m*+1 array of vector/matrix/tensor view objects. Typical application is an *m*-ary scalar function that produces a single scalar output.

Since the user function takes an argument of a pointer to an array of *m*+1 pointers to scalars, it is free to treat any of the scalars as input, outputs, or inputs-outputs.

The exact order of computation is undefined.

Prototypes

```
void vsip_vmary_f(
                      void (*f)(  vsip_scalar_f *[],
                            vsip_length m,
                            vsip_scalar_vi index),
                      const vsip_vview_f *v[],
                      vsip_length m);
void vsip_vmary_i(
                      void (*f)(  vsip_scalar_i *[],
                            vsip_length m,
                            vsip_scalar_vi index),
                      const vsip_vview_i *v[],
                      vsip_length m);
void vsip_vmary_bl(
                      void (*f)(  vsip_scalar_bl *[],
                            vsip_length m,
                            vsip_scalar_vi index),
                      const vsip_vview_bl *v[],
                      vsip_length m);
```

```
void vsip_vmary_vi(
                      void (*f)(vsip_scalar_vi *[],
                            vsip_length m,
                            vsip_scalar_vi index),
                      const vsip_vview_vi *v[],
                      vsip_length m);
void vsip_vmary_mi(
                      void (*f)(vsip_scalar_mi *[],
                            vsip_length m,
                            vsip_scalar_vi index),
                      const vsip_vview_mi *v[],
                      vsip_length m);
void vsip_vmary_ti(
                      void (*f)(vsip_scalar_ti *[],
                            vsip_length m,
                            vsip_scalar_vi index),
                      const vsip_vview_ti *v[],
                      vsip_length m);
void vsip_mmary_f(
                      void (*f)(  vsip_scalar_f *[],
                            vsip_length m,
                            vsip_scalar_mi index),
                      const vsip_mview_f *v[],
                      vsip_length m);
void vsip_mmary_i(
                      void (*f)(  vsip_scalar_i *[],
                            vsip_length m,
                            vsip_scalar_mi index),
                      const vsip_mview_i *v[],
                      vsip_length m);
void vsip_mmary_bl(
                      void (*f)(  vsip_scalar_bl *[],
                            vsip_length m,
                            vsip_scalar_mi index),
                      const vsip_mview_bl *v[],
                      vsip_length m);
 void vsip_tmary_f(
                      void (*f)(  vsip_scalar_f *[],
                            vsip_length m,
                            vsip_scalar_ti index),
                      const vsip_tview_f *v[],
                      vsip_length m);
void vsip_tmary_i(
                      void (*f)(  vsip_scalar_i *[],
                            vsip_length m,
                            vsip_scalar_ti index),
                      const vsip_tview_i *v[],
                      vsip_length m);
```

```
void vsip_tmary_bl(
                      void (*f)(  vsip_scalar_bl *[],
                            vsip_length m,
                            vsip_scalar_ti index),
                      const vsip_tview_bl *v[],
                      vsip_length m);
```

Arguments

`*f()`	User specified function of an array of pointers to vector/matrix/tensor view objects, the length of the array, and the current element index; returning void.
`*v`	Array of *m*+1 pointers to vector/matrix/tensor view objects
`m`	One less than the number of elements in the array **v**. (*m*-ary order of the user-specified function.)

Return Value

None.

Restrictions

Errors

The arguments must conform to the following:

1. Input/output views must all be the same size.
2. All view objects must be valid.
3. The input and output views must be identical views of the same block (in-place), or must not overlap.
4. The pointer to the user-specified function must be valid – non-*null*.
5. *m* must be positive.

Notes/References

There are no complex versions of this function. This is a consequence of supporting the implementation of complex blocks with split storage, which is not compatible with a `vsip_cscalar_`*p* data type.

By convention, the vector of pointers to view objects is ordered: inputs, input/outputs, and last outputs.

Examples

1. To implement the common vector function VMMA: $y_j = (a_j \cdot b_j) + (c_j \cdot d_j)$

[`Static` *is just to limit the scope of the function name.*]

```
static void mma_f(vsip_scalar_f *q [], unsigned int m, vsip_index i){
  *q[4] = (*q[0] * *q[1]) + (*q[2] * *q[3]);
}

void user_vmma_f(const vsip_vview_f *a, const vsip_vview_f *b,
        const vsip_vview_f *c, const vsip_vview_f *d,
        const vsip_vview_f *y){
  const vsip_vview_f *qnary[5];
  qnary[0] = a; qnary[1] = b; qnary[2] = c; qnary[3] = d; qnary[4] = y;
  vsip_vmary_f(mma_sp, qnary, 4);
}
```

2.To sum *four* vectors together:

```
static void vmsum_d(vsip_scalar_d *v [], unsigned int M, vsip_index i){
  int i;
  *v[M-1] = *v[0];
  for(i=1; i<M-1; i++)*v[M-1] += *v[i];
}
void user_vsum4_d(const vsip_vview_d *a, const vsip_vview_d *b,
        const vsip_vview_d *c, const vsip_vview_d *d,
        const vsip_vview_d *y){
  const vsip_vview_d *qnary[5];
  qnary[0] = a; qnary[1] = b; qnary[2] = c; qnary[3] = d; qnary[4] = y;
  vsip_vmary_d(vmsum_d, qnary, 4);
}
```

3 .To implement the common matrix function MMMA: $y_{i,j} = (a_{i,j} \cdot b_{i,j}) + (c_{i,j} \cdot d_{i,j})$

```
static void mma_f(vsip_scalar_f *qary[], vsip_length m,
       vsip_scalar_mi index){
  *qary[4] = (*qary[0] * *qary[1]) + (*qary[2] * *qary[3]);
}
void user_mmma_f(const vsip_mview_f *a, const vsip_mview_f *b,
        const vsip_mview_f *c, const vsip_mview_f *d,
        const vsip_mview_f *y){
  const vsip_mview_f *X[5];
  X[0] = a; X[1] = b; X[2] = c; X[3] = d; X[4] = y;
  vsip_mrandom_f(mma_sp, X, 4);
}
```

[*Of course you should use a more efficient method for such lightweight functions.*]

See Also

`vsip_`***s***`bool_`***p***, `vsip_`***s***`binary_`***p***, `vsip_`***s***`nary_`***p***, `vsip_`***s***`serialmary_`***p***, and `vsip_`***s***`unary_`***p***

vsip_snary_*p* User-Specified Stream Function

Computes a user-specified scalar function that takes the vector/matrix/tensor index of the element and returns a scalar, by element, of a vector/matrix/tensor.

Functionality

Computes the "*null*-ary" (no element value arguments) vector/matrix/tensor function

$y_j \leftarrow \mathrm{f}(i)$

$y_{\mathrm{i,j}} \leftarrow \mathrm{f}(i, j)$

$y_{h,i,j} \leftarrow \mathrm{f}(h, i, j)$

element by element. The exact order of computation is undefined The user specifies a function of one/two/three indices that returns a scalar result.

Prototypes

```
void vsip_vnary_f(
                      vsip_scalar_f (*f)( vsip_index),
                      const vsip_vview_f *y);
void vsip_vnary_i(
                      vsip_scalar_i (*f)( vsip_index),
                      const vsip_vview_i *y);
void vsip_vnary_bl(
                      vsip_scalar_bl (*f)(vsip_index),
                      const vsip_vview_bl *y);
void vsip_vnary_vi(
                      vsip_scalar_vi (*f)(vsip_index),
                      const vsip_vview_vi *y);
void vsip_vnary_mi(
                      vsip_scalar_mi (*f)( vsip_index),
                      const vsip_vview_mi *y);
void vsip_vnary_ti(
                      vsip_scalar_ti (*f)( vsip_index),
                      const vsip_vview_ti *y);
void vsip_mnary_f(
                      vsip_scalar_f (*f)(vsip_index, vsip_index),
                      const vsip_mview_f *y);
void vsip_mnary_i(
                      vsip_scalar_i (*f)(vsip_index, vsip_index),
                      const vsip_mview_i *y);
void vsip_mnary_bl(
                      vsip_scalar_bl (*f)(vsip_index, vsip_index),
                      const vsip_mview_bl *y);
void vsip_tnary_f (
                      vsip_scalar_f (*f)(vsip_index, vsip_index, vsip_index),
                      const vsip_tview_f *y);
```

```
void vsip_tnary_i (
                      vsip_scalar_i (*f)(vsip_index, vsip_index, vsip_index),
                      const vsip_tview_i *y);
void vsip_tnary_bl(
                      vsip_scalar_bl (*f)(vsip_index,vsip_index,vsip_index),
                      const vsip_tview_bl *y);
```

Arguments

`*f()`	User specified *null*-ary function of one/two/three indices
`*y`	Vector/matrix/tensor view object of result

Return Value

None.

Restrictions

Errors

The arguments must conform to the following:

1. All view objects must be valid.
2. The pointer to the user-specified function must be valid – non-*null*.

Notes/References

There are no complex versions of this function. This is a consequence of supporting the implementation of complex blocks with split storage, which is not compatible with a `vsip_cscalar_p` data type.

Examples

See Also

`vsip_sbool_p`, `vsip_sbinary_p`, `vsip_smary_p`, `vsip_sserialmary_p`, and `vsip_sunary_p`

vsip_sserialmary_p User-Specified Serial M-ary Function

Computes a user-specified specified *m-ary* scalar function, by element, of *m* vectors/matrices/tensors. The order of evaluation is serial.

Functionality

This function applies the scalar function

Vector:

```
void (*f)( vsip_scalar_p *p[],      Pointer to array of m+1 pointers to scalars
           vsip_length m,           m
           vsip_scalar_vi index)    index i
```

Matrix:

```
void (*f)( vsip_scalar_p *p[],      Pointer to array of m+1 pointers to scalars
           vsip_length m,           m
           vsip_scalar_mi index)    index (i, j)
```

Tensor:

```
void (*f)( vsip_scalar_p *p[],      Pointer to array of m+1 pointers to scalars
           vsip_length m,           m
           vsip_scalar_ti index)    index (h, i, j)
```

by element to the elements of an *m*+1 array of vector/matrix/tensor view objects. Typical application is an *m*-ary scalar function that produces a single scalar output.

Since the user function takes an argument of a pointer to an array of *m*+1 pointers to scalars, it is free to treat any of the scalars as input, outputs, or inputs-outputs.

The semantics of the order of evaluation is serial from smallest to largest index. For matrices and tensors, this means that the index with the smallest intra-indices stride varies the fastest and the index with the largest intra-indices stride varies the slowest. For example, a matrix where the stride between successive elements of a row is one element is evaluated in the order:

for $i = 0$ to M-1
  for $j = 0$ to N-1
… ← f(**v**, m, (i, j))

Prototypes

```
void vsip_vserialmary_f(
                     void (*f)(  vsip_scalar_f *[],
                           vsip_length m,
                           vsip_scalar_vi index),
                     const vsip_vview_f *v[],
                     vsip_length m);
void vsip_vserialmary_i(
                     void (*f)(  vsip_scalar_i *[],
                           vsip_length m,
                           vsip_scalar_vi index),
                     const vsip_vview_i *v[],
                     vsip_length m);
```

```
void vsip_vserialmary_bl(
                     void (*f)(  vsip_scalar_bl *[],
                           vsip_length m,
                           vsip_scalar_vi index),
                     const vsip_vview_bl *v[],
                     vsip_length m);
void vsip_vserialmary_vi(
                     void (*f)(vsip_scalar_vi *[],
                           vsip_length m,
                           vsip_scalar_vi index),
                     const vsip_vview_vi *v[],
                     vsip_length m);
void vsip_vserialmary_mi(
                     void (*f)(vsip_scalar_mi *[],
                           vsip_length m,
                           vsip_scalar_vi index),
                     const vsip_vview_mi *v[],
                     vsip_length m);
void vsip_vserialmary_ti(
                     void (*f)(vsip_scalar_ti *[],
                           vsip_length m,
                           vsip_scalar_vi index),
                     const vsip_vview_ti *v[],
                     vsip_length m);
void vsip_mserialmary_f(
                     void (*f)(  vsip_scalar_f *[],
                           vsip_length m,
                           vsip_scalar_mi index),
                     const vsip_mview_f *v[],
                     vsip_length m);
void vsip_mserialmary_i(
                     void (*f)(  vsip_scalar_i *[],
                           vsip_length m,
                           vsip_scalar_mi index),
                     const vsip_mview_i *v[],
                     vsip_length m);
void vsip_mserialmary_bl(
                     void (*f)(  vsip_scalar_bl *[],
                           vsip_length m,
                           vsip_scalar_mi index),
                     const vsip_mview_bl *v[],
                     vsip_length m);
 void vsip_tserialmary_f(
                     void (*f)(  vsip_scalar_f *[],
                           vsip_length m,
                           vsip_scalar_ti index),
                     const vsip_tview_f *v[],
                     vsip_length m);
```

```
void vsip_tserialmary_i(
                      void (*f)(  vsip_scalar_i *[],
                            vsip_length m,
                            vsip_scalar_ti index),
                      const vsip_tview_i *v[],
                      vsip_length m);
void vsip_tserialmary_bl(
                      void (*f)(  vsip_scalar_bl *[],
                            vsip_length m,
                            vsip_scalar_ti index),
                      const vsip_tview_bl *v[],
                      vsip_length m);
```

Arguments

`*f()`	User specified function of an array of pointers to vector/matrix/tensor view objects, the length of the array, and the current element index; returning void.
`*v`	Array of *m*+1 pointers to vector/matrix/tensor view objects
`m`	One less than the number of elements in the array **v**. (*m*-ary order of the user-specified function.)

Return Value

None.

Restrictions

Errors

The arguments must conform to the following:

1. Input/output views must all be the same size.
2. All view objects must be valid.
3. The pointer to the user-specified function must be valid – non-*null*.
4. The arity, *m*, of the function must be positive.

Notes/References

Since the order of evaluation is specified, it is allowed for the input and output to overlap.

There are no complex versions of this function. This is a consequence of supporting the implementation of complex blocks with split storage, which is not compatible with a `vsip_cscalar_p` data type.

By convention, the vector of pointers to view objects is ordered: inputs, input/outputs, and last outputs.

Examples

See Also

`vsip_sbool_p`, `vsip_sbinary_p`, `vsip_smary_p`, `vsip_snary_p`, and `vsip_sunary_p`

vsip_sunary_p User-Specified Unary Function

Computes a user-specified unary scalar function, by element, of a vector/matrix/tensor.

Functionality

Computes the unary tensor function $x_{h,i,j} = f(h, i, j)$ element by element where the exact order of computation is undefined. The user-specified function is a function of a scalar tensor index and returns a result scalar.

This function computes the unary vector/matrix/tensor function

$y_j \leftarrow f(x_j)$

$y_{i,j} \leftarrow f(x_{i,j})$

$y_{h,i,j} \leftarrow f(x_{h,i,j})$

element by element. The exact order of computation is undefined. The user specifies a unary function of a scalar that returns a scalar result.

Prototypes

```
void vsip_vunary_f(
                     vsip_scalar_f (*f)(vsip_scalar_f),
                     const vsip_vview_f *x,
                     const vsip_vview_f *y);
void vsip_vunary_i(
                     vsip_scalar_i (*f)(vsip_scalar_i),
                     const vsip_vview_i *x,
                     const vsip_vview_i *y);
void vsip_vunary_vi(
                     vsip_scalar_vi (*f)(vsip_scalar_vi),
                     const vsip_vview_vi *x,
                     const vsip_vview_vi *y);
void vsip_vunary_mi(
                     vsip_scalar_mi (*f)(vsip_scalar_mi),
                     const vsip_vview_mi *x,
                     const vsip_vview_mi *y);
void vsip_vunary_ti(
                     vsip_scalar_ti (*f)(vsip_scalar_ti),
                     const vsip_vview_ti *x,
                     const vsip_vview_ti *y);
void vsip_munary_f(
                     vsip_scalar_f (*f)(vsip_scalar_f),
                     const vsip_mview_f *x,
                     const vsip_mview_f *y);
void vsip_munary_i(
                     vsip_scalar_i (*f)(vsip_scalar_i),
                     const vsip_mview_i *x,
                     const vsip_mview_i *y);
```

```
void vsip_tunary_f (
                     vsip_scalar_f (*f)(vsip_scalar_f),
                     const vsip_tview_f *x,
                     const vsip_tview_f *y);
void vsip_tunary_i (
                     vsip_scalar_i (*f)(vsip_scalar_i),
                     const vsip_tview_i *x,
                     const vsip_tview_i *y);
```

Arguments

`*f()`	User specified function of one/two/three scalar indices
`*x`	Vector/matrix/tensor view object of result

Return Value

None.

Restrictions

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.
3. The input and output views must be identical views of the same block (in-place), or must not overlap.
4. The pointer to the user-specified function must be valid – non-*null*.

Notes/References

There are no complex versions of this function. This is a consequence of supporting the implementation of complex blocks with split storage, which is not compatible with a `vsip_cscalar_p` data type.

Examples

See Also

`vsip_sbool_p`, `vsip_sbinary_p`, `vsip_smary_p`, `vsip_snary_p`, and `vsip_sserialmary`

SIGNAL PROCESSING FUNCTIONS

VSIPL Signal Processing - Introduction

This chapter provides specifications for the FFT, Window, Convolution/Correlation, Filter and Miscellaneous routines for the VSIPL library.

VSIPL FFT Routines

Introduction

All FFT routines compute a forward or inverse operation with a user provided scaling.

1D Fourier transform operations are supported for all values of N. The basic implementation requirement is for an $\mathbf{O}(N \log N)$ fast algorithm for the cases $N = 2^n$ or $N = 3 \cdot 2^n$, where n is a nonnegative integer. Some implementations may provide fast algorithms for other combinations of small prime factors and may even handle the general case of large prime factors or prime sizes. When an implementation does not provide a fast algorithm, a discrete Fourier transform of $\mathbf{O}(N^2)$ or faster will be performed.

2D FFT operations are supported for all values of M and N. However, the basic implementation requirement is for an $\mathbf{O}(M \cdot N \log (M \cdot N))$ fast algorithm for the cases of $M = 2^m$ or $M = 3 \cdot 2^m$ and $N = 2^n$ or $N = 3 \cdot 2^n$, where m and n are nonnegative integers. Some implementations may provide fast algorithms for other combinations of small prime factors and may even handle the general case of large prime factors or prime sizes. When an implementation does not provide a fast algorithm, a DFT of $\mathbf{O}(M^2 \cdot N + M \cdot N^2)$ or faster will be performed.

3D FFT operations are supported for all values of M, N, and P. However, the basic implementation requirement is for an $\mathbf{O}(M \cdot N \cdot P \log (M \cdot N \cdot P))$ fast algorithm for the cases of $M = 2^m$ or $M = 3 \cdot 2^m$, $N = 2^n$ or $N = 3 \cdot 2^n$ and $P = 2^p$ or $P = 3 \cdot 2^p$, where m, n, and p are nonnegative integers. Some implementations may provide fast algorithms for other combinations of small prime factors and may even handle the general case of large prime factors or prime sizes. When an implementation does not provide a fast algorithm, a DFT of $\mathbf{O}(M^2 \cdot N \cdot P + M \cdot N^2 \cdot P + M \cdot N \cdot P^2)$ or faster will be performed.

1D FFT

`vsip_`**`ccfftop_create_`**`f`	initialize FFT object for the routine `vsip_`**`ccfftop_`**`f`
`vsip_`**`ccfftip_create_`**`f`	initialize FFT object for the routine `vsip_`**`ccfftip_`**`f`
`vsip_`**`crfftop_create_`**`f`	initialize FFT object for the routine `vsip_`**`crfftop_`**`f`
`vsip_`**`rcfftop_create_`**`f`	initialize FFT object for the routine `vsip_`**`rcfftop_`**`f`
`vsip_`**`ccfftop_`**`f`	complex-to-complex out-of-place
`vsip_`**`ccfftip_`**`f`	complex-to-complex in-place
`vsip_`**`crfftop_`**`f`	complex-to-real out-of-place
`vsip_`**`rcfftop_`**`f`	real-to-complex out-of-place

VSIPL Multiple 1D FFT

vsip_**ccfftmop_create**_*f*	initialize FFT object for the routine vsip_**ccfftm**_*f*
vsip_**ccfftmip_create**_*f*	initialize FFT object for the routine vsip_**ccfftm**_*f*
vsip_**crfftmop_create**_*f*	initialize FFT object for the routine vsip_**crfftm**_*f*
vsip_**rcfftmop_create**_*f*	initialize FFT object for the routine vsip_**rcfftm**_*f*
vsip_**ccfftmop**_*f*	complex-to-complex out-of-place
vsip_**ccfftmip**_*f*	complex-to-complex in-place
vsip_**crfftmop**_*f*	complex-to-real out-of-place
vsip_**rcfftmop**_*f*	real-to-complex out-of-place

2D FFT

vsip_**ccfft2dop_create**_*f*	initialize FFT object for the routine vsip_**ccfft2dop**_*f*
vsip_**ccfft2dip_create**_*f*	initialize FFT object for the routine vsip_**ccfft2dip**_*f*
vsip_**crfft2dop_create**_*f*	initialize FFT object for the routine vsip_**crfft2dop**_*f*
vsip_**rcfft2dop_create**_*f*	initialize FFT object for the routine vsip_**rcfft2dop**_*f*
vsip_**ccfft2dop**_*f*	complex-to-complex out-of-place
vsip_**ccfft2dip**_*f*	complex-to-complex in-place
vsip_**crfft2dop**_*f*	complex-to-real out-of-place
vsip_**rcfft2dop**_*f*	real-to-complex out-of-place

VSIPL 3D FFT

vsip_**ccfft3dop_create**_*f*	initialize FFT object for the routine vsip_**ccfft3dop**_*f*
vsip_**ccfft3dip_create**_*f*	initialize FFT object for the routine vsip_**ccfft3dip**_*f*
vsip_**crfft3dop_create**_*f*	initialize FFT object for the routine vsip_**crfft3dop**_*f*
vsip_**rcfft3dop_create**_*f*	initialize FFT object for the routine vsip_**rcfft3dop**_*f*
vsip_**ccfft3dop**_*f*	complex-to-complex out-of-place
vsip_**ccfft3dip**_*f*	complex-to-complex in-place
vsip_**crfft3dop**_*f*	complex-to-real out-of-place
vsip_**rcfft3dop**_*f*	real-to-complex out-of-place

FFT Object Utility Routines

vsip_**fft_destroy**_*f*	destroy the FFT object
vsip_**fft_getattr**_*f*	get the attributes of the FFT object
vsip_**fft_setwindow**_*f*	window the FFT input as part of the FFT calculation

Summary of VSIPL FFT Routines

The following table lists the functionality supported by the FFT routines

1D	Multi 1D	2D	3D
vsip_ccfftop_*f* vsip_ccfftip_*f*	vsip_ccfftmop_*f* vsip_ccfftmip_*f*	vsip_ccfft2dop_*f* vsip_ccfft2dip_*f*	vsip_ccfft3dop_*f* vsip_ccfft3dip_*f*
vsip_crfftop_*f*	vsip_crfftmop_*f*	vsip_crfft2dop_*f*	vsip_crfft3dop_*f*
vsip_rcfftop_*f*	vsip_rcfftmop_*f*	vsip_rcfft2dop_*f*	vsip_rcfft3dop_*f*

Computational Routines

1D	Multi 1D	2D	3D
vsip_ccfftop_create_*f* vsip_ccfftip_create_*f*	vsip_ccfftmop_create_*f* vsip_ccfftmip_create_*f*	vsip_ccfft2dop_create_*f* vsip_ccfft2dip_create_*f*	vsip_ccfft3dop_create_*f* vsip_ccfft3dip_create_*f*
vsip_crfftop_create_*f*	vsip_crfftmop_create_*f*	vsip_crfft2dop_create_*f*	vsip_crfft3dop_create_*f*
vsip_rcfftop_create_*f*	vsip_rcfftmop_create_*f*	vsip_rcfft2dop_create_*f*	vsip_rcfft3dop_create_*f*

FFT Object Initialization Routines

References

Charles Van Loan, **Computational Frameworks for the Fast Fourier Transform**, Society for Industrial and Applied Mathematics, 1992.

Winthrop W. Smith and Joanne M. Smith**, Handbook of Real-Time Fast Fourier Transforms**, IEEE Press, 1995.

VSIPL Window Routines

Introduction

VSIPL provides only a minimum set of common window functions. All window routines create a block of the requested window length, create and bind a real vector of unit stride, zero offset, and window length to the block, and return the vector initialized to the window weights. For other windows the user can bind a block to a user array of pre-computed weights and admit the data to VSIPL, or can compute a set of appropriate weights using VSIPL functionality.

Window Routines

`vsip_`**`vcreate_hanning_`***`f`*	create a Hanning window vector
`vsip_`**`vcreate_blackman_`***`f`*	create a Blackman window vector
`vsip_`**`vcreate_kaiser_`***`f`*	create a Kaiser window vector
`vsip_`**`vcreate_cheby_`***`f`*	create a Dolph-Chebyshev window vector

References

Alan V. Oppenheim, Ronald W. Schaefer, **Discrete-Time Signal Processing**, Prentice-Hall, Inc., 1989.

Edited by the IEEE ASSP Society, **Programs for Digital Signal Processing**, IEEE Press, 1979.

Ronald Diderich, **Calculating Chebyshev Coefficients via the Discrete Fourier Transform,** *Proceedings of the IEEE*, pg. 1395, October 1974.

Albert H. Nuttall, **Generation of Dolph-Chebyshev Weights via a Fast Fourier Transform,** *Proceedings of the IEEE*, pg. 1396, October 1974.

VSIPL Filter Routines

Introduction

Filter Routines

vsip_*d*fir_create_*f*	Create Decimated FIR Filter
vsip_*d*firflt_*f*	Decimated FIR Filter
vsip_*d*fir_getattr_*f*	Get FIR Filter Attributes
vsip_*d*fir_destroy_*f*	Destroy FIR Filter
vsip_*d*fir_reset_*f*	Reset FIR filter object to initial state
vsip_iir_create_*f*	Create IIR Filter
vsip_iirflt_*f*	IIR Filter
vsip_iir_getattr_*f*	Get IIR Filter Attributes
vsip_iir_destroy_*f*	Destroy IIR Filter

References

Alan V. Oppenheim, Ronald W. Schaefer, **Discrete-Time Signal Processing**, Prentice-Hall, Inc., 1989.

VSIPL Convolution/Correlation Routines

Introduction

1D Convolution Routines

vsip_*d*conv1d_create_*f*	create 1D filter convolution object
vsip_*d*convolve1d_*f*	compute 1D convolution
vsip_*d*conv1d_destroy_*f*	destroy a 1D convolution object
vsip_*d*conv1d_getattr_*f*	get 1D convolution object attributes

2D Convolution Routines

vsip_conv2d_create_*f*	create 2D filter convolution object
vsip_convolve2d_*f*	compute 2D convolution
vsip_conv2d_destroy_*f*	destroy a 2D convolution object
vsip_conv2d_getattr_*f*	get 2D convolution object attributes

1D Correlation Routines

vsip_corr1d_create_*f*	create 1D correlation object
vsip_ccorr1d_create_*f*	create 1D complex correlation object
vsip_correlate1d_*f*	compute 1D correlation
vsip_ccorrelate1d_*f*	compute 1D complex correlation
vsip_corr1d_destroy_*f*	destroy a 1D correlation object
vsip_ccorr1d_destroy_*f*	destroy a 1D complex correlation object
vsip_corr1d_getattr_*f*	get 1D correlation object attributes
vsip_ccorr1d_getattr_*f*	get 1D complex correlation object attributes

2D Correlation Routines

vsip_corr2d_create_*f*	create 2D correlation object
vsip_ccorr2d_create_*f*	create 2D complex correlation object
vsip_correlate2d_*f*	compute 2D correlation
vsip_ccorrelate2d_*f*	compute 2D complex correlation
vsip_corr2d_destroy_*f*	destroy a 2D correlation object
vsip_ccorr2d_destroy_*f*	destroy a 2D complex correlation object
vsip_corr2d_getattr_*f*	get 2D correlation object attributes
vsip_ccorr2d_getattr_*f*	get 2D complex correlation object attributes

References

Alan V. Oppenheim, Ronald W. Schaefer, **Discrete-Time Signal Processing**, Prentice-Hall, Inc., 1989.

VSIPL Miscellaneous Routines

Introduction

Miscellaneous Routines

vsip_**histo**_*f*	Compute the histogram of a vector (matrix)
vsip_*ds***freqswap**_*f*	Frequency swap (map zero frequency to center)

References

FFT Functions

vsip_**ccfftX**_*f*	FFT Complex to Complex
vsip_**crfftop**_*f*	FFT Complex to Real
vsip_**rcfftop**_*f*	FFT Real to Complex
vsip_*d***fftX_create**_*f*	Create 1D FFT Object
vsip_**ccfftmX**_*f*	FFT Multiple Complex to Complex
vsip_**crfftmop**_*f*	FFT Multiple Complex to Real
vsip_**rcfftmop**_*f*	FFT Multiple Real to Complex
vsip_**fft_setwindow**_*f*	Set a FFT Window
vsip_*d***fftmX_create**_*f*	Create Multiple FFT Object
vsip_**ccfft2dx**_*f*	2D FFT Complex to Complex
vsip_**crfft2dop**_*f*	2D FFT Complex to Real
vsip_**rcfft2dop**_*f*	2D FFT Real to Complex
vsip_**fftm_setwindow**_*f*	Set a Multiple FFT Window
vsip_*d***fft2dX_create**_*f*	Create 2D FFT Object
vsip_**ccfft3dX**_*f*	3D FFT Complex to Complex
vsip_**crfft3dop**_*f*	3D FFT Complex to Real
vsip_**rcfft3dop**_*f*	3D FFT Real to Complex
vsip_*d***fft3dX_create**_*f*	Create 3D FFT Object
vsip_**fft***n***_destroy**_*f*	Destroy FFT Object
vsip_**fft***n***_getattr**_*f*	FFT Get Attributes

vsip_ccfftx_f FFT Complex to Complex

Apply a complex-to-complex Fast Fourier Transform (FFT)

Functionality

Computes a complex-to-complex Fast Fourier Transform (FFT) of the complex vector $x = (x_n)$, and stores the results in the complex vector $y = (y_k)$.

$$y_k \leftarrow scale \cdot \sum_{n=0}^{N-1} x_n W_N^{k \cdot n}, \text{ for } k = 0, 1, \ldots, N\text{-}1$$

where:

$$W_N \equiv e^{sign \cdot j2\pi/N}$$

$$j \equiv \sqrt{-1}$$

sign = -1 for a *forward* transform or +1 for an *inverse* transform

Prototypes

Out-of-place:

```
void vsip_ccfftop_f (
                     const vsip_fft_f *fft,
                     const vsip_cvview_f *x,
                     const vsip_cvview_f *y);
```

In-place:

```
void vsip_ccfftip_f (
                     const vsip_fft_f *fft,
                     const vsip_cvview_f *xy);
```

Arguments

`*fft`	Pointer to a 1D FFT object, created by `vsip_ccfftx_create_f`
`*x`	View of input complex vector of length *N*
`*y`	View of output complex vector of length *N*
`*xy`	View of input/output complex vector of length *N*

Return Value

None.

Restrictions

Errors

The arguments must conform to the following:

1. All objects must be valid.
2. The FFT object must be a complex-to-complex FFT object.
3. `vsip_ccfftop_f` requires an *out-of-place* FFT object, `vsip_ccfftip_f` requires an *in-place* FFT object.
4. The input and output must be complex vector views of length *N*, where *N* is obtained from the FFT object.
5. For an out-of-place FFT, the input and output vector views must not overlap.

Notes/References

Examples

In-place complex-to-complex FFT

```
#include <stdio.h>
#include <vsip.h>

#define N 8
#define ACMPLX(x) vsip_real_d(x),\
   (vsip_imag_d(x) < 0.0 ? "-i" : "+i"), fabs(vsip_imag_d(x))
#define SCMPLX "%g%s%g"

int main()
{
  int i;
  vsip_cscalar_d  z;
  vsip_rand_state *state;
  vsip_scalar_d   data[2*N]; /* a public data space for I/O */
  vsip_scalar_d   *ptr1, *ptr2; /* for use in release */
  vsip_fft_d      *ccfftNip;
  vsip_cblock_d   *block;
  vsip_cvview_d   *inout;

  /* Initialize Random Number Generator */
  int seed =0, num_procs=1, id=1;

  vsip_init((void *)0);
  state = vsip_randcreate(seed, num_procs, id, VSIP_PRNG);
  ccfftNip =  /* Create an in-place Cmplx->Cmplx N-pt FFT */
    vsip_ccfftip_create_d(N, 1.0, VSIP_FFT_FWD, 1, VSIP_ALG_TIME);
  /* Create a block object and bind it to the array data */
  block = vsip_cblockbind_d(data, NULL, N, VSIP_MEM_NONE);
  inout = vsip_cvbind_d(block, 0, 1, N);

  /* Admit block to VSIPL for processing and initialize with
     complex Gaussian noise N(0,1) */
  vsip_cblockadmit_d(block, VSIP_FALSE);
  vsip_cvrandn_d(state, inout);
  printf("\nComplex Input Vector\n");
  for(i=0; i<N; i++) {
    z = vsip_cvget_d(inout, i);
    printf(SCMPLX "\n", ACMPLX(z));
  }

  /* Compute an in-place Cmplx->Cmplx N-pt FFT using the ccfftNip object */
  vsip_ccfftip_d(ccfftNip, inout);

  /* Print it */
  /* Release the block from VSIPL so that data can be directly accessed */
  vsip_cblockrelease_d(block,VSIP_TRUE,&ptr1,&ptr2);
  printf("\nComplex Output Vector (Real, Imag)\n");
  for(i=0; i<N; i++)
    printf("(%g, %g)\n", data[2*i], data[2*i+1]);

  /* Destroy the ccfftNip and block objects */
  vsip_fft_destroy_d(ccfftNip);
  vsip_cvalldestroy_d(inout);
  vsip_randdestroy(state);
  vsip_finalize((void *)0);
  return(0);
}
```

```
Output
Complex Input Vector
-0.615549+i0.217406
0.810217+i1.18112
1.46004+i0.540183
-1.27425+i0.688241
-0.956159-i0.135591
0.434556-i0.432679
-0.209061+i0.719197
-0.0821027-i1.4201
Complex Output Vector (Real, Imag)
(-0.432307, 1.35778)
(3.90216, -1.08846)
(-1.34239, -3.77869)
(2.04297, 2.94914)
(-0.209147, 1.32461)
(-3.57896, -1.54376)
(-4.30299, 1.42356)
(-1.00372, 1.09507)
```

See Also

`vsip_dfft`***x***`_create_`***f***, and `vsip_fft`***n***`_destroy_`***f***

vsip_crfftop_*f* FFT Complex to Real

Apply a complex-to-real Fast Fourier Transform (FFT)

Functionality

Computes a complex-to-real (*inverse*) Fast Fourier Transform (FFT) of the complex vector $x = (x_n)$, and stores the results in the real vector $y = (y_k)$.

$$y_k \leftarrow scale \cdot \sum_{n=0}^{N-1} x_n W_N^{k \cdot n}, \text{ for } k = 0, 1, \ldots, N\text{-}1$$

(See Notes/References for more details.)

where:

$$W_N \equiv e^{+j2\pi/N}$$

$$j \equiv \sqrt{-1}$$

Prototypes

```
void vsip_crfftop_f (
                     const vsip_fft_f *fft,
                     const vsip_cvview_f *x,
                     const vsip_vview_f *y);
```

Arguments

`*fft`	Pointer to a 1D FFT object, created by `vsip_crfftx_create_f`
`*x`	View of input complex vector of length *N*/2 +1 where the value indexed as 0 contains the DC (0 frequency) value and the value indexed as *N/2* contains the folding frequency (one half the sample rate) value.
`*y`	View of output real vector of length *N*

Return Value

None.

Restrictions

Only unit stride views are supported.

The length, *N*, must be even.

Errors

The arguments must conform to the following:

1. All objects must be valid.
2. The FFT object must be a complex-to-real out-of-place FFT object.
3. The input must be a complex vector view of length *N*/2 +1, where *N* is obtained from the FFT object.
4. The output must be a real vector view of length *N*, where *N* is obtained from the FFT object.
5. For an out-of-place FFT, the input and output vector views must not overlap.
6. The input and output vector views must be unit stride.

Notes/References

Generally, the FFT transforms a complex sequence into a complex sequence. However, in certain applications we may know the output sequence is real. Often, this is the case because

the complex input sequence was the transform of a real sequence. In this case, you can save about half of the computational work.

For the output sequence, y, to be a real sequence, the following identity on the input sequence, x, must be true:

$x_n = x^*_{N-n}$, for $\lfloor N/2 \rfloor < n < N$

The input values x_n for $n > \lfloor N/2 \rfloor$ need not be supplied; they can be inferred from the first half of the input.

Thus, in the complex-to-real routine, `vsip_crfftx_f`, x is a complex vector of length $\lfloor N/2 \rfloor+1$ and y is a real vector of length N. Even though only $\lfloor N/2 \rfloor+1$ input complex values are supplied, the size of the transform is still N in this case, because implicitly you are using the FFT formula for a sequence of length N.

The first value of the input vector, x_o must be a real number that is, it must have zero imaginary part. The first value coresponds to the zero (DC) frequency component of the data. Since we restrict N to be an even number then the last value of the input vector, $x_{N/2}$, must also be real. The last value coresponds to one half the Nyquist rate (or sample rate). This value is sometimes called the folding frequency The routine `vsip_crfftop_f` assumes that these values are real; if you specify a nonzero imaginary part, it is ignored.

Examples

```
#include <stdio.h>
#include <vsip.h>

#define N 8
#define ACMPLX(x) vsip_real_d(x), \
     (vsip_imag_d(x) < 0.0 ? "-i" : "+i"), fabs(vsip_imag_d(x))
#define SCMPLX "%g%s%g"

int main()
{
  int i;
  vsip_cscalar_d z;
  vsip_scalar_d  data[N]; /* a public data space for I/O */
  vsip_fft_d     *rcfftNop;
  vsip_block_d   *block;
  vsip_vview_d   *xin;
  vsip_cvview_d  *yout;

  vsip_init((void *)0);
  rcfftNop =  /* Create an out-of-place Real->Cmplx N-pt FFT */
  vsip_rcfftop_create_d(N, 1.0, 1, VSIP_ALG_TIME);

  /* Create a block object and bind it to the array data */
  block = vsip_blockbind_d(data, N, VSIP_MEM_NONE);
  xin = vsip_vbind_d(block, 0, 1, N);

  /* Create another block and complex vector view for the
     symmetric output */
  yout = vsip_cvcreate_d((N/2)+1, VSIP_MEM_NONE);

  /* Admit block to VSIPL for processing and initialize with a
     linear ramp */
```

```
  vsip_blockadmit_d(block, VSIP_FALSE); vsip_vramp_d(0.0, 1.0, xin);

  /* Compute an out-of-place Real->Cmplx N-pt FFT using the rcfftNop
     object */
  vsip_rcfftop_d(rcfftNop, xin, yout);

  /* print it */
  printf("Real Input Vector\n");
  for(i=0; i<N; i++) {
    printf("%g\n", vsip_vget_d(xin,i));
  }
  printf("\nComplex Output Vector\n");
  for(i=0; i<(N/2)+1; i++) {
    z = vsip_cvget_d(yout,i);
    printf(SCMPLX "\n", ACMPLX(z));
  }

  /* Destroy the rcfftNop, blocks, and view objects */
  vsip_fft_destroy_d(rcfftNop);
  vsip_cvalldestroy_d(xin);
  vsip_cvalldestroy_d(yout);
  vsip_finalize((void *)0);
  return(0);
}
```

See Also

`vsip_`*d*`fftx_create_`*f*, `vsip_rcfftop_`*f*, and `vsip_fft`*n*`_destroy_`*f*

vsip_rcfftop_*f* FFT Real to Complex

Apply a real-to-complex Fast Fourier Transform (FFT)

Functionality

Computes a real-to-complex (*forward*) Fast Fourier Transform (FFT) of the real vector $x = (x_n)$, and stores the results in the complex vector $y = (y_k)$.

$$y_k \leftarrow scale \cdot \sum_{n=0}^{N-1} x_n W_N^{k \cdot n}, \text{ for } k = 0, \ldots, N\text{-}1$$

(See Notes/References for more details.)

where:

$$W_N \equiv e^{-j2\pi/N}$$

$$j \equiv \sqrt{-1}$$

Prototypes

Out-of-place:

```
void vsip_rcfftop_f (
                     const vsip_fft_f *fft,
                     const vsip_vview_f *x,
                     const vsip_cvview_f *y);
```

Arguments

`*fft`	Pointer to a 1D FFT object, created by `vsip_rcfftx_create_f`
`*x`	View of input real vector of length *N*
`*y`	View of output complex vector of length *N*/2+1. The first value placed in the output vector is the DC frequency value, and the last value is the folding frequency value equal to one half the sample rate of the input vector.

Return Value

None.

Restrictions

Only unit stride views are supported.

The length, *N*, must be even.

Errors

The arguments must conform to the following:

1. All objects must be valid.
2. The FFT object must be a real-to-complex out-of-place FFT object.
3. *N* even, where *N* is obtained from the FFT object.
4. The input must be a real vector view of even length *N*.
5. The output must be a complex vector view of length (*N*/2) + 1.
6. For an out-of-place FFT, the input and output vector views must not overlap.
7. The input and output vector views must be unit stride.

Notes/References

The mathematical definition of the Fourier transform takes a sequence *N* complex values and transforms it to another sequence of *N* complex values. A complex-to-complex FFT routine, such as `vsip_ccfft`***`x_f`***, will take *N* complex input values, and produce *N* complex output values.

The purpose of a separate real-to-complex FFT routine is efficiency. Since the input data are real, you can make use of this fact to save almost half of the computational work. The theory of Fourier transforms tells us that for real input data, you have to compute only the first $\lfloor N/2 \rfloor+1$ complex output values, because the remaining values can be computed from the first half of the values by the simple formula:

$$y_k = y^*_{N-k}, \text{ for } \lfloor N/2 \rfloor < k < N$$

For real input data, the first output value, y_0, will always be a real number; therefore, the imaginary part will be exactly zero. The first output value is sometimes called the DC component of the FFT and corresponds to zero frequency. Since we restrict *N* to be an even number, $y_{N/2}$, will also be real and thus, have zero imaginary part. The last value is called the folding frequency and is equal to one half the sample rate of the input data.

Thus, in the real-to-complex routine, `vsip_rcfftop_`***`f`***, *x* is a real array of even length *N* and *y* is a complex array of length *N*/2 +1.

Examples

Out-Of-Place Real-to-Complex FFT

```
#include <stdio.h>
#include <vsip.h>

#define N 8
#define ACMPLX(x) vsip_real_d(x), \
   (vsip_imag_d(x) < 0.0 ? "-i" : "+i"), fabs(vsip_imag_d(x))
#define SCMPLX "%g%s%g"

int main()
{
  int i;
  vsip_cscalar_d z;
  vsip_scalar_d data[N]; /* a public data space for I/O */
  vsip_fft_d    *rcfftNop;
  vsip_block_d  *block;
  vsip_vview_d  *xin;
  vsip_cvview_d *yout;
```

```
  vsip_init((void *)0);
  rcfftNop = /* Create an out-of-place Real->Cmplx N-pt FFT */
  vsip_rcfftop_create_d(N, 1.0, 1, VSIP_ALG_TIME);

  /* Create a block object and bind it to the array data */
  block = vsip_blockbind_d(data, N, VSIP_MEM_NONE); xin =
            vsip_vbind_d(block, 0, 1, N);

  /* Create another block and complex vector view for the
     symmetric output */
  yout = vsip_cvcreate_d((N/2)+1, VSIP_MEM_NONE);

  /* Admit block to VSIPL for processing and initialize with a
     linear ramp */
  vsip_blockadmit_d(block, VSIP_FALSE);
  vsip_vramp_d(0.0, 1.0, xin);

  /* Compute an out-of-place Real->Cmplx N-pt FFT using the
     rcfftNop object */
  vsip_rcfftop_d(rcfftNop, xin, yout);

  /* print it */
  printf("Real Input Vector\n");
  for(i=0; i<N; i++) {
    printf("%g\n", vsip_vget_d(xin,i));
  }
  printf("\nComplex Output Vector\n");
  for(i=0; i<(N/2)+1; i++) {
    z = vsip_cvget_d(yout,i);
    printf(SCMPLX "\n", ACMPLX(z));
  }
  /* Destroy the rcfftNop, blocks, and view objects */
  vsip_fft_destroy_d(rcfftNop);
  vsip_cvalldestroy_d(xin);
  vsip_cvalldestroy_d(yout);
  vsip_finalize((void *)0);
  return(0);
}
```

See Also

`vsip_dfftx_create_f`, `vsip_crfftop_f`, and `vsip_fftn_destroy_f`

vsip_*d*fft*x*_create_*f* Create 1D FFT Object

Create a 1D FFT object.

Functionality

Creates a 1D FFT object. The FFT object encapsulates the information on what type of FFT is to be computed and may at the implementor's discretion partially pre-compute or optimize the FFT based on this information.

The FFT object is used to compute a complex to complex, real to complex, or complex-to-real Fast Fourier Transform (FFT) of vector $x = (x_n)$, which stores the results in the vector $y = (y_k)$.

$$y_k \leftarrow scale \cdot \sum_{n=0}^{N-1} x_n W_N^{k \cdot n}, \text{ for } k = 0, 1, \ldots, N\text{-}1$$

where:

$$W_N \equiv e^{sign \cdot j2\pi/N}$$

$$j \equiv \sqrt{-1}$$

sign = -1 for a *forward* transform or +1 for an *inverse* transform

Prototypes

```
vsip_fft_f *vsip_dfftx_create_f(
                    vsip_length N,
                    vsip_scalar_f scale,
                    vsip_fft_dir dir,
                    vsip_length ntimes,
                    vsip_alg_hint hint);
```

Where:

d is one of {`cc, cr, rc`} which corresponds to:
complex-to-complex, complex-to-real, and real-to-complex

x is one of {`op, ip`} which corresponds to:
out-of-place, and in-place

```
vsip_fft_f *vsip_ccfftip_create_f(
                    vsip_length N,
                    vsip_scalar_f scale,
                    vsip_fft_dir dir,
                    vsip_length ntimes,
                    vsip_alg_hint hint);
```

```
vsip_fft_f *vsip_ccfftop_create_f(
                    vsip_length N,
                    vsip_scalar_f scale,
                    vsip_fft_dir dir,
                    vsip_length ntimes,
                    vsip_alg_hint hint);
```

```
vsip_fft_f *vsip_rcfftop_create_f(
                    vsip_length N,
                    vsip_scalar_f scale,
                    vsip_length ntimes,
                    vsip_alg_hint hint);
vsip_fft_f *vsip_crfftop_create_f(
                    vsip_length N,
                    vsip_scalar_f scale,
                    vsip_length ntimes,
                    vsip_alg_hint hint);
```

Arguments

`N` Length of FFT

`scale` Real scale factor, typical values of scale are 1, $1/N$, and $1/\sqrt{N}$

`dir` Forward or Inverse FFT (note the argument is only for complex-to-complex)

```
typedef enum {
   VSIP_FFT_FWD = -1,
   VSIP_FFT_INV = 1
} vsip_fft_dir;
```

`ntimes` Estimate how many times the FFT object will be invoked. A value of zero is treated as semi-infinite.

`hint` Hint to help determine filtering approach.

```
typedef enum {
   VSIP_ALG_TIME = 0,
   VSIP_ALG_SPACE = 1,
   VSIP_ALG_NOISE = 2
} vsip_alg_hint;
```

Return Value

The return value is a pointer to a 1D FFT object, or *null* if it fails.

Restrictions

For complex-to-real and real-to-complex FFTs, *N* must be even.

Errors

The arguments must conform to the following:

1. *N*, the length of the FFT must be positive, non-zero. For complex-to-real and real-to-complex FFTs, *N* must be even.
2. *dir* must be a valid member of the `vsip_fft_dir` enumeration.
3. *hint* must be a valid member of the `vsip_alg_hint` enumeration.

Notes/References

For the complex-to-complex Fourier transform, the transform direction must be specified. For the real-to-complex Fourier transform, it is an implied *forward* transform. For the complex-to-real Fourier transform, it is an implied *inverse* transform.

FFT operations are supported for all values of *N*. However, the basic implementation requirement is for an $\mathbf{O}(N \log N)$ fast algorithm for the cases $N = 2^n$ or $N = 3 \cdot 2^n$, where *n* is a nonnegative integer. Some implementations may provide fast algorithms for other combinations of small prime factors and may even handle the general case of large prime factors

or prime sizes. When an implementation does not provide a fast algorithm, a DFT of $\mathbf{O}(N^2)$ or faster will be performed.

An implementation of this function may do nothing beyond save a copy of its calling parameters. It is suggested that this function be used to initialize (if necessary) a global twiddle table that all threads can read.

The parameter *ntimes* in conjunction with the *hint* is used (at the implementor's discretion) to pre-compute or optimize the FFT based on this information. If an FFT is to be called once or a few times, pre-computing may be not worthwhile. Pre-computing/optimization *may* include, but is not limited to, building a "twiddle table," allocating a workspace, building an algorithmic plan, and building an optimal FFT. Ideally the implementation uses a-priori time and space information with *ntimes* to optimize the FFT object to meet the user's *hint.*

Hints are just that. Implementations are free to ignore them and it is up to the implementor to determine the precise effect of the hints. However, the spirit of the hints is:

Minimize total FFT execution time.

Minimize the FFT total memory requirements.

Maximize numeric accuracy/stability (minimize numeric noise).

Only one hint may be specified.

Examples

See examples in `vsip_ccfftx_f`, `vsip_crfftx_f`, and `vsip_rcfftx_f`.

See Also

`vsip_dfftx_f`

vsip_fft_setwindow_*f* Set a FFT Window

Set a window (data taper) as part of the FFT object.

Functionality

The FFT object is created with a standard default boxcar window. This function will set a user defined window as part of the FFT object which will do the data windowing in conjunction with the FFT.

The FFT object is used to compute a complex to complex, real to complex, or complex-to-real Fast Fourier Transform (FFT) of vector $x = (x_n)$, which stores the results in the vector $y = (y_k)$.

When a window $t = (t_n)$ is set then

$$y_k \leftarrow scale \cdot \sum_{n=0}^{N-1} t_n x_n W_N^{k \cdot n}, \text{ for } k = 0, 1, \ldots, N\text{-}1$$

where:

$$W_N \equiv e^{sign \cdot j2\pi/N}$$

$$j \equiv \sqrt{-1}$$

sign = -1 for a *forward* transform or +1 for an *inverse* transform

Prototypes

```
void *vsip_fft_setwindow_f(
                     vsip_vview_f *t,
                     vsip_fft_f *fft);
```

Arguments

`t`	Vector of window values. This vector must be the same length as the fft input data set when the FFT object was created.
`fft`	The FFT object to modify with new window values.

Return Value

Restrictions

Error

1. All objects must be valid.
2. The window vector must be the proper length for the FFT object.

Notes/References

When the FFT object is created the default window is $t_n = 1$ for all n.

Examples

See Also

vsip_ccfftm*x*_*f* FFT Multiple Complex to Complex

Apply a multiple complex-to-complex Fast Fourier Transform (FFT)

Functionality

Computes a complex-to-complex Fast Fourier Transform (FFT) of the complex matrix $X = (x_{m,n})$, and stores the results in the complex matrix $Y = (y_{k,l})$.

A series of 1D complex vectors is stored in a matrix object in row major or column major order. Multiple 1D FFTs are then performed on the series of vectors along the specified row or column direction, as specified in the creation of the FFT object.

By rows:

$$y_{l,k} \leftarrow scale \cdot \sum_{p=0}^{P-1} x_{l,p} W_P^{p \cdot k}, \text{ for } k = 0, 1, \ldots, N\text{-}1 \text{ and } l = 0, 1, \ldots, M\text{-}1$$

or by columns:

$$y_{k,l} \leftarrow scale \cdot \sum_{n=0}^{N-1} x_{n,l} W_N^{k \cdot n}, \text{ for } k = 0, 1, \ldots, M\text{-}1 \text{ and } l = 0, 1, \ldots, N\text{-}1$$

where:

$W_P \equiv e^{sign \cdot j2\pi/P}$, for $P \in \{ M, N\}$

$j \equiv \sqrt{-1}$

sign = -1 for a *forward* transform or +1 for an *inverse* transform

Prototypes

Out-of-Place:

```
void vsip_ccfftmop_f (
                      const vsip_fftm_f *fft,
                      const vsip_cmview_f *X,
                      const vsip_cmview_f *Y);
```

In-Place:

```
void vsip_ccfftmip_f (
                      const vsip_fftm_f *fft,
                      const vsip_cmview_f *XY);
```

Arguments

`*fft`	Pointer to a 1D FFT object, created by `vsip_ccfftmx_create_f`
`*X`	View of input complex matrix of size *M* by *N*
`*Y`	View of output complex matrix of size *M* by *N*
`*XY`	View of input/output complex matrix of size *M* by *N*

Return Value

None.

Restrictions

Errors

The arguments must conform to the following:

1. All objects must be valid.
2. The FFT object must be a complex-to-complex multiple FFT object.
3. `vsip_ccfftmop_f` requires an out-of-place multiple FFT object, `vsip_ccfftmip_f` requires an in-place multiple FFT object.
4. The input and output must be a complex matrix views of size *M* by *N*, where *M* and *N* are obtained from the FFT object.
5. For an out-of-place FFT, the input and output matrix views must not overlap.

Notes/References

Examples

```
#include <stdio.h>
#include <vsip.h>

#define ROWS 64
#define N 16
#define ACMPLX(x) vsip_real_d(x),\
     (vsip_imag_d(x) < 0.0 ? "-i" : "+i"), fabs(vsip_imag_d(x))
#define SCMPLX "%g%s%g"

int main()
{
  int i;
  vsip_randstate *state;
  vsip_cmview_d   *xy;
  vsip_cvview_d   *xy_row;
  vsip_length     stride;
  vsip_cvview_d   *sum;
  vsip_fftm_d     *ccfftmip;
  vsip_length seed =0, num_procs=1, id=1;
  vsip_init((void *)0);

  /* Initialize Random Number Generator */
  state = vsip_randcreate(seed, num_procs, id, VSIP_PRNG);

  /* Row major, 64 (ROWS) of 16 point data vectors */
  xy = vsip_cmcreate_d(ROWS, 16, VSIP_ROW, VSIP_MEM_NONE);

  /* Bind xy_row view initially to row 0 of xy */
  xy_row = vsip_cmrowview_d(xy, 0);

   /* Stride between column elements of xy */
  stride = vsip_cmgetcolstride_d(xy);
  sum = vsip_cvcreate_d(N, VSIP_MEM_NONE);

  /* Create an in-place Cmplx->Cmplx Multiple  N-pt FFT */
  ccfftmip =
            vsip_ccfftmip_create_d(ROWS, N, 1.0, VSIP_FFT_FWD,
            VSIP_ROW, 1, VSIP_ALG_TIME);

  /* Initialize xy by rows with complex Gaussian noise N(0,1) */
  for (i=0; i<ROWS; i++) {
```

```
    vsip_cvputoffset_d(xy_row, i*stride); /* view of row i of xy */
    vsip_cvrand_d(state,xy_row);  /* Initialize row i of xy */
  }

  /* Compute an in-place Cmplx->Cmplx Multiple N-pt FFT using the
     ccfftmip object*/
  vsip_ccfftmip_d(ccfftmip, xy);

  /* Coherently sum the rows together (in the Freq domain) */
  vsip_cvputoffset_d(xy_row, 0);
  vsip_cvcopy_d_d(xy_row, sum); /* sum = row 0 of xy */
  for (i=1; i<ROWS; i++) {
    vsip_cvputoffset_d(xy_row, i*stride); /* view of row i of xy */
    vsip_cvadd_d(xy_row, sum, sum); /* sum += row i of xy */
  }

  /* Print it */
  printf("\nComplex Output Vector (Real, Imag)\n");
  for(i=0; i<N; i++)
    printf("%d:\t" SCMPLX "\n", i, ACMPLX(vsip_cvget_d(sum,i)));
  printf("\n");

  /* Destroy all the objects */
  vsip_fftm_destroy_d(ccfftmip);
  vsip_cvdestroy(xy_row);
  vsip_cvdestroy(sum);
  vsip_cmalldestroy_d(xy);
  vsip_randdestroy(state);
  vsip_finalize((void *)0);
  return(0);
}
```

See Also

`vsip_`*d*`fftm`***x***`_create_`***f***, and `vsip_fft`***n***`_destroy_`***f***

vsip_crfftmop_*f* FFT Multiple Complex to Real

Apply a multiple complex-to-real Fast Fourier Transform (FFT)

Functionality

Computes a complex-to-real (*inverse*) Fast Fourier Transform (FFT) of the complex matrix $X = (x_{m,n})$, and stores the results in the real matrix $Y = (y_{k,l})$.

A series of 1D complex vectors are stored in a matrix object in row major or column major order. Multiple 1D FFTs are then performed on the series along the unit stride (row or column) direction and the result is stored as a real matrix.

By rows:

$$y_{l,k} \leftarrow scale \cdot \sum_{p=0}^{P-1} x_{l,p} W_P^{p \cdot k}, \text{ for } k = 0, 1, \ldots, N\text{-}1 \text{ and } l = 0, 1, \ldots, M\text{-}1$$

or by columns:

$$y_{k,l} \leftarrow scale \cdot \sum_{n=0}^{N-1} x_{n,l} W_N^{k \cdot n}, \text{ for } k = 0, 1, \ldots, M\text{-}1 \text{ and } l = 0, 1, \ldots, N\text{-}1$$

where:

$$W_P \equiv e^{+j2\pi/P}, \text{ for } P \in \{M, N\}$$

$$j \equiv \sqrt{-1}$$

Prototypes

Out-of-Place:

```
void vsip_crfftmop_f (
                      const vsip_fftm_f *fft,
                      const vsip_cmview_f *X,
                      const vsip_mview_f *Y);
```

Arguments

`*fft`	Pointer to a 1D FFT object, created by `vsip_crffftmx_create_f`
`*X`	View of input complex matrix of size- $M/2 + 1$ by N, or M by $N/2 + 1$
`*Y`	View of output real matrix of size M by N

Return Value

None.

Restrictions

Only unit stride along the specified row or column FFT direction is supported.

The output length of the individual FFTs must be even.

Errors

The arguments must conform to the following:

1. All objects must be valid.
2. The FFT object must be a complex-to-real multiple FFT object.
3. `vsip_crfftmop_f` requires an out-of-place multiple FFT object.
4. The input must be a complex matrix view of length of size:
   - By Rows: M by $N/2 + 1$, N even.
   - By Column: $M/2 + 1$ by N, M even.

   where M and N are obtained from the FFT object.
5. The output must be a real matrix view of length of size M by N, where M and N are obtained from the FFT object.
6. For a multiple out-of-place FFT, the input and output matrix views must not overlap.
7. The input and output matrix views must be *unit-stride* in the transform direction.

Notes/References

The mathematical definition of the Fourier transform takes a sequence N complex values and transforms it to another sequence of N complex values. A complex-to-complex FFT routine, such as `vsip_ccfftm_f`, will take M (N) sets of N (M) complex input values, and produce N (M) complex output values.

Fourier transforms of length N (M), for the output matrix, Y, to be a real vector, the following identity on the input matrix, X, must be true:

$$x_{l,k} = x^*_{l,M-k}, \text{ for } \lfloor M/2 \rfloor < k < M$$

or

$$x_{k,l} = x^*_{N-k,l}, \text{ for } \lfloor N/2 \rfloor < k < N$$

And, in fact, the input values $x_{l,k}$ for $k > \lfloor M/2 \rfloor$ ($x_{k,l}$ for $k > \lfloor M/2 \rfloor$) are unnecessary; they can be inferred from the first half of the input.

Another implication is that $x_{l,0}$ ($x_{0,l}$), must be a real number. Also since N (M) is an even number, $x_{l,N/2}$ ($x_{M/2,l}$) will be real. The routine `vsip_crfftmop_f` assumes that these values are real. The imaginary part is ignored.

Thus, X is a complex matrix view of size $M/2 + 1$ by N, or M by $N/2 + 1$.

Examples

See Also

`vsip_dfftmx_create_f`, and `vsip_fftn_destroy_f`

vsip_rcfftmop_*f* FFT Multiple Real to Complex

Apply a multiple real-to-complex out of place Fast Fourier Transform (FFT)

Functionality

Computes a real-to-complex (*forward*) Fast Fourier Transform (FFT) of the real matrix $X = (x_{m,n})$, and stores the results in the complex matrix $Y = (y_{k,l})$.

A series of 1D real vectors are stored in a matrix object in row major or column major order. Multiple 1D FFTs are then performed on the series along the unit stride (row or column) direction and the result is stored as a complex matrix.

By rows:

$$y_{l,k} \leftarrow scale \cdot \sum_{p=0}^{P-1} x_{l,p} W_P^{p \cdot k}, \text{ for } k = 0, 1, \ldots, N\text{-}1 \text{ and } l = 0, 1, \ldots, M\text{-}1$$

or by columns:

$$y_{k,l} \leftarrow scale \cdot \sum_{n=0}^{N-1} x_{n,l} W_N^{k \cdot n}, \text{ for } k = 0, 1, \ldots, M\text{-}1 \text{ and } l = 0, 1, \ldots, N\text{-}1$$

where:

$$W_P \equiv e^{-j2\pi/P}, \text{ for } P \in \{ M, N\}$$

$$j \equiv \sqrt{-1}$$

Prototypes

Out-of-Place:

```
void vsip_rcfftmop_f (
                      const vsip_fftm_f *fft,
                      const vsip_mview_f *X,
                      const vsip_cmview_f *Y);
```

Arguments

`*fft`	Pointer to a 1D FFT object, created by `vsip_rcfftmop_create_f`
`*X`	View of input real matrix of size M by *N*
`*Y`	View of output complex matrix of size *M*/2 +1 by *N*, or *M* by *N*/2 +1

Return Value

None.

Restrictions

Only unit stride along the specified row or column FFT direction is supported. (FFT direction is specified in the create function.)

The input lengths of the individual FFTs must be even.

Errors

The arguments must conform to the following:

1. All objects must be valid.
2. The FFT object must be a real-to-complex multiple FFT object.
3. `vsip_rcfftmop_f` requires an out-of-place multiple FFT object.
4. The input must be a real matrix view of length of size *M* by *N*, where *M* and *N* are obtained from the FFT object.
5. The output must be a complex matrix view of length of size:
   - By Rows: *M* by *N*/2 +1, *N* even
   - By Column: *M*/2 +1 by *N*, *M* even

   where *M* and *N* are obtained from the FFT object.
6. For a multiple out-of-place FFT, the input and output matrix views must not overlap.
7. The input and output matrix must be *unit-stride* in the transform direction.

Notes/References

The mathematical definition of the Fourier transform takes a sequence of *N* complex values and transforms it to another sequence of *N* complex values. A complex-to-complex FFT routine, such as `vsip_ccfftm_f`, will take *M* (*N*) sets of *N* (*M*) complex input values, and produce *N* (*M*) complex output values.

The reason for having a separate real-to-complex FFT routine is efficiency. Because the input data are real, you can make use of this fact to save almost half of the computational work. For real input data, you have to compute only the first $\lfloor N/2 \rfloor+1$ ($\lfloor M/2 \rfloor+1$) complex output values, because the remaining values can be computed from the first half of the values by the simple formula:

$$y_{l,k} = y^*_{l,N-k}, \text{ for } \lfloor M/2 \rfloor < k < M$$

or

$$y_{k,l} = y^*_{N-k,l}, \text{ for } \lfloor N/2 \rfloor < k < N$$

For real input data, the output value, $y_{l,0}$ ($y_{0,l}$), will always be a real number. Also, since N(*M)* is an even number, $y_{l,N/2}$ ($y_{M/2,l}$) will be real.

Thus, in the real-to-complex routine, `vsip_rcfftmx_f`, *X* is a real matrix of size M by *N*, and *Y* is a complex matrix of size- $\lfloor M/2 \rfloor+1$ by *N*, or *M* by $\lfloor N/2 \rfloor+1$.

Examples

See Also

`vsip_dfftx_create_f`

vsip_*d*fftm*x*_create_*f* Create Multiple FFT Object

Create a 1D multiple FFT object.

Functionality

Creates a 1D multiple FFT object. The FFT object encapsulates the information on what type of FFT is to be computed and may at the implementor's discretion pre-compute or optimize the FFT based on this information.

This FFT object is used to compute multiple complex to complex, real to complex, or complex-to-real Fast Fourier Transforms (FFTs) of matrix $X = (x_{m,n})$, which stores the results in the matrix $Y = (y_{k,l})$.

The 1D data to be transformed is stored in a matrix object in row major or column major order. Multiple 1D FFTs are then performed along the specified row or column direction.

By rows:

$$y_{l,k} \leftarrow scale \cdot \sum_{p=0}^{P-1} x_{l,p} W_P^{p \cdot k}, \text{ for } k = 0, \ldots, N\text{-}1 \text{ and } l = 0, \ldots, M\text{-}1$$

or by columns:

$$y_{k,l} \leftarrow scale \cdot \sum_{n=0}^{N-1} x_{n,l} W_N^{k \cdot n}, \text{ for } k = 0, \ldots, M\text{-}1 \text{ and } l = 0, \ldots, N\text{-}1$$

where:

$W_P \equiv e^{sign \cdot j2\pi/P}$, for $P = M, N$

$j \equiv \sqrt{-1}$

sign = -1 for a *forward* transform or +1 for an *inverse* transform

Prototypes

```
vsip_fftm_f *vsip_dfftmx_create_f(
                    vsip_length M,
                    vsip_length N,
                    vsip_scalar_f scale,
                    vsip_fft_dir dir,
                    vsip_major major,
                    vsip_length ntimes,
                    vsip_alg_hint hint);
```

Where: ***d*** is one of {`cc`, `cr`, `rc`} which corresponds to:
complex-to-complex, complex-to-real, and real-to-complex

x is one of {`op`, `ip`} which corresponds to:
out-of-place, and in-place

```
vsip_fftm_f *vsip_ccfftmop_create_f(
                    vsip_length M,
                    vsip_length N,
                    vsip_scalar_f scale,
                    vsip_fft_dir dir,
                    vsip_major major,
                    vsip_length ntimes,
                    vsip_alg_hint hint);
```

```
vsip_fftm_f *vsip_ccfftmip_create_f(
                    vsip_length M,
                    vsip_length N,
                    vsip_scalar_f scale,
                    vsip_fft_dir dir,
                    vsip_major major,
                    vsip_length ntimes,
                    vsip_alg_hint hint);
vsip_fftm_f *vsip_rcfftmop_create_f(
                    vsip_length M,
                    vsip_length N,
                    vsip_scalar_f scale,
                    vsip_major major,
                    vsip_length ntimes,
                    vsip_alg_hint hint);
vsip_fftm_f *vsip_crfftmop_create_f(
                    vsip_length M,
                    vsip_length N,
                    vsip_scalar_f scale,
                    vsip_major major,
                    vsip_length ntimes,
                    vsip_alg_hint hint);
```

Arguments

`M` Length of column FFT or number of row ffts

`N` Length of row FFT or number of column ffts

`scale` Real scale factor, typical values of *scale* are: 1, $1/M$, $1/N$, $1/\sqrt{M}$, and $1/\sqrt{N}$

`dir` Forward or Inverse FFT (note the argument is only for complex-to-complex)

```
typedef enum {
  VSIP_FFT_FWD = -1,
  VSIP_FFT_INV = 1
} vsip_fft_dir;
```

`major` Direction of multiple FFT

```
typedef enum {
  VSIP_ROW = 0,          FFT each Row
  VSIP_COL = 1           FFT each Column
} vsip_major;
```

`ntimes` Estimate of how many times the FFT object will be invoked. A value of zero is treated as semi-infinite.

`hint` Hint to help determine algorithm

```
typedef enum {
  VSIP_ALG_TIME = 0,
  VSIP_ALG_SPACE = 1,
  VSIP_ALG_NOISE = 2
} vsip_alg_hint;
```

Return Value

The return value is a pointer to a 1D multiple FFT object, or *null* if it fails.

Restrictions

Real-to-complex and complex-to-real FFTs are restricted to unit stride along the specified, `major`, row or column FFT direction.

The length in the unit stride direction of these functions must be even.

Implementations may limit the maximum size, *M* and/or *N*.

Errors

The arguments must conform to the following:

1. *M,* and *N*, must be greater than zero.
2. *dir* must be a valid member of the `vsip_fft_dir` enumeration.
3. *major* must be a valid member of the `vsip_major` enumeration.
4. *hint* must be a valid member of the `vsip_alg_hint` enumeration.

Notes/References

For the complex-to-complex Fourier transform, the transform direction must be specified. For the real-to-complex Fourier transform it is an implied *forward* tranfsform. For the complex-to-real Fourier transform it is an implied *inverse* transform.

FFT operations are supported for all positive values of *N* and *M* (up to implementation dependent limits). However, the basic implementation requirement is for an $\mathbf{O}(N \log N)$ fast algorithm for the cases $N = 2^n$ or $N = 3 \cdot 2^n$, where *n* is a nonnegative integer. Some implementations may provide fast algorithms for other combinations of small prime factors and may even handle the general case of large prime factors or prime sizes. When an implementation does not provide a fast algorithm, a DFT of $\mathbf{O}(N^2)$ or faster will be performed.

For many computer systems, multiple 1D FFTs can be computed more efficiently with a special algorithm than simply calling a 1D FFT multiple times. This is particularly true for small FFTs. This method is sometimes called *stacked FFT* or *vector FFT*.

Performing a multiple 1D FFT on the data by rows and then by columns (or vice versa) is equivalent to performing a 2D FFT on the matrix, although it might be less efficient than the `vsip_dfft2dx_p`. Note that this would require two multiple FFT objects, one by rows, and one by columns.

FFTs performed along directions which have large strides between successive elements will have lower performance on many systems.

An implementation of this function may do nothing beyond save a copy of its calling parameters. It is suggested that this function be used to initialize (if necessary) a global twiddle table that all threads can read.

The parameter *ntimes* in conjunction with the *hint* is used (at the implementor's discretion) to pre-compute or optimize the FFT based on this information. This may include, but is not limited to, building a "twiddle table," allocating a workspace, building an algorithmic plan, and building an optimal FFT. Ideally, the implementation uses a-priori time and space information with *ntimes* to optimize the FFT object to meet the user's *hint.*

Hints are just that. Implementations are free to ignore them and it is up to the implementor to determine the precise effect of the hints. However, the spirit of the hints are:

1. Minimize total FFT execution time.
2. Minimize the FFT total memory requirements.
3. Maximize numeric accuracy/stability (minimize numeric noise).

Only one hint may be specified.

Examples

See Also

`vsip_dfftm`***x***`_`***f***, and `vsip_fft`***n***`_destroy_`***f***

vsip_fftm_setwindow_*f* Set a Multiple FFT Window

Set a window (data taper) as part of the multiple FFT object.

Functionality

The multiple FFT object is created with a standard default boxcar window. This function will set a user defined window as part of the FFT object which will do the data windowing in conjunction with the FFT.

The multiple FFT object is used to compute a complex to complex, real to complex, or complex-to-real Fast Fourier Transform (FFT) of matrix $X = (x_{m,n})$, which stores the results in the matrix $Y = (y_{k,l})$.

When a window $t = (t_n)$ is set then

By rows:

$$y_{l,k} \leftarrow scale \cdot \sum_{p=0}^{P-1} t_p x_{l,p} W_P^{p \cdot k}, \text{ for } k = 0, \ldots, N\text{-}1 \text{ and } l = 0, \ldots, M\text{-}1$$

or by columns:

$$y_{k,l} \leftarrow scale \cdot \sum_{n=0}^{N-1} t_n x_{n,l} W_N^{k \cdot n}, \text{ for } k = 0, \ldots, M\text{-}1 \text{ and } l = 0, \ldots, N\text{-}1$$

where:

$W_P \equiv e^{sign \cdot j2\pi/P}$, for $P = M, N$

$j \equiv \sqrt{-1}$

sign = -1 for a *forward* transform or +1 for an *inverse* transform

Prototypes

```
void *vsip_fftm_setwindow_f(
                      vsip_vview_f *t,
                      vsip_fftm_f *fft);
```

Arguments

`t`	Vector of window values. This vector must be the same length as the fft input data set when the multiple FFT object was created.
`fft`	The multiple FFT object to modify with new window values.

Return Value

Restrictions

Errors

The arguments must conform to the following:

1. All objects must be valid.
2. The window vector must be the proper length for the multiple FFT object.

Notes/References

When the multiple FFT object is created the default window is $t_n = 1$ for all n.

Examples

See Also

vsip_ccfft2d*x*_*f* 2D FFT Complex to Complex

Apply a complex-to-complex 2D Fast Fourier Transform (FFT)

Functionality

Compute a complex-to-complex 2D Fast Fourier Transform (FFT) of the complex M by N matrix $X = (x_{m,n})$, and store the results in the complex matrix $Y = (y_{u,v})$.

$$y_{p,q} \leftarrow scale \cdot \sum_{m=0}^{M-1}\sum_{n=0}^{N-1} x_{m,n} W_M^{m \cdot p} W_N^{q \cdot n}, \text{ for } u = 0, 1, \ldots, M\text{-}1; \text{ for } v = 0, \ldots, N\text{-}1;$$

where:

$W_P \equiv e^{sign \cdot j2\pi/P}$ for $P = M, N$

$j \equiv \sqrt{-1}$

$sign$ = -1 for a *forward* transform or +1 for an *inverse* transform

Prototypes

Out-of-Place:

```
void vsip_ccfft2dop_f (
                      const vsip_fft2d_f *fft,
                      const vsip_cmview_f *X,
                      const vsip_cmview_f *Y);
```

In-Place:

```
void vsip_ccfft2dip_f (
                      const vsip_fft2d_f *fft,
                      const vsip_cmview_f *XY);
```

Arguments

`*fft`	Pointer to a 2D FFT object, created by `vsip_ccfft2dx_create_f`
`*X`	View of input complex matrix of size M by N
`*Y`	View of output complex matrix of size M by N
`*XY`	View of output (input) complex matrix of size M by N

Return Value

None.

Restrictions

Errors

The arguments must conform to the following:

1. All objects must be valid.
2. The FFT object must be a complex-to-complex 2D FFT object.
3. `vsip_ccfft2dop_f` requires an out-of-place 2D FFT object, `vsip_ccfft2dip_f` requires an *in-place* 2D FFT object.
4. The input and output must be complex matrix views of size M by N, where M, and N, are obtained from the FFT object.
5. For an out-of-place *2D* FFT, the input and output matrix views must not overlap.

Notes/References

Examples

See Also

`vsip_`*d*`fft2d`*x*`_create_`*f*, and `vsip_fft`*n*`_destroy_`*f*

vsip_crfft2dop_*f* 2D FFT Complex to Real

Apply a complex-to-real 2D Fast Fourier Transform (FFT)

Functionality

Computes a complex-to-real (*inverse*) 2D Fast Fourier Transform (FFT) of the complex matrix $X = (x_{m,n})$, and stores the results in the real matrix $Y = (y_{u,v})$.

$$y_{p,q} \leftarrow scale \cdot \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} x_{m,n} W_M^{m \cdot p} W_N^{q \cdot n}, \text{ for } u = 0, 1, \ldots, \text{M-1; for } v = 0, \ldots, N\text{-1;}$$

where:

$W_P \equiv e^{+j2\pi/P}$ for $P = M, N$

$j \equiv \sqrt{-1}$

The 2D data to be transformed is stored in a matrix object in row major or column major order.

Prototypes

Out-of-Place:

```
void vsip_crfft2dop_f (
                    const vsip_fft2d_f *fft,
                    const vsip_cmview_f *X,
                    const vsip_mview_f *Y);
```

Arguments

`*fft` Pointer to a 2D FFT object, created by `vsip_crfft2dx_create_f`

`*X` View of input complex matrix of size

Unit Column Stride: M by $\lfloor N/2 \rfloor + 1$

Unit Row Stride: $\lfloor M/2 \rfloor + 1$ by N

`*Y` View of output real matrix of size M by N

Return Value

None.

Restrictions

Unit stride is required along one of the stride directions.

The length in the unit stride direction of the real matrix must be even.

Errors

The arguments must conform to the following:

1. All objects must be valid.
2. The FFT object must be a complex-to-real 2D FFT object.
3. `vsip_crfft2dop_f` requires an *out-of-place* 2D FFT object, `vsip_crfft2dip_f` requires an *in-place* multiple FFT object.
4. The input must be a complex matrix view of length of size:
   - By Rows: M by $N/2+1$, N even
   - By Column: $M/2+1$ by N, M even

   where M and N are obtained from the FFT object.
5. The output must be a real matrix view of length of size M by N, where M and N are obtained from the FFT object.
6. For an out-of-place 2D FFT, the input and output matrix views must not overlap.
7. The input and output matrix views must be unit stride in either the row or column axis direction.

Notes/References

Knowing the output is real, this routines takes the complex-to-complex FFT in the non-unit stride dimension, followed by the complex-to-real FFT in the unit-stride dimension.

In order for the output to be real, the input must have the two-dimensional conjugate symmetry:

$x_{m,n} = x^*_{M-m,N-n}$, for $\lfloor N/2 \rfloor < n < N$

And, in fact, the input values $x_{m,n}$ for $n > \lfloor N/2 \rfloor$ need are not necessary; they can be inferred from the first half of the input.

Thus, in the complex-to-real routine, X is a complex matrix of size M by $\lfloor N/2 \rfloor +1$ and Y is a real matrix of size M by N. Although only $\lfloor N/2 \rfloor +1$ input complex values are supplied, the size of the transform is still N in this case. Because, implicitly you are using the FFT formula for a sequence of length N.

Another implication is that $x_{l,0}$ ($x_{0,l}$), must be a real number. Also, since N (M) is an even number, $x_{l,N/2}$ ($x_{M/2,l}$) will be real. The routine `vsip_crfft2dx_f` assumes that these values are real. The imaginary part is ignored.

Thus, X is a complex matrix view of size $M/2+1$ by N, or M by $N/2+1$.

Examples

See Also

`vsip_dfft2dx_create_f`, and `vsip_fftn_destroy_f`

vsip_rcfft2dop_*f* 2D FFT Real to Complex

Apply a real-to-complex 2D Fast Fourier Transform (FFT)

Functionality

Computes a real-to-complex (*forward*) 2D Fast Fourier Transform (FFT) of the real M by N matrix $X = (x_{m,n})$, and stores the results in the complex matrix $Y = (y_{u,v})$.

$$y_{p,q} \leftarrow scale \cdot \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} x_{m,n} W_M^{m \cdot p} W_N^{q \cdot n}, \text{ for } u = 0, 1, \ldots, \text{M-1; for } v = 0, \ldots, N\text{-1;}$$

where:

$$W_P \equiv e^{-j2\pi/P} \text{ for } P = M, N$$

$$j \equiv \sqrt{-1}$$

The 2D data to be transformed is stored in a matrix object in row major or column major order.

Prototypes

Out-of-Place:

```
void vsip_rcfft2dop_f (
                    const vsip_fft2d_f *fft,
                    const vsip_cmview_f *X,
                    const vsip_mview_f *Y);
```

Arguments

`*fft`	Pointer to a 2D FFT object, created by `vsip_rcfft2dx_create_f`
`*X`	View of input real matrix of size M by N
`*Y`	View of output complex matrix of size Unit Column Stride: M by $N/2 + 1$ Unit Row Stride: $M/2 + 1$ by N

Return Value

None.

Restrictions

Unit stride is required along one of the stride directions.

The length in the unit stride direction for the real matrix must be even.

Errors

The arguments must conform to the following:

1. All objects must be valid.
2. The FFT object must be a complex-to-real 2D FFT object.
3. `vsip_crfft2dop_f` requires an out-of-place 2D FFT object.
4. The input must be a real matrix view of length of size M by N, where M and N are obtained from the FFT object.
5. The output must be a complex matrix view of length of size:
   - By Rows: M by $N/2 +1$, N even
   - By Column: $M/2 +1$ by N, M even

   where M and N are obtained from the FFT object.
6. For an out-of-place 2D FFT, the input and output matrix views must not overlap.
7. The input and output matrix views must be unit stride in either the row or column axis direction.

Notes/References

The mathematical definition of the Fourier transform takes a sequence of N complex values and transforms it to another sequence of N complex values. A complex-to-complex FFT routine, such as `vsip_ccfftip_f`, will take N complex input values, and produce N complex output values.

This routine computes a real-to-complex transform along the unit stride dimension, followed by the complex-to-complex transform in the other dimension. The reason for having a separate real-to-complex FFT routine is efficiency. Because the input data are real, you can make use of this fact to save almost half of the computational work The two-dimensional analog of the conjugate formula is as follows:

$y_{u,v} = y^*_{M-u,N-v}$, for $\lfloor N/2 \rfloor < v < N$

Thus, you have to compute only (slightly more than) half of the output values, namely:

$y_{u,v}$ for $0 \le u \le \lfloor M/2 \rfloor +1$ and $0 \le v < N$

For real input data, the output value, $y_{l,0}$ ($y_{0,l}$), will always be a real number. Also, since N (M) is an even number, $y_{l,N/2}$ ($y_{M/2,l}$) will be real.

Thus, in the real-to-complex routine, `vsip_rcfft2dx_f`, X is a real matrix of size M by N, and Y is a complex matrix of size $M/2 +1$ by N, or M by $N/2 +1$.

Examples

See Also

`vsip_dfft2dx_create_f`, and `vsip_fftn_destroy_f`

vsip_*d*fft2d*x*_create_*f* Create 2D FFT Object

Create a 2D FFT object.

Functionality

Creates a 2D FFT object. The FFT object encapsulates the information on what type of FFT is to be computed and may at the implementor's discretion pre-compute or optimize the FFT based on this information.

The 2D FFT object is used to compute complex to complex, real to complex, or complex-to-real Fast Fourier Transforms (FFTs) of matrix $X = (x_{m,n})$, which stores the results in the matrix $Y = (y_{u,v})$.

$$y_{u,v} \leftarrow scale \cdot \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} x_{m,n} W_M^{m \cdot u} W_N^{v \cdot n} \text{ , for } u = 0, 1, \ldots, \text{M-1; for } v = 0, \ldots, N\text{-1;}$$

where:

$$W_P \equiv e^{sign \cdot j2\pi/P} \text{, for } P = M, N$$

$$j \equiv \sqrt{-1}$$

sign = -1 for a *forward* transform or +1 for an *inverse* transform

Prototypes

```
vsip_fft2d_f *vsip_dfft2dx_create_f(
                    vsip_length M,
                    vsip_length N,
                    vsip_scalar_f scale,
                    vsip_fft_dir dir,
                    vsip_length ntimes,
                    vsip_alg_hint hint);
```

Where: ***d*** is one of {`cc, cr, rc`} which corresponds to:
complex-to-complex, complex-to-real, and real-to-complex

x is one of {`op, ip`} which corresponds to:
out-of-place, and in-place

```
vsip_fft2d_f *vsip_ccfft2dop_create_f(
                    vsip_length M,
                    vsip_length N,
                    vsip_scalar_f scale,
                    vsip_fft_dir dir,
                    vsip_length ntimes,
                    vsip_alg_hint hint);
vsip_fft2d_f *vsip_ccfft2dip_create_f(
                    vsip_length M,
                    vsip_length N,
                    vsip_scalar_f scale,
                    vsip_fft_dir dir,
                    vsip_length ntimes,
                    vsip_alg_hint hint);
```

```
vsip_fft2d_f *vsip_rcfft2dop_create_f(
                    vsip_length M,
                    vsip_length N,
                    vsip_scalar_f scale,
                    vsip_length ntimes,
                    vsip_alg_hint hint);
vsip_fft2d_f *vsip_crfft2dop_create_f(
                    vsip_length M,
                    vsip_length N,
                    vsip_scalar_f scale,
                    vsip_length ntimes,
                    vsip_alg_hint hint);
```

Arguments

`M` FFT size is *M* by *N*.

`N` FFT size is *M* by *N*.

`scale` Real scale factor, typical values of *scale* are: 1, 1/*M*, 1/*N*, $1/(M \cdot N), 1/\sqrt{M}$, $1/\sqrt{N}$,and $1/\sqrt{M \cdot N}$

`dir` Forward or Inverse FFT (note the argument is only for complex-to-complex)

```
typedef enum {
  VSIP_FFT_FWD = -1,
  VSIP_FFT_INV = 1
} vsip_fft_dir;
```

`ntimes` Approximately how many times the FFT object will be invoked. A value of zero is treated as semi-infinite.

`hint` Hint to help determine algorithm

```
typedef enum {
  VSIP_ALG_TIME = 0,
  VSIP_ALG_SPACE = 1,
  VSIP_ALG_NOISE = 2
} vsip_alg_hint;
```

Return Value

The return value is a pointer to a 2D FFT object, or *null* if it fails.

Restrictions

Real-to-complex and complex-to-real FFTs are restricted to views with unit stride along either the row or column direction.

The lengths in those directions in those functions must be even.

Implementations may limit the maximum size, *M* and *N*.

Errors

The arguments must conform to the following:

1. *M,* and *N*, must be positive, non-zero.
2. *dir* must be valid must be a valid member of the `vsip_fft_dir` enumeration.
3. *hint* must be a valid member of the `vsip_alg_hint` enumeration.

Notes/References

For the complex-to-complex Fourier transform, the transform direction must be specified. For the real-to-complex Fourier transform it is an implied *forward* transform. For the complex-to-real Fourier transform it is an implied *inverse* transform.

2D FFT operations are supported for all values of M and N. However, the basic implementation requirement is for an $\mathbf{O}(M{\cdot}N \log(M{\cdot}N))$ fast algorithm for the cases of $M = 2^m$ or $M = 3{\cdot}2^m$ and $N = 2^n$ or $N = 3{\cdot}2^n$, where m and n are nonnegative integers. Some implementations may provide fast algorithms for other combinations of small prime factors and may even handle the general case of large prime factors or prime sizes. When an implementation does not provide a fast algorithm, a DFT of $\mathbf{O}(N{\cdot}M^{2} + M{\cdot}N^{2})$ or faster will be performed.

An implementation of this function may do nothing beyond save a copy of its calling parameters. It is suggested that this function be used to initialize (if necessary) a global twiddle table that all threads can read.

The parameter *ntimes* in conjunction with the *hint* is used (at the implementor's discretion) to pre-compute or optimize the FFT based on this information. This *may* include, but is not limited to, building a "twiddle table," allocating a workspace, building an algorithmic plan, and building an optimal FFT. Ideally the implementation uses a-priori time and space information with *ntimes* to optimize the FFT object to meet the user's *hint.*

Hints are just that. Implementations are free to ignore them and it is up to the implementor to determine the precise effect of the hints. However, the spirit of the hints are:

1. Minimize total FFT execution time.
2. Minimize the FFT total memory requirements.
3. Maximize numeric accuracy/stability (minimize numeric noise).

Only one hint may be specified.

Examples

See Also

`vsip_dfft2dx_f`, and `vsip_fftn_destroy_f`

vsip_ccfft3d*x*_*f* 3D FFT Complex to Complex

Apply a complex-to-complex 3D Fast Fourier Transform (FFT)

Functionality

Computes a complex-to-complex 3D Fast Fourier Transform (FFT) of the complex P by M by N tensor $X = (x_{p,m,n})$, and stores the results in the complex tensor $Y = (y_{u,v,w})$.

$$y_{u,v,w} \leftarrow scale \cdot \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} \sum_{p=0}^{P-1} x_{m,n,p} \cdot W_M^{m \cdot u} \cdot W_N^{n \cdot v} \cdot W_N^{p \cdot w},$$

for u = 0, 1, …, M-1; for v = 0, …, N-1, for w = 0, …, P-1;

where:

$W_K \equiv e^{sign \cdot j2\pi/K}$, for $K = M, N, P$

$j \equiv \sqrt{-1}$

sign = -1 for a *forward* transform or +1 for an *inverse* transform

Prototypes

Out-of-Place:

```
void vsip_ccfft3dop_f (
                    const vsip_fft3d_f *fft,
                    const vsip_ctview_f *X,
                    const vsip_ctview_f *Y);
```

In-Place:

```
void vsip_ccfft3dip_f (
                    const vsip_fft3d_f *fft,
                    const vsip_ctview_f *XY);
```

Arguments

`*fft`	Pointer to a 3D FFT object, created by `vsip_ccfft3dx_create_f`
`*X`	View of input complex tensor of size P by M by N
`*Y`	View of output complex matrix of size P by M by N
`*XY`	View of input/output complex matrix of size P by M by N

Return Value

None.

Restrictions

Errors

The FFT, input and output objects must conform to the following:

1. All objects must be valid.
2. The FFT object must be a complex-to-complex 3D FFT object.
3. `vsip_ccfft3dop_f` requires an *out-of-place* 3D FFT object, `vsip_ccfft3dip_f` requires an *in-place* 3D FFT object.
4. The input and output must be a complex tensor views of size *P* by *M* by *N*, where *P*, *M*, and *N* are obtained from the FFT object.
5. For an out-of-place 3D FFT, the input and output tensor views must not overlap.

Notes/References

Examples

See Also

`vsip_dfft3dx_create_f`, and `vsip_fftn_destroy_f`

vsip_crfft3dop_*f* 3D FFT Complex to Real

Apply a complex-to-real 3D Fast Fourier Transform (FFT)

Functionality

Computes a complex-to-real (*inverse*) 3D Fast Fourier Transform (FFT) of the complex P tensor $X = (x_{p,m,n})$, and stores the results in the P by M by N real tensor $Y = (y_{u,v,w})$.

$$y_{u,v,w} \leftarrow scale \cdot \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} \sum_{p=0}^{P-1} x_{m,n,p} \cdot W_M^{m \cdot u} \cdot W_N^{n \cdot v} \cdot W_P^{p \cdot w},$$

for $u = 0, 1, \ldots, M-1$; for $v = 0, \ldots, N-1$, for $w = 0, \ldots, P-1$;

where:

$$W_K \equiv e^{+j2\pi/K}, \text{ for } K = M, N, P$$

$$j \equiv \sqrt{-1}$$

The 3D data to be transformed is stored in a tensor object in Y major, X major, or Z major order.

Prototypes

Out-of-Place:

```
void vsip_crfft3dop_f (
                    const vsip_fft3d_f *fft,
                    const vsip_ctview_f *X,
                    const vsip_tview_f *Y);
```

Arguments

`*fft` Pointer to a 3D FFT object, created by `vsip_crfft3dx_create_f`

`*X` View of input complex tensor of size

Unit X Stride: P by M by $\lfloor N/2 \rfloor + 1$

Unit Y Stride: P by $\lfloor M/2 \rfloor + 1$ by N

Unit Z Stride: $\lfloor P/2 \rfloor + 1$ by M by N

`*Y` View of output real tensor of P by M by N

Return Value

None.

Restrictions

Unit stride is required along one of the stride directions.

The length in the unit stride direction of the real tensor must be even.

Errors

The FFT, input, and output objects must conform to the following:

1. All objects must be valid.
2. The FFT object must be a complex-to-real 3D FFT object.
3. `vsip_crfft3dop_f` requires an out-of-place 3D FFT object.
4. The input must be a complex tensor view of size:
   - *P* by *M* by *N*/2 +1, *N* even
   - or *P* by *M*/2 +1 by *N*, *M* even
   - or *P*/2 +1 by *M* by *N*, *P* even

   where *P*, *M* and *N* are obtained from the FFT object.
5. The output must be a real tensor view of size *P* by *M* by *N*, where *P*, *M*, and *N* are obtained from the FFT object.
6. For an out-of-place 3D FFT, the input and output tensor views must not overlap.
7 The input and output matrix views must be unit stride in either the Z, Y or X axis direction

Notes/References

Knowing the output is real, this routines takes the complex-to-complex FFT in the non-unit stride dimensions, followed by the complex-to-real FFT in the unit stride dimension.

Examples

See Also

`vsip_dfft3dx_create_f`, and `vsip_fftn_destroy_f`

vsip_rcfft3dop_*f* 3D FFT Real to Complex

Apply a real-to-complex 3D Fast Fourier Transform (FFT)

Functionality

Computes a real-to-complex (*forward*) 3D Fast Fourier Transform (FFT) of the complex P by M by N tensor $X = (x_{p,m,n})$, and stores the results in the complex tensor $Y = (y_{u,v,w})$.

$$y_{u,v,w} \leftarrow scale \cdot \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} \sum_{p=0}^{P-1} x_{m,n,p} W_M^{m \cdot u} \cdot W_N^{n \cdot v} \cdot W_P^{p \cdot w},$$

for $u = 0, 1, \ldots, M-1$; for $v = 0, \ldots, N-1$, for $w = 0, \ldots, P-1$;

where:

$W_K \equiv e^{-j2\pi/K}$, for $K = M, N, P$

$j \equiv \sqrt{-1}$

The 3D data to be transformed is stored in a tensor object in X major, Y major, or Z major order.

Prototypes

Out-of-Place:

```
void vsip_rcfft3dop_f (
                    const vsip_fft3d_f *fft,
                    const vsip_tview_f *X,
                    const vsip_ctview_f *Y);
```

Arguments

`*fft`	Pointer to a 3D FFT object, created by `vsip_rcfft3dx_create_f`
`*X`	View of input real tensor of size P by M by N
`*Y`	View of output complex tensor of size Unit X Stride: P by M by $\lfloor N/2 \rfloor+1$ Unit Y: P by $\lfloor M/2 \rfloor+1$ by N Unit Z: $\lfloor P/2 \rfloor+1$ by M by N

Return Value

None.

Restrictions

Unit stride is required along one of the stride directions.

The length of the real tensor in the unit stride direction must be even.

Errors

The FFT, input, and output objects must conform to the following:

1. All objects must be valid.
2. The FFT object must be a complex-to-real 3D FFT object.
3. `vsip_rcfft3dop_f` requires an out-of-place 3D FFT object.
4. The input must be a real tensor view of size *P* by *M* by *N*, where *P*, *M*, and *N* are obtained from the FFT object.
5. The output must be a complex tensor view of size:
   - *P* by *M* by *N*/2 +1, *N* even
   - or *P* by *M*/2 +1 by *N*, *M* even
   - or *P*/2 +1 by *M* by *N*, *P* even

   where *P*, *M* and *N* are obtained from the FFT object.
6. For an out-of-place 3D FFT, the input and output tensor views must not overlap.
7. The input and output matrix views must be unit stride in either the Z, X or Y axis.

Notes/References

Knowing the input is real, this routines takes the real-to-complex FFT in the unit stride dimension, followed by the complex-to-complex FFT in the remaining dimensions.

Examples

See Also

`vsip_dfft3dx_create_f`, and `vsip_fftn_destroy_f`

vsip_*d*fft3d*x*_create_*f* Create 3D FFT Object

Create a 3D FFT object.

Functionality

Creates a 3D FFT object. The FFT object encapsulates the information on what type of FFT is to be computed and may at the implementor's discretion pre-compute or optimize the FFT based on this information.

This 3D FFT object is used to compute complex to complex, real to complex, or complex-to-real Fast Fourier Transforms (FFTs) of tensor $X = (x_{p,m,n})$, which stores the results in the tensor $Y = (y_{u,v,w})$.

$$y_{u,v,w} \leftarrow scale \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} \sum_{p=0}^{P-1} x_{m,n} W_M^{m \cdot n} W_N^{n \cdot v} W_P^{p \cdot w},$$

for $u = 0, 1, \ldots, M-1$; for $v = 0, \ldots, N-1$, for $w = 0, \ldots, P-1$;

where:

$W_K \equiv e^{sign \cdot j2\pi/K}$, for $K = M, N, P$

$j \equiv \sqrt{-1}$

sign = -1 for a *forward* transform or +1 for an *inverse* transform

Prototypes

```
vsip_fft3d_f *vsip_dfft3dx_create_f(
                     vsip_length P,
                     vsip_length M,
                     vsip_length N,
                     vsip_scalar_f scale,
                     vsip_fft_dir dir,
                     vsip_length ntimes,
                     vsip_alg_hint hint);
```

Where: ***d*** is one of {`cc, cr, rc`} which corresponds to:
complex-to-complex, complex-to-real, and real-to-complex

x is one of {`op, ip`} which corresponds to:
out-of-place, and in-place

```
vsip_fft3d_f *vsip_ccfft3dop_create_f(
                     vsip_length P,
                     vsip_length M,
                     vsip_length N,
                     vsip_scalar_f scale,
                     vsip_fft_dir dir,
                     vsip_length ntimes,
                     vsip_alg_hint hint);
vsip_fft3d_f *vsip_ccfft3dip_create_f(
                     vsip_length P,
                     vsip_length M,
                     vsip_length N,
                     vsip_scalar_f scale,
                     vsip_fft_dir dir,
                     vsip_length ntimes,
                     vsip_alg_hint hint);
```

```
vsip_fft3d_f *vsip_rcfft3dop_create_f(
                    vsip_length P,
                    vsip_length M,
                    vsip_length N,
                    vsip_scalar_f scale,
                    vsip_length ntimes,
                    vsip_alg_hint hint);
vsip_fft3d_f *vsip_crfft3dop_create_f(
                    vsip_length P,
                    vsip_length M,
                    vsip_length N,
                    vsip_scalar_f scale,
                    vsip_length ntimes,
                    vsip_alg_hint hint);
```

Arguments

`P` FFT size is *P* by *M* by *N*.

`M` FFT size is *P* by *M* by *N*.

`N` FFT size is *P* by *M* by *N*.

`scale` Real scale factor, typical values of *scale* are: 1, and $1/\sqrt{M \cdot N \cdot P}$

`dir` Forward or inverse FFT (note the argument is only for complex-to-complex)

```
typedef enum {
  VSIP_FFT_FWD = -1,
  VSIP_FFT_INV = 1
} vsip_fft_dir;
```

`ntimes` Estimate of how many times the FFT object will be invoked. A value of zero is treated as semi-infinite.

`hint` Hint to help determine algorithm

```
typedef enum {
  VSIP_ALG_TIME = 0,
  VSIP_ALG_SPACE = 1,
  VSIP_ALG_NOISE = 2
} vsip_alg_hint;
```

Return Value

The return value is a pointer to a 3D FFT object, or *null* if it fails.

Restrictions

Real-to-complex and complex-to-real FFTs are restricted to views with unit stride along one of the storage directions.

Implementations may limit the maximum size, *P*, *M*, and *N*.

Errors

The arguments must conform to the following:

1. *P*, *M,* and *N*, must be positive, non-zero.
2. *dir* must be a valid member of the `vsip_fft_dir` enumeration.
3. *hint* must be a valid member of the `vsip_alg_hint` enumeration.

Notes/References

For the complex-to-complex Fourier transform, the transform direction must be specified. For the real-to-complex Fourier transform it is an implied *forward* transform. For the complex-to-real Fourier transform it is an implied *inverse* transform.

3D FFT operations are supported for all values of M, N, and P. However, the basic implementation requirement is for an $\mathbf{O}(M{\cdot}N{\cdot}P \log(M{\cdot}N{\cdot}P))$ fast algorithm for the cases of $M = 2^m$ or $M = 3{\cdot}2^m$, $N = 2^n$ or $N = 3{\cdot}2^n$ and $P = 2^p$ or $P = 3{\cdot}2^p$, where m, n, and p are nonnegative integers. Some implementations may provide fast algorithms for other combinations of small prime factors and may even handle the general case of large prime factors or prime sizes. When an implementation does not provide a fast algorithm, a DFT of $\mathbf{O}(N{\cdot}N{\cdot}P + M{\cdot}Q{\cdot}P + M{\cdot}N{\cdot}P^2)$ or faster will be performed.

An implementation of this function may do nothing beyond save a copy of its calling parameters. It is suggested that this function be used to initialize (if necessary) a global twiddle table that all threads can read.

The parameter *ntimes* in conjunction with the *hint* is used (at the implementor's discretion) to pre-compute or optimize the FFT based on this information. This *may* include, but is not limited to, building a "twiddle table," allocating a workspace, building an algorithmic plan, and building an optimal FFT. Ideally the implementation uses a-priori time and space information with *ntimes* to optimize the FFT object to meet the user's *hint.*

Hints are just that. Implementations are free to ignore them and it is up to the implementor to determine the precise effect of the hints. However, the spirit of the hints are:

1. Minimize total FFT execution time.
2. Minimize the FFT total memory requirements.
3. Maximize numeric accuracy/stability (minimize numeric noise).

Only one hint may be specified.

Examples

See Also

`vsip_dfft3dx_f`, and `vsip_fftn_destroy_f`

vsip_fft*n*_destroy_*f* Destroy FFT Object

Destroy an FFT object.

Functionality

Destroys (free memory) an FFT object returning *null* on success, and non- *null* on failure.

Prototypes

```
int vsip_fftn_destroy_f(
                    vsip_fftn_f *fft);
```

Where: *n* is one of { , m, 2d, 3d} which corresponds to:
1D FFT, Multiple 1D FFTs, 2D FFT, and 3D FFT

```
int vsip_fft_destroy_f(
                    vsip_fft_f *fft);
int vsip_fftm_destroy_f(
                    vsip_fftm_f *fft);
int vsip_fft2d_destroy_f(
                    vsip_fft2d_f *fft);
int vsip_fft3d_destroy_f(
                    vsip_fft3d_f *fft);
```

Arguments

`*fft`	Pointer to an FFT object, created by `vsip_dfftx_create_f`, `vsip_dfftmx_create_f`, `vsip_dfft2dx_create_f`, or `vsip_dfft3dx_create_f`

Return Value

Returns zero on success and non-zero on failure.

Restrictions

Errors

The input object must conform to the following:

1. The FFT object must be valid. An argument of *null* is not an error.

Notes/References

An argument of *null* is not an error.

Note to Implementors: If the create of an FFT object creates/modifies shared information such as a twiddle table, then the shared object must maintain sufficient information to determine when it is possible to delete this shared information.

Examples

See Also

`vsip_dfftx_create_f`, `vsip_dfftmx_create_f`, `vsip_dfft2dx_create_f`, and `vsip_dfft3dx_create_f`

vsip_fft*n*_getattr_*f* FFT Get Attributes

Returns the attributes of an FFT object.

Functionality

Returns the attribute values of an FFT object in structure passed by reference.

The attributes are:

- input data size in elements of input type (*N*, *M* by *N*, or *P* by *M* by *N*)
- output data size in elements of output type (*N*, *M* by *N*, or *P* by *M* by *N*)
- FFT sign
- In-Place/Out-of-Place
- scale factor

```
typedef struct {
  vsip_scalar_t input;
  vsip_scalar_t output;
  vsip_fft_place place;
  vsip_scalar_f scale;
  vsip_fft_dir dir;
  vsip_major major; /* Only for fftm */
} vsip_fftn_attr_f;
```

Where: ***n*** is one of{ , m, 2d, 3d} which corresponds to:
1D FFT, Multiple 1D FFT, 2D FFT, and 3D FFT

t is one of {vi, mi, ti}which corresponds to:
1D, multiple 1D or 2D, and 3D

```
typedef enum {
  VSIP_FFT_FWD = -1,          Forward
  VSIP_FFT_INV= 1             Inverse
} vsip_fft_dir;

typedef enum {
  VSIP_FFT_IP = 0,            In-Place
  VSIP_FFT_OP = 1             Out-of-Place
} vsip_fft_place;

typedef enum {
  VSIP_ROW = 0,               FFT each Row
  VSIP_COL = 1,               FFT each Column
} vsip_major;
```

Prototypes

```
void vsip_fftn_getattr_f(
                      const vsip_fftn_f *fft,
                      vsip_fftn_attr_f *attr);
```

Below are the prototypes for the `vsip_fftn_getattr_f` functions and definitions for the `vsip_fftn_attr_f` structures.

```
void vsip_fft_getattr_f(
                      const vsip_fft_f *fft,
                      vsip_fft_attr_f *attr);
```

```
typedef struct {
   vsip_scalar_vi input;
   vsip_scalar_vi output;
   vsip_fft_place place;
   vsip_scalar_f scale;
   vsip_fft_dir dir;
} vsip_fft_attr_f;
```

```
void vsip_fftm_getattr_f(
                      const vsip_fftm_f *fft,
                      vsip_fftm_attr_f *attr);
```

```
typedef struct {
   vsip_scalar_mi input;
   vsip_scalar_mi output;
   vsip_fft_place place;
   vsip_scalar_f scale;
   vsip_fft_dir dir;
   vsip_major major;
} vsip_fftm_attr_f;
```

```
void vsip_fft2d_getattr_f(
                      const vsip_fft2d_f *fft,
                      vsip_fft2d_attr_f *attr);
```

```
typedef struct {
   vsip_scalar_mi input;
   vsip_scalar_mi output;
   vsip_fft_place place;
   vsip_scalar_f scale;
   vsip_fft_dir dir;
} vsip_fft2d_attr_f;
```

```
void vsip_fft3d_getattr_f(
                      const vsip_fft3d_f *fft,
                      vsip_fft3d_attr_f *attr);
```

```
typedef struct {
   vsip_scalar_ti input;
   vsip_scalar_ti output;
   vsip_fft_place place;
   vsip_scalar_f scale;
   vsip_fft_dir dir;
} vsip_fft3d_attr_f;
```

Arguments

`*fft`	Pointer to an FFT object, created by `vsip_dfftnx_create_f`
`*attr`	Pointer to attribute structure

Return Value

None.

Restrictions

Errors

The arguments must conform to the following:

1. The FFT object must be valid.
2. The output attribute pointer must be valid – non-*null*.

Notes/References

There is no attribute that explicitly indicates complex-to-complex, real-to-complex, or complex-to-real FFTs. This may be inferred from examining the input and output view sizes.

Examples

See Also

`vsip_dfftnx_create_f`, and `vsip_fftn_destroy_f`

Convolution/Correlation Functions

vsip_*d*conv1d_create_*f*	Create 1D Convolution Object
vsip_*d*conv1d_destroy_*f*	Destroy Conv1D Object
vsip_*d*conv1d_getattr_*f*	Conv1D Get Attributes
vsip_*d*convolve1d_*f*	1D Convolution
vsip_conv2d_create_*f*	Create 2D Convolution Object
vsip_conv2d_destroy_*f*	Destroy Conv2d Object
vsip_conv2d_getattr_*f*	Conv2d Get Attributes
vsip_convolve2d_*f*	2D Convolution
vsip_*d*corr1d_create_*f*	Create 1D Correlation Object
vsip_*d*corr1d_destroy_*f*	Destroy Corr1D Object
vsip_*d*corr1d_getattr_*f*	Corr1D Get Attributes
vsip_*d*correlate1d_*f*	1D Correlation
vsip_*d*corr2d_create_*f*	Create 2D Correlation Object
vsip_*d*corr2d_destroy_*f*	Destroy Corr2d Object
vsip_*d*corr2d_getattr_*f*	Corr2d Get Attributes
vsip_*d*correlate2d_*f*	2D Correlation

vsip_*d*conv1d_create_*f* Create 1D Convolution Object

Create a decimated 1D convolution filter object.

Functionality

Creates a decimated convolution filter object and returns a pointer to the object. The user specifies the *kernel* (filter order, symmetry, and filter coefficients), the *region of support*, and the integral output *decimation* factor.

A 1D convolution object is used to compute the convolution of a filter (*kernel*) vector *h,* of length *M*, with a data vector *x*, of length *N*, with an output decimation factor of *D*, producing the output vector, *y*.

Full:

Length $\lfloor (N + M - 2)/D \rfloor + 1$

$$y_n \leftarrow \sum_{k=0}^{M-1} h_k \cdot \langle x_{n \cdot D - k} \rangle, \quad \text{for } n = 0, \cdots, \left\lfloor \frac{N + M - 2}{D} \right\rfloor$$

Same Size:

Length $\lfloor (N-1)/D \rfloor + 1$

$$y_n \leftarrow \sum_{k=0}^{M-1} h_k \cdot \left\langle x_{n \cdot D + \lfloor M/2 \rfloor - k} \right\rangle, \quad \text{for } n = 0, \cdots, \left\lfloor \frac{N-1}{D} \right\rfloor$$

Minimum (non-zero-padded):

Length $\lfloor (N-1)/D \rfloor - \lfloor (M-1)/D \rfloor + 1$

$$y_n \leftarrow \sum_{k=0}^{M-1} h_k \cdot x_{n \cdot D + (M-1) - k}, \quad \text{for } n = 0, \cdots, \left\lfloor \frac{N-1}{D} \right\rfloor - \left\lfloor \frac{M-1}{D} \right\rfloor$$

Case $D = 1$:

Full:

Length $N + M - 1$

$$y_n \leftarrow \sum_{k=0}^{M-1} h_k \cdot \langle x_{n-k} \rangle, \quad \text{for } n = 0, \cdots, N + M - 2$$

Same Size:

Length N

$$y_n \leftarrow \sum_{k=0}^{M-1} h_k \cdot \left\langle x_{n + \lfloor M/2 \rfloor - k} \right\rangle, \quad \text{for } n = 0, \cdots, N - 1$$

Minimum (non-zero-padded):

Length $N - M + 1$

$$y_n \leftarrow \sum_{k=0}^{M-1} h_k \cdot x_{n + (M-1) - k}, \quad \text{for } n = 0, \cdots, N - M$$

Where:

$$\langle x_j \rangle \equiv \begin{cases} x_j, & 0 \le j < N \\ 0, & \text{otherwise} \end{cases}$$

The filter kernel can be even (conjugate) symmetric or non-symmetric. If it is (conjugate) symmetric, only the non-redundant values are specified.

Prototypes

```
vsip_conv1d_f *vsip_conv1d_create_f(
                    const vsip_vview_f *h,
                    vsip_symmetry symm,
                    vsip_length N,
                    vsip_length D,
                    vsip_support_region support,
                    vsip_length ntimes,
                    vsip_alg_hint hint);
vsip_rcconv1d_f *vsip_rcconv1d_create_f(
                    const vsip_vview_f *h,
                    vsip_symmetry symm,
                    vsip_length N,
                    vsip_length D,
                    vsip_support_region support,
                    vsip_length ntimes,
                    vsip_alg_hint hint);
vsip_cconv1d_f *vsip_cconv1d_create_f(
                    const vsip_cvview_f *h,
                    vsip_symmetry symm,
                    vsip_length N,
                    vsip_length D,
                    vsip_support_region support,
                    vsip_length ntimes,
                    vsip_alg_hint hint);
```

Arguments

`*h` Pointer to vector view object of filter coefficients,
non-symmetric: length M,
symmetric: length $\lceil M/2 \rceil$

`symm` Filter symmetry, including length symmetry {*even*, *odd*}

```
typedef enum {
  VSIP_NONSYM = 0,                Non-symmetric
  VSIP_SYM_EVEN_LEN_ODD = 1,      (Even) Symmetric, odd length
  VSIP_SYM_EVEN_LEN_EVEN = 2      (Even) Symmetric, even length
} vsip_symmetry;
```

`N` Length of input vector view

`D` Decimation factor (≥ 1)

`support` Output region of support (indicates which output points are computed).

```
typedef enum {
  VSIP_SUPPORT_FULL = 0,          N+M-1 outputs
  VSIP_SUPPORT_SAME = 1,          N outputs
  VSIP_SUPORT_MIN = 2             N-M+1 outputs
} vsip_support_region;
```

`ntimes` Estimate of how many convolution will be applied. A value of zero is treated as semi-infinite (*a lot of times*).

`hint` Hint to help determine algorithm approach.

```
typedef enum {
  VSIP_ALG_TIME = 0,
  VSIP_ALG_SPACE = 1,
  VSIP_ALG_NOISE = 2
} vsip_alg_hint;
```

Return Value

Returns a pointer to a 1D convolution filter object, or *null* if the create fails.

Restrictions

The filter length must be equal to or smaller than the data length, $M \leq N$.

Errors

The arguments must conform to the following:

1. *kernel* must be a pointer to a valid vector view object.
2. *symm* must be a valid member of the `vsip_symmetry` enumeration.
3. *N* must be greater than or equal to *M* (see *kernel* above).
4. *D* must be greater than zero.
5. *support* must be a valid member of the `vsip_support_region` enumeration.
6. *hint* must be a valid member of the `vsip_alg_hint` enumeration.

Notes/References

If all of the data are not available at one time, use the FIR filtering routines to filter the data in segments.

The decimation factor, *D*, is normally one for non-lowpass filters.

The parameter *ntimes* in conjunction with the *hint* is used (at the implementor's discretion) to pre-compute or optimize the convolution based on this information. Ideally the implementation uses a-priori time and space information with *ntimes* to optimize the convolution object to meet the user's *hint.*

Hints are just that. Implementations are free to ignore them and it is up to the implementor to determine the precise effect of the hints. However, the spirit of the hints are:

1. Minimize total convolution execution time.
2. Minimize the convolution total memory requirements.
3. Maximize numeric accuracy/stability (minimize numeric noise).

Only one hint may be specified.

Examples

See Also

vsip_*d*conv1d_destroy_*f* Destroy Conv1D Object

Destroy a 1D convolution object and deallocate any associated memory.

Functionality

Destroys a 1D convolution object.

Prototypes

```
vsip_length vsip_conv1d_destroy_f(
                    vsip_conv1d_f *conv1d);
vsip_length vsip_rcconv1d_destroy_f(
                    vsip_rcconv1d_f *conv1d);
vsip_length vsip_cconv1d_destroy_f(
                    vsip_cconv1d_f *conv1d);
```

Arguments

`*conv1d` Pointer to a 1D convolution object

Return Value

Returns zero on success and non-zero on failure.

Restrictions

Errors

The arguments must conform to the following:

1. The 1D convolution object must be valid. An argument of *null* is not an error.

Notes/References

An argument of *null* is not an error.

Examples

See Also

vsip_conv1d_getattr_*f* Conv1D Get Attributes

Returns the attributes for a 1D convolution object.

Functionality

Returns the attributes for a 1D convolution object.

The attributes are:

- filter kernel length
- filter kernel symmetry
- required length of input data (vector view)
- required length of output data (vector view)
- region of support for output
- output decimation factor

```
typedef struct {
   vsip_scalar_vi kernel_len;      M
   vsip_symmetry symm;
   vsip_scalar_vi data_len;        N
   vsip_support_region support,
   vsip_scalar_vi out_len;
   vsip_length decimation;         D
} vsip_conv1d_attr_f;
```

Where:

```
   typedef enum {
      VSIP_NONSYM = 0,                 Non-symmetric
      VSIP_SYM_EVEN_LEN_ODD = 1,       (Even) Symmetric, odd length
      VSIP_SYM_EVEN_LEN_EVEN = 2       (Even) Symmetric, odd length
   } vsip_symmetry;
   typedef enum {
      VSIP_SUPPORT_FULL = 0,           N+M-1 outputs
      VSIP_SUPPORT_SAME = 1,           N outputs
      VSIP_SUPORT_MIN = 2              N-M+1 outputs
   } vsip_support_region;
```

Prototypes

```
void vsip_conv1d_getattr_f(
                     const vsip_conv1d_f *conv1d,
                     vsip_conv1d_attr_f *attr);
void vsip_rcconv1d_getattr_f(
                     const vsip_rcconv1d_f *conv1d,
                     vsip_conv1d_attr_f *attr);
void vsip_cconv1d_getattr_f(
                     const vsip_cconv1d_f *conv1d,
                     vsip_conv1d_attr_f *attr);
```

Arguments

`*conv1d`	Pointer to a 1D convolution object
`*attr`	Pointer to a `vsip_conv1d_attr_f` structure

Return Value

None.

Restrictions

Errors

The arguments must conform to the following:

1. The 1D convolution object must be valid.
2. The output attribute pointer must be valid – non-*null*.

Notes/References

The length of the *kernel* is also known to as the *"filter order*."

Examples

See Also

vsip_*d*convolve1d_*f* 1D Convolution

Compute a decimated real one-dimensional (1D) convolution of two vectors.

Functionality

Uses a 1D convolution object is to compute the convolution of a filter (*kernel*) vector *h,* of length *M*, with a data vector *x*, of length *N*, with an output decimation factor of *D*, producing the output vector, *y*.

Full:

Length $\lfloor (N + M - 2)/D \rfloor + 1$

$$y_n \leftarrow \sum_{k=0}^{M-1} h_k \cdot \langle x_{n \cdot D - k} \rangle, \quad \text{for } n = 0, \cdots, \left\lfloor \frac{N + M - 2}{D} \right\rfloor$$

Same Size:

Length $\lfloor (N-1)/D \rfloor + 1$

$$y_n \leftarrow \sum_{k=0}^{M-1} h_k \cdot \langle x_{n \cdot D + \lfloor M/2 \rfloor - k} \rangle, \quad \text{for } n = 0, \cdots, \left\lfloor \frac{(N-1)}{D} \right\rfloor$$

Minimum (non-zero-padded):

Length $\lfloor (N-1)/D \rfloor - \lfloor (M-1)/D \rfloor + 1$

$$y_n \leftarrow \sum_{k=0}^{M-1} h_k \cdot x_{n \cdot D + (M-1) - k}, \quad \text{for } n = 0, \cdots, \left\lfloor \frac{N-1}{D} \right\rfloor - \left\lfloor \frac{M-1}{D} \right\rfloor$$

Case $D = 1$:

Full:

Length $N + M - 1$

$$y_n \leftarrow \sum_{k=0}^{M-1} h_k \cdot \langle x_{n-k} \rangle, \quad \text{for } n = 0, \cdots, N + M - 2$$

Same Size:

Length N

$$y_n \leftarrow \sum_{k=0}^{M-1} h_k \cdot \langle x_{n + \lfloor M/2 \rfloor - k} \rangle, \quad \text{for } n = 0, \cdots, N - 1$$

Minimum (non-zero-padded):

Length $N - M + 1$

$$y_n \leftarrow \sum_{k=0}^{M-1} h_k \cdot x_{n + (M-1) - k}, \quad \text{for } n = 0, \cdots, N - M$$

Where:

$$\langle x_j \rangle \equiv \begin{cases} x_j, & 0 \le j < N \\ 0, & \text{otherwise} \end{cases}$$

Prototypes

```
void vsip_convolve1d_f(
                      const vsip_conv1d_f *conv1d,
                      const vsip_vview_f *x,
                      const vsip_vview_f *y);
void vsip_rcconvolve1d_f(
                      const vsip_rcconv1d_f *conv1d,
                      const vsip_cvview_f *x,
                      const vsip_cvview_f *y);
void vsip_cconvolve1d_f(
                      const vsip_cconv1d_f *conv1d,
                      const vsip_cvview_f *x,
                      const vsip_cvview_f *y);
```

Arguments

`*conv1d`	Convolution filter object, (it includes the filter kernel, h)
`*x`	View of real input data vector of length N
`*y`	View of real destination vector of length Full: $\lfloor (N + M - 2)/D \rfloor + 1$, Same: $\lfloor (N-1)/D \rfloor + 1$, or Minimum: $\lfloor (N-1)/D \rfloor - \lfloor (M-1)/D \rfloor + 1$

Return Value

None.

Restrictions

Errors

The arguments must conform to the following:

- The 1D convolution object must be valid.
- The x input vector view must be of length N (conformant with the 1D convolution object).
- The y output vector view must be of length
  1. Full: $\lfloor (N + M - 2)/D \rfloor + 1$,
  2. Same: $\lfloor (N-1)/D \rfloor + 1$, or
  3. Minimum: $\lfloor (N-1)/D \rfloor - \lfloor (M-1)/D \rfloor + 1$
     (conformant with the 1D convolution object).
- The input x, and the output y, must not overlap.

Notes/References

The decimation factor, D, is normally one for non-lowpass filters.

Note `vsip_rcconvolve1d_f` convolves a real kernel with a complex vector.

If all of the data are not available at one time, use the FIR filtering routines to filter the data in segments.

Examples

See Also

vsip_conv2d_create_*f* Create 2D Convolution Object

Create a decimated 2D convolution filter object.

Functionality

Creates a decimated 2D convolution filter object and returns a pointer to the object. The user specifies the *kernel* (filter order, symmetry, and filter coefficients), the *region of support*, and the integral output *decimation* factor.

A 2D convolution object is used to compute the convolution of a real filter (*kernel*) matrix H, of size M by N, with a real data matrix X, of size P by Q, producing the output matrix Y. The filter must be smaller than or equal to the size of the data.

Let $H = (h_{i,j}) \in \Re^{M \times N}$, $X = (x_{i,j}) \in \Re^{P \times Q}$ denote the filter and data matrices.

Full:

Size $\lfloor (P + M - 2)/D \rfloor + 1$ by $\lfloor (Q + N - 2)/D \rfloor + 1$

$$y_{i,j} \leftarrow \sum_{u=0}^{M-1} \sum_{v=0}^{N-1} h_{u,v} \cdot \left\langle x_{i \cdot D - u, j \cdot D - v} \right\rangle, \quad \begin{array}{l} \text{for } i = 0, \cdots, \left\lfloor \dfrac{P + M - 2}{D} \right\rfloor \\ \text{for } j = 0, \cdots, \left\lfloor \dfrac{Q + N - 2}{D} \right\rfloor \end{array}$$

Same Size:

Size $\lfloor (P-1)/D \rfloor + 1$ by $\lfloor (Q-1)/D \rfloor + 1$

$$y_{i,j} \leftarrow \sum_{u=0}^{M-1} \sum_{v=0}^{N-1} h_{u,v} \cdot \left\langle x_{i \cdot D + \lfloor M/2 \rfloor - u, j \cdot D + \lfloor N/2 \rfloor - v} \right\rangle, \quad \begin{array}{l} \text{for } i = 0, \cdots, \left\lfloor \dfrac{P - 1}{D} \right\rfloor \\ \text{for } j = 0, \cdots, \left\lfloor \dfrac{Q - 1}{D} \right\rfloor \end{array}$$

Minimum (non-zero-padded):

Size $\lfloor (P-1)/D \rfloor - \lfloor (M-1)/D \rfloor + 1$ by $\lfloor (Q-1)/D \rfloor - \lfloor (N-1)/D \rfloor + 1$

$$y_{i,j} \leftarrow \sum_{u=0}^{M-1} \sum_{v=0}^{N-1} h_{u,v} \cdot x_{i \cdot D + (M-1) - u, j \cdot D + (N-1) - v}, \quad \begin{array}{l} \text{for } i = 0, \cdots, \left\lfloor \dfrac{P - 1}{D} \right\rfloor - \left\lfloor \dfrac{M - 1}{D} \right\rfloor \\ \text{for } j = 0, \cdots, \left\lfloor \dfrac{Q - 1}{D} \right\rfloor - \left\lfloor \dfrac{N - 1}{D} \right\rfloor \end{array}$$

Case $D = 1$:

Full:

Size $P + M - 1$ by $Q + N - 1$

$$y_{i,j} \leftarrow \sum_{u=0}^{M-1} \sum_{v=0}^{N-1} h_{u,v} \cdot \left\langle x_{i-u, j-v} \right\rangle, \quad \begin{array}{l} \text{for } i = 0, \cdots, P + M - 2 \\ \text{for } j = 0, \cdots, Q + N - 2 \end{array}$$

Same Size:

Size P by Q

$$y_{i,j} \leftarrow \sum_{u=0}^{M-1}\sum_{v=0}^{N-1} h_{u,v} \cdot \left\langle x_{(i+\lfloor M/2 \rfloor)-u,(j+\lfloor N/2 \rfloor)-v} \right\rangle, \quad \begin{array}{l} \text{for } i = 0,\cdots,P-1 \\ \text{for } j = 0,\cdots,Q-1 \end{array}$$

Minimum (non-zero-padded):

Size $P - M + 1$ by $Q - N + 1$

$$y_{i,j} \leftarrow \sum_{u=0}^{M-1}\sum_{v=0}^{N-1} h_{u,v} \cdot x_{i+(M-1)-u,\, j+(N-1)-v}, \quad \begin{array}{l} \text{for } i = 0,\cdots,P-M \\ \text{for } j = 0,\cdots,Q-N \end{array}$$

Where:

$$\left\langle x_{i,j} \right\rangle \equiv \begin{cases} x_{i,j} & 0 \le i < N_1 \text{ and } 0 \le j < N_2 \\ 0 & \text{otherwise} \end{cases}$$

The filter kernel can be even symmetric or non-symmetric. If it is symmetric, only the non-redundant values are specified.

Prototypes

```
vsip_conv2d_f *vsip_conv2d_create_f(
                    const vsip_mview_f *H,
                    vsip_symmetry symm,
                    vsip_length P,
                    vsip_length Q,
                    vsip_length decimate,
                    vsip_support_region support,
                    vsip_length ntimes,
                    vsip_alg_hint hint);
```

Arguments

`*H` Pointer to matrix view object of filter coefficients,
non-symmetric: size M by N,
(even) symmetric: size $\lceil M/2 \rceil$ by $\lceil N/2 \rceil$

`symm` Filter row and column symmetry including length symmetry {*even*, *odd*}. The symmetry applies to both directions of the filter.

```
typedef enum {
  VSIP_NONSYM = 0,                Non-symmetric
  VSIP_SYM_EVEN_LEN_ODD = 1,      (Even) Symmetric, odd length
  VSIP_SYM_EVEN_LEN_EVEN = 2      (Even) Symmetric, even length
} vsip_symmetry;
```

`P` Number of rows in input data matrix, X

`Q` Number of columns in input data matrix X

`D` Decimation factor (≥ 1)

`support` Output region of support (indicates which output points are computed).

```
typedef enum {
  VSIP_SUPPORT_FULL = 0,
  VSIP_SUPPORT_SAME = 1,
  VSIP_SUPORT_MIN = 2
} vsip_support_region;
```

`ntimes` Estimate of how many convolution will be applied. A value of zero is treated as semi-infinite (*a lot of times*).

`hint` Hint to help determine filtering approach.

```
typedef enum {
  VSIP_ALG_TIME = 0,
  VSIP_ALG_SPACE = 1,
  VSIP_ALG_NOISE = 2
} vsip_alg_hint;
```

Return Value

Returns a pointer to a 2D convolution filter object, or *null* if the create fails.

Restrictions

This implementation requires that $M \le P$, and $N \le Q$ (filter is smaller than the data).

Memory major order must be the same for kernel, data, and output.

The kernel, data, and output matrix views must be *unit-stride* in the major direction.

Errors

The arguments must conform to the following:

1. *H* must be a pointer to a valid matrix view object.
2. *symm* must be a valid member of the `vsip_symmetry` enumeration.
   - $M \le P$
   - $N \le Q$ (see *H* above).
3. *D* must be greater than zero.
4. *support* must be a valid member of the `vsip_support_region` enumeration.
5. *hint* must be a valid (a member of the `vsip_alg_hint` enumeration.
6. Memory major order must be the same for kernel, data, and output.
7. The kernel, data, and output matrix views must be *unit-stride* in the major direction.

Notes/References

Note: *symmetry, support*, and *decimation* attributes apply uniformly to all dimensions.

The parameter *ntimes* in conjunction with the *hint* is used (at the implementor's discretion) to pre-compute or optimize the 2D convolution based on this information. Ideally the implementation uses a-priori time and space information with *ntimes* to optimize the 2D convolution object to meet the user's *hint.*

Hints are just that. Implementations are free to ignore them and it is up to the implementor to determine the precise effect of the hints. However, the spirit of the hints are:

1. Minimize total convolution execution time.
2. Minimize the convolution total memory requirements.
3. Maximize numeric accuracy/stability (minimize numeric noise).

Only one hint may be specified.

Examples

See Also

vsip_conv2d_destroy_*f* Destroy Conv2d Object

Destroy a 2D convolution object.

Functionality

Destroys a 2D convolution object.

Prototypes

```
int vsip_conv2d_destroy_f(
                    vsip_conv2d_f *conv2d);
```

Arguments

`*conv2d` Pointer to a 2D convolution object

Return Value

Returns zero on success and non-zero on failure.

Restrictions

Errors

The arguments must conform to the following:

1. The 2D convolution object must be valid. An argument of *null* is not an error.

Notes/References

An argument of *null* is not an error.

Examples

See Also

vsip_conv2d_getattr_*f* Conv2d Get Attributes

Returns the attributes for a 2D convolution object.

Functionality

Returns the attributes for a 2D convolution object.

The attributes are:

- filter kernel size
- filter kernel symmetry
- required size of input data (matrix view)
- required size of output data (matrix view)
- region of support for output
- output decimation factor

```
typedef struct {
   vsip_scalar_mi kernel_size;          M by N
   vsip_symmetry symm;
   vsip_scalar_mi in_size;              P by Q
   vsip_support_region support,
   vsip_scalar_mi out_size;
   vsip_length decimation;              D
} vsip_conv2d_attr_f;
```

Where:

```
   typedef enum {
      VSIP_NONSYM = 0,                  Non-symmetric
      VSIP_SYM_EVEN_LEN_ODD = 1,        (Even) Symmetric, odd length
      VSIP_SYM_EVEN_LEN_EVEN = 2        (Even) Symmetric, even length
   } vsip_symmetry;
```

Prototypes

```
void vsip_conv2d_getattr_f(
                     const vsip_conv2d_f *conv2d,
                     vsip_conv2d_attr_f *attr);
```

Arguments

`*conv2d`	Pointer to a 2D convolution object
`*attr`	Pointer to a `vsip_conv2d_attr_f` structure

Return Value

None.

Restrictions

Errors

The arguments must conform to the following:

1. The 2D convolution object must be valid.
2. The output attribute pointer must be valid – non-*null*.

Notes/References

The size of the *kernel* is also referred to as the *"filter order*."

Examples

See Also

vsip_convolve2d_*f* 2D Convolution

Compute a decimated real two-dimensional (2D) convolution of two matrices.

Functionality

Uses a 2D convolution object to compute the convolution of a real filter (*kernel*) matrix *H*, size *M* by *N*, with a real data matrix *X*, size *P* by *Q*, producing the output matrix *Y*. The filter size must be less than or equal to the size of the data.

Let $H = (h_{i,j}) \in \Re^{M \times N}$, $X = (x_{i,j}) \in \Re^{P \times Q}$ denote the filter and data matrices.

Full:

Size $\lfloor (P + M - 2)/D \rfloor + 1$ by $\lfloor (Q + N - 2)/D \rfloor + 1$

$$y_{i,j} \leftarrow \sum_{u=0}^{M-1} \sum_{v=0}^{N-1} h_{u,v} \cdot \left\langle x_{i \cdot D - u, j \cdot D - v} \right\rangle, \quad \begin{array}{l} \text{for } i = 0, \cdots, \left\lfloor \dfrac{P + M - 2}{D} \right\rfloor \\ \text{for } j = 0, \cdots, \left\lfloor \dfrac{Q + N - 2}{D} \right\rfloor \end{array}$$

Same Size:

Length $\lfloor (P-1)/D \rfloor + 1$ by $\lfloor (Q-1)/D \rfloor + 1$

$$y_{i,j} \leftarrow \sum_{u=0}^{M-1} \sum_{v=0}^{N-1} h_{u,v} \cdot \left\langle x_{i \cdot D + \lfloor M/2 \rfloor - u, j \cdot D + \lfloor N/2 \rfloor - v} \right\rangle, \quad \begin{array}{l} \text{for } i = 0, \cdots, \left\lfloor \dfrac{P - 1}{D} \right\rfloor \\ \text{for } j = 0, \cdots, \left\lfloor \dfrac{Q - 1}{D} \right\rfloor \end{array}$$

Minimum (non-zero-padded):

Size $\lfloor (P-1)/D \rfloor - \lfloor (M-1)/D \rfloor + 1$ by $\lfloor (Q-1)/D \rfloor - \lfloor (N-1)/D \rfloor + 1$

$$y_{i,j} \leftarrow \sum_{u=0}^{M-1} \sum_{v=0}^{N-1} h_{u,v} \cdot x_{i \cdot D + (M-1) - u, j \cdot D + (N-1) - v}, \quad \begin{array}{l} \text{for } i = 0, \cdots, \left\lfloor \dfrac{P - 1}{D} \right\rfloor - \left\lfloor \dfrac{M - 1}{D} \right\rfloor \\ \text{for } j = 0, \cdots, \left\lfloor \dfrac{Q - 1}{D} \right\rfloor - \left\lfloor \dfrac{N - 1}{D} \right\rfloor \end{array}$$

Case $D = 1$:

Full:

Length $P + M - 1$ by $Q + N - 1$

$$y_{i,j} \leftarrow \sum_{u=0}^{M-1} \sum_{v=0}^{N-1} h_{u,v} \cdot \left\langle x_{i-u, j-v} \right\rangle, \quad \begin{array}{l} \text{for } i = 0, \cdots, P + M - 2 \\ \text{for } j = 0, \cdots, Q + N - 2 \end{array}$$

Same Size:

Length P by Q

$$y_{i,j} \leftarrow \sum_{u=0}^{M-1} \sum_{v=0}^{N-1} h_{u,v} \cdot \left\langle x_{(i + \lfloor M/2 \rfloor) - u, (j + \lfloor N/2 \rfloor) - v} \right\rangle, \quad \begin{array}{l} \text{for } i = 0, \cdots, P - 1 \\ \text{for } j = 0, \cdots, Q - 1 \end{array}$$

Minimum (non-zero-padded):

Length $P - M + 1$ by $Q - N + 1$

$$y_{i,j} \leftarrow \sum_{u=0}^{M-1}\sum_{v=0}^{N-1} h_{u,v} \cdot x_{i+(M-1)-u,\ j+(N-1)-v}, \quad \begin{array}{l} \text{for } i = 0,\cdots,\ P-M \\ \text{for } j = 0,\cdots,\ Q-N \end{array}$$

Where:

$$\langle x_{i,j} \rangle \equiv \begin{cases} x_{i,j} & 0 \le i < N_1 \text{ and } 0 \le j < N_2 \\ 0 & \text{otherwise} \end{cases}$$

Prototypes

```
void vsip_convolve2d_f(
                    const vsip_conv2d_f *conv2d,
                    const vsip_mview_f *X,
                    const vsip_mview_f *Y);
```

Arguments

`*conv2d`	Convolution filter object, (it includes the filter kernel, H)
`*X`	View of real input data matrix of size P by Q
`*Y`	View of real destination matrix of size Full: $\lfloor (P + M - 2)/D \rfloor + 1$ by $\lfloor (Q + N - 2)/D \rfloor + 1$ Same: $\lfloor (P-1)/D \rfloor + 1$ by $\lfloor (Q-1)/D \rfloor + 1$ Minimum: $\lfloor (P-1)/D \rfloor - \lfloor (M-1)/D \rfloor + 1$ by $\lfloor (Q-1)/D \rfloor - \lfloor (N-1)/D \rfloor + 1$

Return Value

None.

Restrictions

Memory major order must be the same for kernel, data, and output.

The kernel, data, and output matrix views are restricted to *unit-stride* in the major direction.

Errors

The objects must conform to the following:

1. The 2D convolution object must be valid.
2. The X input matrix view must be of size: P by Q (conformant with the 2D convolution object).
3. The Y output vector view must be of size:
   - Full: $\lfloor (P + M - 2)/D \rfloor + 1$ by $\lfloor (Q + N - 2)/D \rfloor + 1$
   - Same: $\lfloor (P-1)/D \rfloor + 1$ by $\lfloor (Q-1)/D \rfloor + 1$
   - Minimum: $\lfloor (P-1)/D \rfloor - \lfloor (M-1)/D \rfloor + 1$ by $\lfloor (Q-1)/D \rfloor - \lfloor (N-1)/D \rfloor + 1$ (conformant with the 2D convolution object).
4. The input X, and the output Y, must not overlap.
5. The input X, and the output Y, and *kernel* H, must have the same memory major order (*i.e.* if *row stride* < *column stride* for H, this must also hold for X, and Y).
6. The kernel, data, and output matrix views are restricted to *unit-stride* in the major direction

Notes/References

The decimation factor, D, is normally one for non-lowpass filters.

Examples

Codelet for a 3×3 Laplacian filter of a 480×640 image. Only the output values where the filter kernel does not extend beyond the image data are computed. The output is 378×638.

The kernel for a Laplacian filter is:

$$H \equiv \begin{bmatrix} \mathbf{.1667} & \mathbf{.6667} & .1667 \\ \mathbf{.6667} & \mathbf{-3.3333} & .6667 \\ .1667 & .6667 & .1667 \end{bmatrix}$$

Because the kernel is symmetric in both the row and column direction, only the **bold** coefficients need to be specified.

```
#define FOREVER 0
#define M 3
#define N 3
#define P 480
#define Q 640

float laplacian[M*N] = {.1667, 0.6667, .6667, -3.3333};
vsip_mview_f *flt_in = vsip_mcreate_f(P,Q,VSIP_MEM_NONE);
vsip_mview_f *flt_out = vsip_mcreate_f(P-M+1,Q-N+1,VSIP_MEM_NONE);
vsip_block_f *blk = vsip_blockbind_f(laplacian,M*N,VSIP_MEM_CONST);
vsip_mview_f *H = vsip_mbind_f(blk,0, M,N, N,1);
vsip_conv2d_f *filter;
vsip_blockadmit_f(blk,VSIP_TRUE);
filter = vsip_conv2d_create_f(H, VSIP_SYM_EVEN_LEN_ODD, P, Q, 1,
                              VSIP_SUPPORT_MIN, FOREVER,VSIP_ALG_TIME);
…
vsip_convolve2d_f(filter, flt_in, flt_out);
...
vsip_conv2d_destroy_f(filter);
vsip_mdestroy_f(H);
vsip_blockdestroy_f(blk);
vsip_mdestroy_f(flt_in);
vsip_mdestroy_f(flt_out);
…
```

See Also

vsip_*d*corr1d_create_*f* Create 1D Correlation Object

Create a 1D correlation object.

Functionality

Creates a (cross-)correlation object and returns a pointer to the object. The user specifies the length of the reference vector *r* and the data vector *x*.

A correlation object is used to compute the correlation of a reference vector *r* of length *M*, with a data vector *x* of length *N*, producing the output vector *y*.

Full:

Length $N + M - 1$

$$\hat{y}_n \leftarrow \sum_{k=0}^{M-1} r_k \cdot \left\langle x^*_{n+k-(M-1)} \right\rangle, \quad \text{for } n = 0, \cdots, N + M - 2$$

$$y_n \leftarrow \hat{y}_n \cdot \begin{cases} 1/(n+1), & \text{for } 0 \le n < M - 1 \\ 1/M, & \text{for } M - 1 \le n < N \\ 1/(N + M - 1 - n), & \text{for } N \le n < N + M - 1 \end{cases}$$

Same Size:

Length N

$$\hat{y}_n \leftarrow \sum_{k=0}^{M-1} r_k \cdot \left\langle x^*_{n+k-\lfloor M/2 \rfloor} \right\rangle, \quad \text{for } n = 0, \cdots, N - 1$$

$$y_n \leftarrow \hat{y}_n \cdot \begin{cases} 1/(n + \lceil M/2 \rceil), & \text{for} \quad 0 \le n < \lfloor M/2 \rfloor \\ 1/M, & \text{for} \quad \lfloor M/2 \rfloor \le n < N - \lfloor M/2 \rfloor \\ 1/(N + \lfloor M/2 \rfloor - n), & \text{for} \quad N - \lfloor M/2 \rfloor \le n < N \end{cases}$$

Minimum (non-zero-padded):

Length $N - M + 1$

$$\hat{y}_n \leftarrow \sum_{k=0}^{M-1} r_k \cdot x^*_{n+k}, \quad \text{for } n = 0, \cdots, N - M$$

$$y_n \leftarrow \hat{y}_n / M$$

Where:

$$\left\langle x_j \right\rangle \equiv \begin{cases} x_j & \text{where } 0 \le j < N \\ 0 & \text{otherwise} \end{cases}$$

The values $\hat{y}_j$ are the biased correlation estimates while y_j are unbiased estimates. (The unbiased estimates are scaled by the number of terms in the summation for each lag where $\left\langle x_j \right\rangle$ is not defined to be zero.)

Prototypes

Real vectors

```
vsip_corr1d_f *vsip_corr1d_create_f(
                    vsip_length M,
                    vsip_length N,
                    vsip_support_region support,
                    vsip_length ntimes,
                    vsip_alg_hint hint);
```

Complex vectors

```
vsip_ccorr1d_f *vsip_ccorr1d_create_f(
                    vsip_length M,
                    vsip_length N,
                    vsip_support_region support,
                    vsip_length ntimes,
                    vsip_alg_hint hint);
```

Arguments

`M` Length of input reference vector view, **r**

`N` Length of input data vector view, **x**

`support` Output region of support (indicates which output points are computed).

```
typedef enum {
  VSIP_SUPPORT_FULL = 0,
  VSIP_SUPPORT_SAME = 1,
  VSIP_SUPORT_MIN = 2
} vsip_support_region;
```

`ntimes` Estimate of how many correlations will be applied. A value of zero is treated as semi-infinite (*a lot of times*).

`hint` Hint to help determine algorithm approach.

```
typedef enum {
  VSIP_ALG_TIME = 0,
  VSIP_ALG_SPACE = 1,
  VSIP_ALG_NOISE = 2
} vsip_alg_hint;
```

Return Value

Returns a pointer to an 1D correlation object, or *null* if the create fails.

Restrictions

The reference length must be less than or equal to the data length, $M \leq N$.

Errors

The arguments must conform to the following:

1. *M* must be greater than zero.
   N must be greater than zero and greater than or equal to *M*.
2. *symm* must be a valid member of the `vsip_symmetry` enumeration.
3. *support* must be a valid member of the `vsip_support_region` enumeration.
4. *hint* must be a valid member of the `vsip_alg_hint` enumeration.

Notes/References

The parameter *ntimes* in conjunction with the *hint* is used (at the implementor's discretion) to pre-compute or optimize the correlation based on this information. Ideally, the implementation uses a-priori time and space information with *ntimes* to optimize the correlation object to meet the user's *hint.*

Hints are just that. Implementations are free to ignore them and it is up to the implementor to determine the precise effect of the hints. However, the spirit of the hints are:

1. Minimize total convolution execution time.
2. Minimize the convolution total memory requirements.
3. Maximize numeric accuracy/stability (minimize numeric noise).

Only one hint may be specified.

If all of the data are not available at one time, use the FIR filtering routines to filter the data in segments. Specify the FIR kernel as the reverse indexed clone of the reference data.

Examples

See Also

vsip_*d*corr1d_destroy_*f* Destroy Corr1D Object

Destroy a 1D correlation object.

Functionality

Destroys a 1D correlation object. Returns zero on failure.

Prototypes

Real vectors

```
int vsip_corr1d_destroy_f(
                    vsip_corr1d_f *corr1d);
```

Complex vectors

```
int vsip_ccorr1d_destroy_f(
                    vsip_ccorr1d_f *corr1d);
```

Arguments

`*corr1d` Pointer to a 1D correlation object

Return Value

Returns zero on success and non-zero on failure.

Restrictions

Errors

The arguments must conform to the following:

1. The 1D correlation object must be valid. An argument of *null* is not an error.

Notes/References

An argument of *null* is not an error.

Examples

See Also

vsip_*d*corr1d_getattr_*f* Corr1D Get Attributes

Returns the attributes for a 1D correlation object.

Functionality

Returns the attributes for a 1D (cross-) correlation object.

The attributes are:

- required length of reference data (vector view)
- required length of input data (vector view)
- output region of support
- required length of output data (vector view)

Prototypes

Real vectors

```
void vsip_corr1d_getattr_f(
                      const vsip_corr1d_f *corr1d,
                      vsip_corr1d_attr_f *attr);
typedef struct {
   vsip_scalar_vi ref_len;
   vsip_scalar_vi data_len;
   vsip_support_region support;
   vsip_scalar_vi lag_len;
} vsip_corr1d_attr_f;
```

Complex vectors

```
void vsip_ccorr1d_getattr_f(
                      const vsip_ccorr1d_f *corr1d,
                      vsip_ccorr1d_attr_f *attr);
typedef struct {
   vsip_scalar_vi ref_len;
   vsip_scalar_vi data_len;
   vsip_support_region support;
   vsip_scalar_vi lag_len;
} vsip_ccorr1d_attr_f;
```

Arguments

`*corr1d`	Pointer to a 1D correlation object
`*attr`	Pointer to a `vsip_dcorr1d_attr_f` structure

Return Value

None.

Restrictions

Errors

The arguments must conform to the following:

1. The 1D correlation object must be valid.
2. The output attribute pointer must be valid – non-*null*.

Notes/References

Examples

See Also

vsip_*d*correlate1d_*f* 1D Correlation

Compute a real one-dimensional (1D) correlation of two vectors.

Functionality

Computes the (cross-) correlation of a reference vector r of length M, with a data vector x of length N, producing the output vector y.

Full:

Length $N + M - 1$

$$\hat{y}_n \leftarrow \sum_{k=0}^{M-1} r_k \cdot \left\langle x^*_{n+k-(M-1)} \right\rangle, \quad \text{for } n = 0, \cdots, N + M - 2$$

$$y_n \leftarrow \hat{y}_n \cdot \begin{cases} 1/(n+1), & \text{for } 0 \le n < M-1 \\ 1/M, & \text{for } M-1 \le n < N \\ 1/(N+M-1-n), & \text{for } N \le n < N+M-1 \end{cases}$$

Same Size:

Length N

$$\hat{y}_n \leftarrow \sum_{k=0}^{M-1} r_k \cdot \left\langle x^*_{n+k-\lfloor M/2 \rfloor} \right\rangle, \quad \text{for } n = 0, \cdots, N-1$$

$$y_n \leftarrow \hat{y}_n \cdot \begin{cases} 1/(n+\lceil M/2 \rceil), & \text{for } \quad 0 \le n < \lfloor M/2 \rfloor \\ 1/M, & \text{for } \quad \lfloor M/2 \rfloor \le n < N - \lfloor M/2 \rfloor \\ 1/(N+\lfloor M/2 \rfloor - n), & \text{for } \quad N - \lfloor M/2 \rfloor \le n < N \end{cases}$$

Minimum (non-zero-padded):

Length $N - M + 1$

$$\hat{y}_n \leftarrow \sum_{k=0}^{M-1} r_k \cdot x^*_{n+k}, \quad \text{for } n = 0, \cdots, N-M$$

$$y_n \leftarrow \hat{y}_n / M$$

Where:

$$\left\langle x_j \right\rangle \equiv \begin{cases} x_j & \text{where } 0 \le j < N \\ 0 & \text{otherwise} \end{cases}$$

The values $\hat{y}_j$ are the biased correlation estimates while y_j are unbiased estimates. (The unbiased estimates are scaled by the number of terms in the summation for each lag where $\left\langle x_j \right\rangle$ is not defined to be zero.)

Prototypes

Real vectors

```
void vsip_correlate1d_f(
                      const vsip_corr1d_f *corr1d,
                      vsip_bias bias,
                      const vsip_vview_f *ref,
                      const vsip_vview_f *x,
                      const vsip_vview_f *y);
```

Complex vectors

```
void vsip_ccorrelate1d_f(
                    const vsip_corr1d_f *corr1d,
                    vsip_bias bias,
                    const vsip_cvview_f *ref,
                    const vsip_cvview_f *x,
                    const vsip_cvview_f *y);
```

Where:

```
typedef enum {
  VSIP_BIASED = 0,
  VSIP_UNBIASED = 1
} vsip_bias;
```

Arguments

`*corr1d`	Pointer to correlation object
`bias`	Select biased or unbiased correlation estimate
`*ref`	View of real (complex) reference data vector of length M
`*x`	View of real (complex) input data vector of length N
`*y`	View of real (complex) lag vector of length ($N + M - 1$), N, or ($N - M + 1$) (full, same, or minimum)

Return Value

None.

Restrictions

The reference length must be less than or equal to the data length, $M \leq N$.

Errors

The arguments must conform to the following:

1. The 1D correlation object must be valid.
2. *bias* must be a valid member of the `vsip_bias` enumeration.
3. The **r** reference input vector view must be of length M (conformant with the 1D correlation object).
4. The **x** data input vector view must be of length N (conformant with the 1D correlation object).
5. The **y** output vector view must be of length:
   - Full: $N + M - 1$,
   - Same: N, or
   - Minimum: $N - M + 1$
     (conformant with the 1D correlation object).
6. The output **y** cannot overlap either of the input vector views, **r** or **x.**

Notes/References

Examples

If all of the data are not available at one time, use the FIR filtering routines to filter the data in segments. Specify the FIR kernel as the reverse indexed clone of the reference data.

See Also

vsip_*d*corr2d_create_*f* Create 2D Correlation Object

Create a 2D correlation object.

Functionality

Creates a 2D correlation object and returns a pointer to the object. The user specifies the size of the reference matrix *R* and the data matrix *X*.

Compute the correlation of a reference matrix *R* with a data matrix *X*, producing the output matrix *Y*. This implementation requires that $M \le P$, and $N \le Q$ (reference size is less than or equal to the data size).

Let $R = (r_{i,j}) \in \Re^{M \times N}$, $X = (x_{i,j}) \in \Re^{P \times Q}$ denote the reference and data matrices.

Full:

Size $P + M - 1$ by $Q + N - 1$

$$\hat{y}_{i,j} \leftarrow \sum_{u=0}^{M-1} \sum_{v=0}^{N-1} r_{u,v} \cdot \left\langle x^*_{u+i-(M-1),v+j-(N-1)} \right\rangle, \quad \begin{array}{l} \text{for } i = 0, \cdots, P + M - 2 \\ \text{for } j = 0, \cdots, Q + N - 2 \end{array}$$

$$y_{i,j} \leftarrow \hat{y}_{i,j} \cdot \begin{cases} 1/(i+1), & \text{for } 0 \le i < M - 1 \\ 1/M, & \text{for } M - 1 \le i < P \\ 1/(P + M - 1 - i), & \text{for } P \le i < P + M - 1 \end{cases} \cdot \begin{cases} 1/(j+1), & \text{for } 0 \le j < N - 1 \\ 1/N, & \text{for } N - 1 \le j < Q \\ 1/(Q + N - 1 - j), & \text{for } Q \le j < Q + N - 1 \end{cases}$$

Same Size:

Size P by Q

$$\hat{y}_{i,j} \leftarrow \sum_{u=0}^{M-1} \sum_{v=0}^{N-1} r_{u,v} \cdot \left\langle x^*_{u+i-\lfloor M_1/2 \rfloor,v+j-\lfloor N/2 \rfloor} \right\rangle, \quad \begin{array}{l} \text{for } i = 0, \cdots, P - 1 \\ \text{for } j = 0, \cdots, Q - 1 \end{array}$$

$$y_{i,j} \leftarrow \hat{y}_{i,j} \cdot \begin{cases} 1/(i + \lceil M/2 \rceil), & \text{for } 0 \le i < \lceil M/2 \rceil \\ 1/M, & \text{for } \lfloor M/2 \rfloor \le i < P - \lfloor M/2 \rfloor \\ 1/(P - 1 + \lceil M/2 \rceil - i), & \text{for } P \le i < P + M - 1 \end{cases} \cdot \begin{cases} 1/(j + \lceil N/2 \rceil), & \text{for } 0 \le j < \lceil N/2 \rceil \\ 1/N, & \text{for } \lfloor N/2 \rfloor \le j < Q - \lfloor N/2 \rfloor \\ 1/(Q - 1 + \lceil N/2 \rceil - j), & \text{for } Q - \lfloor N/2 \rfloor \le j < Q \end{cases}$$

Minimum (Non-zero-padded):

Size $P - M + 1$ by $Q - N + 1$

$$\hat{y}_{i,j} \leftarrow \sum_{u=0}^{M-1} \sum_{v=0}^{N-1} h_{u,v} \cdot x^*_{u+i,v+j}, \quad \begin{array}{l} \text{for } i = 0, \cdots, P - M \\ \text{for } j = 0, \cdots, Q - N \end{array}$$

$$y_{i,j} \leftarrow \hat{y}_{i,j} / (M \cdot N), \quad \text{for } 0 \le i < P - M - 1; \quad \text{for } 0 \le j < Q - N - 1$$

Where:

$$\left\langle x_{i,j} \right\rangle \equiv \begin{cases} x_{i,j} & \text{where } 0 \le i < P \text{ and } 0 \le j < Q \\ 0 & \text{otherwise} \end{cases}$$

The values $\hat{y}_{i,j}$ are the biased correlation estimates while $y_{i,j}$ are unbiased estimates. (The unbiased estimates are scaled by the number of terms in the summation for each lag where $\left\langle x_{i,j} \right\rangle$ is not defined to be zero.)

Prototypes

Real matrices

```
vsip_corr2d_f *vsip_corr2d_create_f(
                    vsip_length M,
                    vsip_length N,
                    vsip_length P,
                    vsip_length Q,
                    vsip_support_region support,
                    vsip_length ntimes,
                    vsip_alg_hint hint);
```

Complex matrices

```
vsip_ccorr2d_f *vsip_ccorr2d_create_f(
                    vsip_length M,
                    vsip_length N,
                    vsip_length P,
                    vsip_length Q,
                    vsip_support_region support,
                    vsip_length ntimes,
                    vsip_alg_hint hint);
```

Arguments

`M` Reference matrix view size is *M* rows by *N* columns

`N` Reference matrix view size is *M* rows by *N* columns

`P` Data matrix view size is *P* rows by *Q* columns

`Q` Data matrix view size is *P* rows by *Q* columns

`support` Output region of support (indicates which output points are computed).

```
typedef enum {
   VSIP_SUPPORT_FULL = 0,
   VSIP_SUPPORT_SAME = 1,
   VSIP_SUPORT_MIN = 2
} vsip_support_region;
```

`ntimes` Estimate of how many correlation will be applied. A value of zero is treated as semi-infinite (*a lot of times*).

`hint` Hint to help determine filtering approach.

```
typedef enum {
   VSIP_ALG_TIME = 0,
   VSIP_ALG_SPACE = 1,
   VSIP_ALG_NOISE = 2
} vsip_alg_hint;
```

Return Value

Returns a pointer to an 2D correlation object, or *null* if the create fails.

Restrictions

Errors

The arguments must conform to the following:

1. *P* must be greater than or equal to *M*
   Q must be greater than or equal to *N* (see *H* above).
2. *support* must be a valid member of the `vsip_support_region` enumeration.
3. *hint* must be a valid member of the `vsip_alg_hint` enumeration.

Notes/References

Note:*, support* attributes apply uniformly to all dimensions.

The parameter *ntimes* in conjunction with the *hint* is used (at the implementor's discretion) to pre-compute or optimize the 2D correlation based on this information. Ideally the implementation uses a-priori time and space information with *ntimes* to optimize the 2D correlation object to meet the user's *hint.*

Hints are just that. Implementations are free to ignore them and it is up to the implementor to determine the precise effect of the hints. However, the spirit of the hints are:

1. Minimize total convolution execution time.
2. Minimize the convolution total memory requirements.
3. Maximize numeric accuracy/stability (minimize numeric noise).

Only one hint may be specified.

Examples

See Also

vsip_*d*corr2d_destroy_*f* Destroy Corr2d Object

Destroy a 2D correlation object.

Functionality

Destroys a 2D correlation object.

Prototypes

Real matrices

```
int vsip_corr2d_destroy_f(
                    vsip_corr2d_f *corr2d);
```

Complex matrices

```
int vsip_ccorr2d_destroy_f(
                    vsip_ccorr2d_f *corr2d);
```

Arguments

`*corr2d` Pointer to a 2D correlation object

Return Value

Returns zero on success and non-zero on failure.

Restrictions

Errors

The arguments must conform to the following:

1. The 2D correlation object must be valid. An argument of *null* is not an error.

Notes/References

An argument of *null* is not an error.

Examples

See Also

vsip_*d*corr2d_getattr_*f* Corr2d Get Attributes

Returns the attributes for a 2D correlation object.

Functionality

Returns the attributes for a 2D (cross-) correlation object.

The attributes are:

- required size of reference data (matrix view)
- required size of input data (matrix view)
- output region of support
- required size of output lags (matrix view)

Prototypes

Real matrices

```
void vsip_corr2d_getattr_f(
                      const vsip_corr2d_f *corr2d,
                      vsip_corr2d_attr_f *attr);
typedef struct vsip_ {
   vsip_scalar_mi ref_size;
   vsip_scalar_mi data_size;
   vsip_support_region support,
   vsip_scalar_mi lag_size;
} vsip_corr2d_attr_f;
```

Complex matrices

```
void vsip_ccorr2d_getattr_f(
                      const vsip_ccorr2d_f *corr2d,
                      vsip_ccorr1d_attr_f *attr);
typedef struct vsip_ {
   vsip_scalar_mi ref_size;
   vsip_scalar_mi data_size;
   vsip_support_region support,
   vsip_scalar_mi lag_size;
} vsip_ccorr2d_attr_f;
```

Arguments

`*corr2d`	Pointer to a 2D correlation object
`*attr`	Pointer to a `vsip_dcorr2d_attr_f` structure

Return Value

None.

Restrictions

Errors

The arguments must conform to the following:

1. The 2D correlation object must be valid.
2. The output attribute pointer must be valid – non-*null*.

Notes/References

Examples

See Also

vsip_*d*correlate2d_*f* 2D Correlation

Compute a two-dimensional correlation of two matrices.

Functionality

Computes the (cross-) correlation of a reference matrix R with a data matrix X, producing the output matrix Y. This implementation requires that $M \le P$, and $N \le Q$ (reference size is less than or equal to the data size).

Let $R = \left(r_{i,j}\right) \in \Re^{M \times N}$, $X = \left(x_{i,j}\right) \in \Re^{P \times Q}$ denote the reference and data matrices.

Full:

Size $P + M - 1$ by $Q + N - 1$

$$\hat{y}_{i,j} \leftarrow \sum_{u=0}^{M-1} \sum_{v=0}^{N-1} r_{u,v} \cdot \left\langle x^*_{u+i-(M-1),v+j-(N-1)} \right\rangle, \quad \begin{array}{l} \text{for } i = 0, \cdots, P + M - 2 \\ \text{for } j = 0, \cdots, Q + N - 2 \end{array}$$

$$y_{i,j} \leftarrow \hat{y}_{i,j} \cdot \begin{cases} 1/(i+1), & \text{for } 0 \le i < M - 1 \\ 1/M, & \text{for } M - 1 \le i < P \\ 1/(P + M - 1 - i), & \text{for } P \le i < P + M - 1 \end{cases}$$

$$\cdot \begin{cases} 1/(j+1), & \text{for } 0 \le j < N - 1 \\ 1/N, & \text{for } N - 1 \le j < Q \\ 1/(Q + N - 1 - j), & \text{for } Q \le j < Q + N - 1 \end{cases}$$

Same Size:

Size P by Q

$$\hat{y}_{i,j} \leftarrow \sum_{u=0}^{M-1} \sum_{v=0}^{N-1} r_{u,v} \cdot \left\langle x^*_{u+i-\lfloor M_1/2 \rfloor,v+j-\lfloor N/2 \rfloor} \right\rangle, \quad \begin{array}{l} \text{for } i = 0, \cdots, P - 1 \\ \text{for } j = 0, \cdots, Q - 1 \end{array}$$

$$y_{i,j} \leftarrow \hat{y}_{i,j} \cdot \begin{cases} 1/(i + \lceil M/2 \rceil), & \text{for } 0 \le i < \lceil M/2 \rceil \\ 1/M, & \text{for } \lfloor M/2 \rfloor \le i < P - \lfloor M/2 \rfloor \\ 1/(P - 1 + \lceil M/2 \rceil - i), & \text{for } P \le i < P + M - 1 \end{cases}$$

$$\cdot \begin{cases} 1/(j + \lceil N/2 \rceil), & \text{for } 0 \le j < \lceil N/2 \rceil \\ 1/N, & \text{for } \lfloor N/2 \rfloor \le j < Q - \lfloor N/2 \rfloor \\ 1/(Q - 1 + \lceil N/2 \rceil - j), & \text{for } Q - \lfloor N/2 \rfloor \le j < Q \end{cases}$$

Minimum (Non-zero-padded):

Size $P - M + 1$ by $Q - N + 1$

$$\hat{y}_{i,j} \leftarrow \sum_{u=0}^{M-1} \sum_{v=0}^{N-1} h_{u,v} \cdot x^*_{u+i,v+j}, \quad \begin{array}{l} \text{for } i = 0, \cdots, P - M \\ \text{for } j = 0, \cdots, Q - N \end{array}$$

$$y_{i,j} \leftarrow \hat{y}_{i,j} / (M \cdot N), \quad \text{for } 0 \le i < P - M - 1; \quad \text{for } 0 \le j < Q - N - 1$$

Where:

$$\left\langle x_{i,j} \right\rangle \equiv \begin{cases} x_{i,j} & \text{where } 0 \le i < P \text{ and } 0 \le j < Q \\ 0 & \text{otherwise} \end{cases}$$

The values $\hat{y}_{i,j}$ are the biased correlation estimates while $y_{i,j}$ are unbiased estimates. (The unbiased estimates are scaled by the number of terms in the summation for each lag where $\left\langle x_{i,j} \right\rangle$ is not defined to be zero.)

Prototypes

Real matrices

```
void vsip_correlate2d_f(
                      const vsip_corr2d_f *corr2d,
                      vsip_bias bias,
                      const vsip_mview_f *R,
                      const vsip_mview_f *X,
                      const vsip_mview_f *Y);
```

Complex matrices

```
void vsip_ccorrelate2d_f(
                      const vsip_ccorr2d_f *corr2d,
                      vsip_bias bias,
                      const vsip_cmview_f *R,
                      const vsip_cmview_f *X,
                      const vsip_cmview_f *Y);
```

Where:

```
typedef enum {
   VSIP_BIASED = 0,
   VSIP_UNBIASED = 1
} vsip_bias;
```

Arguments

`*corr2d`	Correlation object
`bias`	Biased or unbiased correlation estimate
`*R`	View of real (complex) reference data matrix of size M by N
`*X`	View of real (complex) input data matrix of size P by Q
`*Y`	View of real (complex) lag matrix of size Full: $P+M-1$ by $Q+N-1$ Same Size: P by Q Minimum: $P-M+1$ by $Q-N+1$

Return Value

None.

Restrictions

The *reference* **R**, *data* **X**, and *lag* output **Y** must all have the same memory major ordering. The matrix views must be *unit-stride* in the major direction.

Errors

The arguments must conform to the following:

1. The 2D correlation object must be valid.
2. *bias* must be a valid member of the `vsip_bias` enumeration.
3. The **R** reference input matrix view must be of size M by N (conformant with the 2D correlation object).
4. The **X** data input matrix view must be of size P by Q (conformant with the 2D correlation object).
5. The **Y** output matrix view must be of size
   - Full: $P+M-1$ by $Q+N-1$
   - Same Size: P by Q
   - Minimum: $P-M+1$ by $Q-N+1$ (conformant with the 2D correlation object).
6. The output **Y** cannot overlap either the reference **R** or the data **X**.
7. The inputs **R**, and **X**, and the output **Y**, must have the same memory major order (*i.e.* if *row stride* < *column stride* for **R**, this must also hold for **X**, and **Y**).
8. The input and output matrix views must be all be *unit-stride* in the major direction.

Notes/References

Examples

See Also

Window Functions

vsip_**vcreate_blackman**_*f*	Create Blackman Window
vsip_**vcreate_cheby**_*f*	Create Chebyshev Window
vsip_**vcreate_hanning**_*f*	Create Hanning Window
vsip_**vcreate_kaiser**_*f*	Create Kaiser Window

vsip_vcreate_blackman_*f* Create Blackman Window

Create a vector with Blackman window weights.

Functionality

Creates a vector initialized with a Blackman window of length *N,*

$$wind_k \leftarrow 0.42 - 0.5 \cdot \cos\left(\frac{2 \cdot \pi \cdot k}{N-1}\right) + 0.08 \cdot \cos\left(\frac{4 \cdot \pi \cdot k}{N-1}\right), \text{ for } k = 0, \ldots, N\text{-}1$$

and returns a pointer to a real vector view object, or *null* if the create fails.

Prototypes

```
vsip_vview_f *vsip_vcreate_blackman_f (
                    vsip_length N,
                    vsip_memory_hint hint);
```

Arguments

`N` Length of window vector

`hint` Memory type hints (Typically `VSIP_MEM_CONST`)

```
typedef enum {
  VSIP_MEM_NONE = 0,                 No hint
  VSIP_MEM_RDONLY = 1,               Read Only
  VSIP_MEM_CONST = 2,                Constant
  VSIP_MEM_SHARED = 3,               Shared
  VSIP_MEM_SHARED_RDONLY = 4,        Shared, Read Only
  VSIP_MEM_SHARED_CONST = 5          Shared, Constant
} vsip_memory_hint;
```

Return Value

Returns a pointer to the vector view object, or *null* on failure.

Restrictions

Errors

The arguments must conform to the following:

1. $N > 1$.
2. *hint* must be a valid member of the `vsip_memory_hint` enumeration.

Notes/References

Examples

```
#define N 256
…
vsip_vview_sp blackman = vsip_vcreate_blackman_sp(N,
          VSIP_MEM_CONST);
vsip_vview_sp x = vsip_vcreate(N, VSIP_MEM_NONE);

/* window the data in x */
vsip_vmul_sp(x,blackman,x);
vsip_valldestroy_sp(blackman); vsip_valldestroy_sp(x);
```

See Also

vsip_vcreate_cheby_*f* Create Chebyshev Window

Create a vector with Dolph-Chebyshev window weights.

Functionality

Creates a real vector initialized with a Dolph-Chebyshev window of length *N,*

$$\delta_p = 10^{-ripple/20}$$

$$\tau_p = \left(1+\delta_P\right)/\delta_p$$

$$\delta_f = \frac{1}{\pi}\cos^{-1}\left[\cosh\left(\cosh^{-1}\left(\tau_p\right)/\left(N-1\right)\right)^{-1}\right]$$

$$src_0 = \left(3-\cos(2\cdot\pi\cdot\delta_f)\right)/\left(1+\cos(2\cdot\pi\cdot\delta_f)\right)$$

$$x_k = \frac{x_0+1}{2}\cdot\cos(2\pi\cdot k/N)+\frac{x_0-1}{2}$$

N is an odd integer:

$$w_k = \begin{cases} \delta_p\cosh\left(\frac{N-1}{2}\cdot\cosh^{-1}\left(x_k\right)\right) & \left|x_k\right|>1, \quad 0\le k<N \\ \delta_p\cos\left(\frac{N-1}{2}\cdot\cos^{-1}\left(x_k\right)\right) & \left|x_k\right|\le 1, \quad 0\le k<N \end{cases}$$

N is an even integer:

$$w_k = \begin{cases} \delta_p\cosh\left(\frac{N-1}{2}\cdot\cosh^{-1}\left(x_k\right)\right)\cdot e^{-j\pi\cdot k/N} & \left|x_k\right|>1, \quad 0\le k\le\left\lfloor\frac{N}{2}\right\rfloor \\ -\delta_p\cosh\left(\frac{N-1}{2}\cdot\cosh^{-1}\left(x_k\right)\right)\cdot e^{-j\pi\cdot k/N} & \left|x_k\right|>1, \quad \left\lfloor\frac{N}{2}\right\rfloor<k\le N-1 \\ \delta_p\cos\left(\frac{N-1}{2}\cdot\cos^{-1}\left(x_k\right)\right)\cdot e^{-j\pi\cdot k/N} & \left|x_k\right|\le 1, \quad 0\le k\le\left\lfloor\frac{N}{2}\right\rfloor \\ -\delta_p\cos\left(\frac{N-1}{2}\cdot\cos^{-1}\left(x_k\right)\right)\cdot e^{-j\pi\cdot k/N} & \left|x_k\right|\le 1, \quad \left\lfloor\frac{N}{2}\right\rfloor<k\le N-1 \end{cases}$$

$$w_n = \sum_{k=0}^{N-1} W_k e^{-\frac{j2\pi\cdot k\cdot n}{N}} \qquad \text{[FFT of } W_k\text{]}$$

$$wind_k \leftarrow \begin{cases} \mathrm{Re}\left(\hat{w}_{k+\lfloor N/2\rfloor}/\hat{w}_0\right) & 0\le k<\left\lfloor\frac{N}{2}\right\rfloor \\ \mathrm{Re}\left(\hat{w}_{k-\lfloor N/2\rfloor}/\hat{w}_0\right) & \left\lfloor\frac{N}{2}\right\rfloor\le k<N \end{cases} \qquad \text{[Frequency swap of } w_n\text{]}$$

This function returns a pointer to a real vector view object, or *null* if the create fails.

Prototypes

```
vsip_vview_f *vsip_vcreate_cheby_f (
                    vsip_length N,
                    vsip_scalar_f ripple,
                    vsip_memory_hint hint);
```

Arguments

`N`	Length of window vector
`ripple`	Window ripple in db (side-lobes are *ripple* db below the main-lobe)
`hint`	Memory type hints (Typically `VSIP_MEM_CONST`)

```
typedef enum {
  VSIP_MEM_NONE = 0,                  No hint
  VSIP_MEM_RDONLY = 1,                Read Only
  VSIP_MEM_CONST = 2,                 Constant
  VSIP_MEM_SHARED = 3,                Shared
  VSIP_MEM_SHARED_RDONLY = 4,         Shared, Read Only
  VSIP_MEM_SHARED_CONST = 5           Shared, Constant
} vsip_memory_hint;
```

Return Value

Returns a pointer to the vector view object, or *null* on failure.

Restrictions

Errors

The arguments must conform to the following:

1. *N* must be greater than zero.
2. *hint* must be a valid member of the `vsip_memory_hint` enumeration.

Notes/References

Examples

```
#define N 256
#define RIPPLE 60.0 /* dB */
…
vsip_vview_sp cheby = vsip_vcreate_cheby_sp(N, RIPPLE,
          VSIP_MEM_CONST);
vsip_vview_sp x = vsip_vcreate_sp(N, VSIP_MEM_NONE);
…
/* window the data in x */
vsip_vmul_sp(x,cheby,x);
…
vsip_valldestroy_sp(cheby); vsip_valldestroy_sp(x);
```

See Also

References for the Dolph-Chebyshev window are in the introduction to this chapter.

vsip_vcreate_hanning_*f* Create Hanning Window

Create a vector with Hanning window weights.

Functionality

Creates a vector initialized with a Hanning window of length *N,*

$$wind_k \leftarrow \frac{1}{2} \cdot \left(1 - \cos\left(\frac{2\pi \cdot (k+1)}{N+1}\right)\right), \text{ for } k = 0, \ldots, N\text{-}1$$

and returns a pointer to a real vector view object, or *null* if the create fails.

Prototypes

```
vsip_vview_f *vsip_vcreate_hanning_f (
                    vsip_length N,
                    vsip_memory_hint hint);
```

Arguments

`N`	Window length
`hint`	Memory type hints (Typically `VSIP_MEM_CONST`)

```
typedef enum {
  VSIP_MEM_NONE = 0,                 No hint
  VSIP_MEM_RDONLY = 1,               Read Only
  VSIP_MEM_CONST = 2,                Constant
  VSIP_MEM_SHARED = 3,               Shared
  VSIP_MEM_SHARED_RDONLY = 4,        Shared, Read Only
  VSIP_MEM_SHARED_CONST = 5          Shared, Constant
} vsip_memory_hint;
```

Return Value

Returns a pointer to the vector view object, or *null* on failure.

Restrictions

Restrictions

Errors

The arguments must conform to the following:

1. *N* must be greater than zero.
2. *hint* must be a valid member of the `vsip_memory_hint` enumeration.

Notes/References

There are two different widely used definitions of the Hanning window. The other being:

$$wind_k \leftarrow \frac{1}{2} \cdot \left(1 - \cos\left(\frac{2\pi \cdot k}{N-1}\right)\right), \text{ for } k = 0, \ldots, N\text{-}1$$

This form has a weight of zero for both end points of the window; we use the form that does not have zero end points.

If you want the window to be periodic of length *N*, you must generate a Hanning window of length $N-1$, copy it to a vector view of length *N*, and set the last point to 0.0.

Examples

```
#define N 256
…
vsip_vview_sp hanning = vsip_vcreate_hanning_sp(N,
          VSIP_MEM_CONST);
vsip_vview_sp x = vsip_vcreate(N, VSIP_MEM_NONE);
…
/* window the data in x */
vsip_vmul_sp(x,hanning,x);
…
vsip_valldestroy_sp(hanning); vsip_valldestroy_sp(x);
```

See Also

vsip_vcreate_kaiser_*f* Create Kaiser Window

Create a vector with Kaiser window weights.

Functionality

Creates a vector initialized with a Kaiser window of length *N,*

$$wind_k \leftarrow \frac{I_0\left[\beta \cdot \sqrt{1-\left(\frac{2\cdot k-(N-1)}{N-1}\right)^2}\right]}{I_0[\beta]}, \text{ for } k = 0, \ldots, N\text{-}1,$$

where

$$I_0[x] = \sum_{p=0}^{\infty}\left(\frac{x^p}{2^p\, p!}\right)^2$$

This function returns a pointer to a real vector view object, or *null* if the create fails.

Increasing β widens the main-lobe (transition width) and reduces the side-lobes.

Prototypes

```
vsip_vview_f *vsip_vcreate_kaiser_f (
                    vsip_length N,
                    vsip_scalar_f beta,
                    vsip_memory_hint hint);
```

Arguments

`N`	Length of window vector
`beta`	Real scalar, transition width parameter
`hint`	Memory type hints (Typically `VSIP_MEM_CONST`)

```
typedef enum {
  VSIP_MEM_NONE = 0,                 No hint
  VSIP_MEM_RDONLY = 1,               Read Only
  VSIP_MEM_CONST = 2,                Constant
  VSIP_MEM_SHARED = 3,               Shared
  VSIP_MEM_SHARED_RDONLY = 4,        Shared, Read Only
  VSIP_MEM_SHARED_CONST = 5          Shared, Constant
} vsip_memory_hint;
```

Return Value

Returns a pointer to the vector view object, or *null* on failure.

Restrictions

Errors

The arguments must conform to the following:

1. *N* must be greater than zero.
2. *hint* must be a valid member of the `vsip_memory_hint` enumeration.

Notes/References

Examples

```
#define N 256
#define BETA 0.4
…
vsip_vview_sp kaiser = vsip_vcreate_kaiser_sp(N, BETA,
          VSIP_MEM_CONST);
vsip_vview_sp x = vsip_vcreate_sp(N, VSIP_MEM_NONE);
…
/* window the data in x */
vsip_vmul_sp(x,kaiser,x);
…
vsip_valldestroy_sp(kaiser); vsip_valldestroy_sp(x);
```

See Also

Filter Functions

vsip_*d*fir_create_*f*	Create Decimated FIR Filter
vsip_*d*fir_destroy_*f*	Destroy FIR Filter Object
vsip_*d*firflt_*f*	Decimated FIR Filter
vsip_*d*fir_getattr_*f*	FIR Get Attributes
vsip_*d*fir_reset_*f*	Reset FIR Filter Object to Initial State
vsip_*d*iir_create_*f*	Create Cascaded IIR Filter
vsip_*d*iir_destroy_*f*	Destroy IIR Filter Object
vsip_*d*iirflt_*f*	Cascaded IIR Filter
vsip_*d*iir_getattr_*f*	Get IIR Attributes
vsip_diir_reset_f	Reset IIR Filter Object to Initial State

vsip_*d*fir_create_*f* Create Decimated FIR Filter

Create a decimated FIR filter object.

Functionality

Creates a decimated FIR filter object and returns a pointer to the object. The user specifies the *kernel* (filter coefficients and filter order), the integral output decimation factor, D, and the length of input segments (vectors) that will be filtered. The user also provides portable information on how to optimize the filter:
1) the number of segments to be filtered (e.g. 1, 16, etc.),
2) hints on what to optimize, and
3) if the filter will save state information for continuous filtering.

If the create fails, *null* is returned.

If requested the FIR filter object also encapsulates the filter's state information. The state is initialized to zero. The filter state allows long (semi-infinite) data streams to be processed in segments by successive calls to `vsip_dfirflt_f`.

Given a filter *kernel* of order M with coefficients h[3], the segment length is N, and the decimation factor is D. The decimated output y, is of length $\lceil (N - p)/D \rceil$.

The following is an example of how a FIR may be done. The actual method used is vendor dependent and must only supply the same functionality.

A FIR filter object is used to compute:

$$y_k \leftarrow \sum_{i=0}^{M} h_i \cdot \hat{x}_{p+k\cdot D-i}, \text{ for } k = 0, 1, \ldots, \lceil (N - p)/D \rceil - 1$$

Where

$$\hat{x}_j \equiv \begin{cases} x_j, & j \geq 0 \\ s_j, & j < 0 \end{cases}$$

Final conditions, s and p, are private internal state information.

$$s_j \leftarrow x_{N+j}, \text{ for } j = -1, -2, \ldots, -M$$

$$p \leftarrow D - 1 - [(N-1-p) \bmod D]$$

When the FIR filter object is created, s and p are initialized to zeros. If the save state option is not selected then s and p will remain initialized to zero.

Prototypes

```
vsip_fir_f *vsip_fir_create_f(
                    const vsip_vview_f *kernel,
                    vsip_symmetry symm,
                    vsip_length N,
                    vsip_length D,
                    vsip_obj_state state,
                    vsip_length ntimes,
                    vsip_alg_hint hint);
```

[3] There are M+1 coefficients for an M order filter.

```
vsip_rcfir_f *vsip_rcfir_create_f(
                    const vsip_vview_f *kernel,
                    vsip_symmetry symm,
                    vsip_length N,
                    vsip_length D,
                    vsip_obj_state state,
                    vsip_length ntimes,
                    vsip_alg_hint hint);
vsip_cfir_f *vsip_cfir_create_f(
                    const vsip_cvview_f *kernel,
                    vsip_symmetry symm,
                    vsip_length N,
                    vsip_length D,
                    vsip_obj_state state,
                    vsip_length ntimes,
                    vsip_alg_hint hint);
```

Arguments

`*kernel` Pointer to vector view object of non-redundant FIR filter coefficients,
non-symmetric: length $M+1$,
(conjugate) symmetric: length $\lceil (M+1)/2 \rceil$

`symm` Kernel symmetry, including length symmetry {*even*, *odd*}

```
typedef enum {
  VSIP_NONSYM = 0,                Non-symmetric
  VSIP_SYM_EVEN_LEN_ODD = 1,      (Even) Symmetric, odd length
  VSIP_SYM_EVEN_LEN_EVEN = 2      (Even) Symmetric, even length
} vsip_symmetry;
```

`N` Length of input data segment

`D` Decimation factor

`state` Object state history requirement. If the object is going to be used to filter more than one vector set to `VSIP_STATE_SAVE`. If the object is going to be used for single call filtering set state to `VSIP_STATE_NO_SAVE`.

```
typedef enum {
  VSIP_STATE_NO_SAVE = 1,         Don't save state, single call filter
  VSIP_STATE_SAVE = 2             Save state for continuous filtering
} vsip_obj_state;
```

`ntimes` Estimate of how many segments will be filtered. A value of zero is treated as semi-infinite (*a lot of times*).

`hint` Hint to help determine algorithm approach.

```
typedef enum {
  VSIP_ALG_TIME = 0,
  VSIP_ALG_SPACE = 1,
  VSIP_ALG_NOISE = 2
} vsip_alg_hint;
```

Return Value

Returns a pointer to an FIR filter object, or *null* if the create fails.

Restrictions

The decimation factor must be less than or equal to the filter length.

Errors

The arguments must conform to the following:

1. *kernel* must be a pointer to a valid vector view object.
2. *symm* must be a valid member of the `vsip_symmetry` enumeration.
3. $N \geq M$ (see *kernel* above).
4. $1 \leq D \leq M$.
5. *hint* must be a valid member of the `vsip_alg_hint` enumeration.
6. *state* must be a valid member of the `vsip_obj_state` enumeration.

Notes/References

For non-lowpass filters, $D = 1$ should be specified.

It is important that the *kernel* vector be only as long as defined under Arguments; i.e., the symmetric values of the filter between the kernel's center and its last value are not to be included in the *kernel*.

It is safe to destroy the *kernel* after creating the FIR filter object. [*Note to* implementors*:* the FIR object is required to encapsulate the information in the *kernel*, but is not required to encapsulate the data values of the *kernel*.]

The parameter *ntimes* in conjunction with the *hint* may be used (at the implementor's discretion) to pre-compute or optimize the FIR filter based on this information. This *may* include, but is not limited to, converting the kernel to the frequency domain for fast convolution. Ideally the implementation uses a-priori time and space information with *ntimes* to optimize the FIR filter object to meet the user's *hint.*

Hints are just that. Implementations are free to ignore them and it is up to the implementor to determine the precise effect of the hints. However, the spirit of the hints are:

1. Minimize total filtering execution time.
2. Minimize the filtering total memory requirements.
3. Maximize numeric accuracy/stability (minimize numeric noise).

Only one hint may be specified.

[*Notes to* Implementors*:* If minimum execution time is the goal, both direct and frequency domain fast convolution should be examined to determine the fastest method and the ideal FFT size for fast convolution. Decimation can be implemented with frequency domain fast convolution by using the equivalent poly-phase filter form. The direct time domain method uses minimal space.]

Examples

See Also

`vsip_`***d***`firflt_`***f***, and `vsip_`***d***`fir_destroy_`***f***

vsip_*d*fir_destroy_*f* Destroy FIR Filter Object

Destroy a FIR filter object.

Functionality

Destroys a FIR filter object freeing the associated memory. Returns non-zero on failure.

Prototypes

```
int vsip_fir_destroy_f(
                    vsip_fir_f *filt);
int vsip_rcfir_destroy_f(
                    vsip_rcfir_f *filt);
int vsip_cfir_destroy_f(
                    vsip_cfir_f *filt);
```

Arguments

`*filt` Pointer to a FIR filter object

Return Value

Returns zero on success and non-zero on failure.

Restrictions

Errors

The arguments must conform to the following:

1. The FIR filter object must be valid. An argument of *null* is not an error.

Notes/References

An argument of *null* is not an error.

[Implementors Note: a FIR filter object may also reference shared information such as a private FFT object. The shared object must maintain sufficient information to determine when it is possible to delete this shared information.]

Examples

See Also

vsip_*d*firflt_*f* Decimated FIR Filter

FIR filter an input sequence and decimate the output.

Functionality

Applies a FIR filter, specified by the FIR filter object to an input segment **x**, and computes a decimated output segment **y**. Initial and final filter state is encapsulated in the FIR filter object. Long data streams can be processed in segments by successive calls to `vsip_dfirflt_f`.

Case	Number of Output Samples	Length of **y**
$N \bmod D = 0$	N/D	N/D
$N \bmod D \neq 0$	$\lfloor N/D \rfloor$ or $\lceil N/D \rceil$	$\lceil N/D \rceil$

The return value is the number of output samples computed. When $\lfloor N/D \rfloor \neq \lceil N/D \rceil$, the output vector **y** will not be fully populated for every invocation.

N and D are determined by the creation of the FIR filter object.

Prototypes

```
int vsip_firflt_f (
                     vsip_fir_f *filt,
                     const vsip_vview_f *x,
                     const vsip_vview_f *y);
int vsip_rcfirflt_f (
                     vsip_rcfir_f *filt,
                     const vsip_cvview_f *x,
                     const vsip_cvview_f *y);
int vsip_cfirflt_f (
                     vsip_cfir_f *filt,
                     const vsip_cvview_f *x,
                     const vsip_cvview_f *y);
```

Arguments

`*filt`	Pointer to FIR filter object
`*x`	Pointer to input sequence (vector view object)
`*y`	Pointer to output sequence (vector view object)

Return Value

The return value is the number of output samples computed.

Restrictions

Filtering can not be performed in place.

Errors

The arguments must conform to the following:

1. The FIR filter object must be valid.
2. The **x** input vector view must be of length N (conformant with the FIR filter object).
3. The **y** output vector view must be of length $\lceil N/D \rceil$ (conformant with the FIR filter object).
4. The input **x**, and the output **y**, must not overlap.

Notes/References

The filter object is not "`const`" since it is modified with the updated state.

Note that `vsip_rcfirflt_f` implies a real filter applied to a complex vector.

[Note to Implementors: The final conditions, s and p, are abstractions and need not explicitly exist. Implementations are only required to correctly handle the filter state to support segment based filtering.]

Examples

Codelet to continuously filter data in segments of 1000 samples. Data comes from external source via a "ping-pong" buffer.

```
#include <vsip.h>

#define FOREVER 0
#define N 1000                    /* Segment is 1000 samples */
#define N2 (N/2)
#define M 17                      /* Filter order is 16 */
#define M2 ((M+1)/2)
extern const volatile vsip_scalar_f *buf_ping, *buf_pong;
float usr_kernel[M2] = {-0.0440,-0.0359,0.0507,0.0304,
                        -0.0364,-0.0965,0.0529,0.3092,0.4536};

/* Low pass half band filter kernel coefficients (symmetric, order 16):
 * -0.0440,-0.0359,0.0507,0.0304,-0.0364,-0.0965,0.0529,0.3092,
 * 0.4536,0.3092,0.0529,-0.0965,-0.0364,0.0304,0.0507,-0.0359,-0.0440 */
int pingpong = 0;
vsip_block_f* buf_blk = vsip_blockbind_f(buf_ping, N, VSIP_MEM_NONE);
vsip_vview_f* flt_in = vsip_vbind_f(buf_blk,0U,1,(vsip_length)N);
vsip_block_f *kblk = vsip_blockbind_f(usr_kernel, M2, VSIP_MEM_RDONLY);
vsip_vview_f *kernel = vsip_vbind_f(kblk,0U,1,(vsip_length)M2);
vsip_vview_f *flt_out = vsip_vcreate_f((vsip_length)N2, VSIP_MEM_NONE);
vsip_fir_f *lowpass;
vsip_blockadmit_f(kblk,VSIP_TRUE);

/* Create lowpass filter: segment length N, decimate by 2
 * filter is linear phase (symmetric) with odd number of coefficients */
  lowpass = vsip_fir_create_f(kernel,VSIP_SYM_EVEN_LEN_ODD,N,2,
            FOREVER,VSIP_ALG_TIME);

while{1) {
  /* Wait until data ready in buf_ping (buf_pong) */
  wait_until_data_ready();
  pingpong = !pingpong;

  /* Release buf_pong  (buf_ping) */
  vsip_blockrelease_f(buf_blk,VSIP_FALSE);
```

```
    /* Start DMA of next data frame into buf_pong (buf_ping) */
    /* Rebind and admit buf_ping (buf_pong) */
    dma_write((vsip_blockrebind_f(buf_blk,
                                  (pingpong ? buf_ping : buf_pong)));
    vsip_blockadmit_f(buf_blk, VSIP_TRUE);

    /* Continuously filter and decimate in segments */
    n = vsip_firflt_f(lowpass, buf, flt_out);
    /* Do some more processing with the decimated filter output */
    ...
  }
```

See Also

`vsip_`***d***`firflt_`***f***, and `vsip_`***d***`fir_destroy_`***f***

vsip_*d*fir_getattr_*f* FIR Get Attributes

Return the attributes of an FIR filter object.

Functionality

Returns the attribute values of a FIR filter object in structure passed by reference.

The attributes are:

- filter kernel length, $M+1$
- kernel symmetry,
- required length of an input segment (vector view),
- required length of an output segment (vector view), $\lceil in_len/D \rceil$
- output decimation factor,
- save state object.

Prototypes

```
void vsip_fir_getattr_f(
                      const vsip_fir_f *filt,
                      vsip_fir_attr_f *attr);
typedef struct {
   vsip_scalar_vi kernel_len;          M + 1
   vsip_symmetry symm;
   vsip_scalar_vi in_len;
   vsip_scalar_vi out_len;
   vsip_length decimation;             D
   vsip_obj_state state;
} vsip_fir_attr_f;
```

```
void vsip_rcfir_getattr_f(
                      const vsip_rcfir_f *filt,
                      vsip_rcfir_attr_f *attr);
typedef struct {
   vsip_scalar_vi kernel_len;          M + 1
   vsip_symmetry symm;
   vsip_scalar_vi in_len;
   vsip_scalar_vi out_len;
   vsip_length decimation;             D
   vsip_obj_state state;
} vsip_rcfir_attr_f;
```

```
void vsip_cfir_getattr_f(
                      const vsip_cfir_f *filt,
                      vsip_cfir_attr_f *attr);
typedef struct {
   vsip_scalar_vi kernel_len;          M + 1
   vsip_symmetry symm;
   vsip_scalar_vi in_len;
   vsip_scalar_vi out_len;
   vsip_length decimation;             D
   vsip_obj_state state;
} vsip_cfir_attr_f;
```

Where:

```
typedef enum {
   VSIP_NONSYM = 0,                 Non-symmetric
   VSIP_SYM_EVEN_LEN_ODD = 1,       (Even) Symmetric, odd length
   VSIP_SYM_EVEN_LEN_EVEN = 2       (Even) Symmetric, even length
} vsip_symmetry;

typedef enum {
   VSIP_STATE_NO_SAVE = 1,          Don't save state, single call filter
   VSIP_STATE_SAVE = 2              Save state for continuous filtering
} vsip_obj_state;
```

Arguments

`*filt`	Pointer to an FIR filter object
`*attr`	Pointer to a `vsip_dfir_attr_f` structure

Return Value

None.

Restrictions

Errors

The arguments must conform to the following:

1. The filter object must be valid.
2. The output attribute pointer must be valid – non-*null*.

Notes/References

The filter coefficient values are not accessible attributes.

For a symmetric kernel, the filter kernel length, $M + 1$, is not the length of the vector view, $\lceil (M+1)/2 \rceil$.

Examples

See Also

`vsip_dfir_create_f`

vsip_*d*fir_reset_*f* FIR Reset

Reset the state of a decimated FIR filter object.

Functionality

Resets the internal state of a previously created FIR filter object to the same state it would have immediately after creation.

Prototypes

```
void vsip_fir_reset_f(
                      vsip_fir_f *filt);
void vsip_rcfir_reset_f(
                      vsip_rcfir_f *filt);
void vsip_cfir_reset_f(
                      vsip_cfir_f *filt);
```

Arguments

`*filt` Pointer to the FIR filter object to be reset.

Return Value

None.

Restrictions

Errors

The arguments must conform to the following:

1. The filter object must be valid.

Notes/References

Examples

See Also

`vsip_dfir_create_f`, `vsip_dfirflt_f`

vsip_*d*iir_create_*f* Create Cascaded IIR Filter

Create a cascaded IIR filter object.

Functionality

Creates a cascaded 2nd order section IIR filter object, of order 2·*M*, and return a pointer to the object. The filter transfer function is:

$$H(z) = \prod_{k=0}^{M-1} \frac{b_{k,0} + b_{k,1} \cdot z^{-1} + b_{k,2} \cdot z^{-2}}{1 + a_{k,0} \cdot z^{-1} + a_{k,1} \cdot z^{-2}}$$

For an order 2·*M* filter, the numerator coefficients, **B**, are passed as an *M* by 3 matrix view object, and the denominator coefficients, **A**, are passed as an *M* by 2 matrix view object. Second order sections are applied in matrix row order.

The IIR filter object also encapsulates the filter's state information if the save state input object is set to `VSIP_STATE_SAVE`. The state is initialized to zero. The filter state allows long (semi-infinite) data streams to be processed in segments by successive calls to `vsip_diirflt_f`. If it is desired that the filter not save state information then the state object is set to `VSIP_STATE_NO_SAVE`.

If the create fails, *null* is returned, otherwise a pointer to the IIR filter object is returned.

Prototypes

```
vsip_iir_f *vsip_iir_create_f(
                    const vsip_mview_f *B,
                    const vsip_mview_f *A,
                    vsip_length N,
                    vsip_obj_state state,
                    vsip_length ntimes,
                    vsip_alg_hint hint);
```

Arguments

`*B` Pointer to a matrix view object of IIR filter numerator coefficients, *M* by 3

`*A` Pointer to a matrix view object of IIR filter denominator coefficients, *M* by 2

`N` *Length* of data segment

`state` Object state history requirement. If the object is going to be used to filter more than one vector set to `VSIP_STATE_SAVE`. If the object is going to be used for single call filtering set state to `VSIP_STATE_NO_SAVE`.

```
typedef enum {
  VSIP_STATE_NO_SAVE = 1,     Don't save state, single call filter
  VSIP_STATE_SAVE = 2         Save state for continuous filtering
} vsip_obj_state;
```

`ntimes` Estimate of how many segments will be filtered. A value of zero is treated as semi-infinite.

`hint` *Hint* to help determine algorithm approach.

```
typedef enum {
  VSIP_ALG_TIME = 0,
  VSIP_ALG_SPACE = 1,
  VSIP_ALG_NOISE = 2
} vsip_alg_hint;
```

Return Value

Returns a pointer to an IIR filter object, or *null* if the create fails.

Restrictions

Errors

The arguments must conform to the following:

1. All objects must be valid.
2. **B** matrix of size *M* by 3.
3. **A** matrix of size *M* by 2.
4. *N* must be greater than or equal to 2·*M*.
5. *hint* must be a valid member of the `vsip_alg_hint` enumeration.
6. *state* must be a valid member of the `vsip_obj_state` enumeration.

Notes/References

It is safe to destroy the filter coefficient matrices after creating the IIR filter object. [*Note to* implementors*:* the IIR object is required to encapsulate the information in the filter coefficient matrices, but is not required to encapsulate the data values of the filter coefficient matrices.]

The parameter *ntimes* in conjunction with the *hint* is used (at the implementor's discretion) to pre-compute or optimize the IIR filter based on this information. Ideally the implementation uses a-priori time and space information with *ntimes* to optimize the IIR filter object to meet the user's *hint.*

Hints are just that. Implementations are free to ignore them and it is up to the implementor to determine the precise effect of the hints. However, the spirit of the hints are:

1. Minimize total filtering execution time.
2. Minimize the filtering total memory requirements.
3. Maximize numeric accuracy/stability (minimize numeric noise).

Only one hint may be specified.

Examples

See Also

`vsip_`***d***`iirflt_`***f***, and `vsip_`***d***`iir_destroy_`***f***

vsip_*d*iir_destroy_*f* Destroy IIR Filter Object

Destroy an IIR filter object.

Functionality

Destroys an IIR filter object freeing the associated memory. Returns non-zero on failure.

Prototypes

```
int vsip_iir_destroy_f(
                      vsip_iir_f *filt);
```

Arguments

`*filt`	Pointer to an IIR filter object

Return Value

Returns zero on success and non-zero on failure.

Restrictions

Errors

The arguments must conform to the following:

1. The filter object must be valid. An argument of *null* is not an error.

Notes/References

An argument of *null* is not an error.

Note to Implementors: an IIR filter object may also reference shared information/object(s). The shared object(s) must maintain sufficient information to determine when it is possible to delete this shared information.

Examples

See Also

`vsip_diir_create_f`

vsip_*d*iirflt_*f* Cascaded IIR Filter

IIR filter an input sequence.

Functionality

Applies a cascaded 2nd order section IIR filter object, of order $2 \cdot M$, specified by the IIR filter object to an input segment x, and compute an output segment y. Initial and final filter state is encapsulated in the IIR filter object if `VSIP_STATE_SAVE` was selected when the filter objecect was created so that long data streams can be processed in segments by successive calls to `vsip_diirflt_f`. If `VSIP_STATE_NO_SAVE` was selected when the filter object was created then each call will filter the input into the output without state information from any previous call.

Prototypes

```
void vsip_iirflt_f (
                    vsip_iir_f *filt,
                    const vsip_vview_f *x,
                    const vsip_vview_f *y);
```

Arguments

`*filt`	Pointer to IIR filter object
`*x`	Pointer to input sequence (vector view object)
`*y`	Pointer to output sequence (vector view object)

Return Value

None.

Restrictions

Filtering can not be performed in place.

Errors

The arguments must conform to the following:

1. All the objects must be valid.
2. The **x** input, and **y** output vectors must be of length N.
3. The input **x**, and the output **y**, must not overlap.

Notes/References

The filter object is not "`const`" since it may be modified with the updated state.

Examples

See Also

`vsip_diir_create_f`, and `vsip_diir_destroy_f`

vsip_*d*iir_getattr_*f* Get IIR Attributes

Returns the attributes of an IIR filter object.

Functionality

Returns the attribute values of a IIR filter object in structure passed by reference.

The attributes are:

- filter order, number of 2nd order sections,
- required length of an input/output segment (vector view),
- save state object.

Prototypes

```
void vsip_iir_getattr_f(
                    const vsip_iir_f *filt,
                    vsip_iir_attr_f *attr);
```

Where

```
typedef struct {
   vsip_length n2nd;
   vsip_scalar_vi seg_len;
   vsip_obj_state state;
} vsip_iir_attr_f;
typedef enum {
   VSIP_STATE_NO_SAVE = 1,        Don't save state, single call filter
   VSIP_STATE_SAVE = 2            Save state for continuous filtering
} vsip_obj_state;
```

Arguments

`*filt`	Pointer to an IIR filter object
`*attr`	Pointer to a `vsip_iir_attr_f` structure

Return Value

None.

Restrictions

Errors

The arguments must conform to the following:

1. The filter object must be valid.
2. The output attribute pointer must be valid – non-*null*.

Notes/References

The filter coefficient values are not accessible attributes.

Examples

See Also

`vsip_diir_create_f`, `vsip_diir_destroy_f`, and `vsip_diir_reset_f`.

vsip_*d*iir_reset_*f* IIR Reset

Reset the state of an IIR filter object.

Functionality

Resets the internal state of a previously created IIR filter object to the same state it would have immediately after creation.

Prototypes

```
void vsip_iir_reset_f(
                      vsip_iir_f *filt);
```

Arguments

`*filt` Pointer to the IIR filter object to be reset.

Return Value

None.

Restrictions

Errors

The arguments must conform to the following:

1. The filter object must be valid.

Notes/References

Examples

See Also

`vsip_diir_create_f`, `vsip_diirflt_f`

Miscellaneous Signal Processing Functions

vsip_***S*histo_*p***	Histogram
vsip_*dS*freqswap_*f*	Frequency Swap

vsip_shisto_*p* Histogram

Compute the histogram of a vector (matrix).

Functionality

To form a histogram of a vector (matrix). The length of the output vector is *P*. The first element of the output vector (index 0) and the last element of the output vector (index *P*-1) are used to accumulate values of the input vector outside the range of interest. The bin size is therfore determined using (*P*-2)(number of bins to collect values less than than *src_max* and greater than or equal to *src_min*). The output vector is initialized to zero if the VSIP_HIST_RESET option is selected, or the histogram is accumulated on top of the current data in the output vector if the VSIP_HIST_ACCUM option is selected:

$c_{j_k} \leftarrow c_{j_k} + 1.$, for $k = 0, 1, \ldots, N\text{-}1$

$c_{j_{k,l}} \leftarrow c_{j_{k,l}} + 1$, for $k = 0, 1, \ldots, M\text{-}1$; for $l = 0, 1, \ldots, N\text{-}1$

where

$$j_k = \begin{cases} 0 & src_k < src_\min \\ P-1 & src_k \geq src_\max \\ \left\lfloor (P-2) \cdot \dfrac{src_k - src_\min}{src_\max - src_\min} \right\rfloor + 1 & src_\min \leq src_k < src_\max \end{cases}$$

or

$$j_{k,l} = \begin{cases} 0 & src_k < src_\min \\ P-1 & src_k \geq src_\max \\ \left\lfloor (P-2) \cdot \dfrac{src_k - src_\min}{src_\max - src_\min} \right\rfloor + 1 & src_\min \leq src_k < src_\max \end{cases}$$

Prototypes

```
void vsip_vhisto_f(
                      const vsip_vview_f *src,
                      vsip_scalar_f min_bin,
                      vsip_scalar_f max_bin,
                      vsip_hist_opt opt,
                      const vsip_vview_f *hist);
void vsip_vhisto_i(
                      const vsip_vview_i *src,
                      vsip_scalar_i min_bin,
                      vsip_scalar_i max_bin,
                      vsip_hist_opt opt,
                      const vsip_vview_i *hist);
```

```
void vsip_mhisto_f(
                     const vsip_mview_f *src,
                     vsip_scalar_f min_bin,
                     vsip_scalar_f max_bin,
                     vsip_hist_opt opt,
                     const vsip_vview_f *hist);
void vsip_mhisto_i(
                     const vsip_mview_i *src,
                     vsip_scalar_i min_bin,
                     vsip_scalar_i max_bin,
                     vsip_hist_opt opt,
                     const vsip_vview_i *hist);
typedef enum{
  VSIP_HIST_RESET = 1,
  VSIP_HIST_ACCUM = 2
} vsip_hist_opt;
```

Arguments

`*src`	View of source vector (matrix)
`min_bin`	Threshold for minimum bin
`max_bin`	Threshold for maximum bin
`opt`	Enumerated type to determine if the output histogram is first initialized to zero, or is accumulated on top of previous data.
`*hist`	View of histogram vector, of length *P*

```
typedef enum{
  VSIP_HIST_RESET = 1,
  VSIP_HIST_ACCUM = 2
} vsip_hist_opt;
```

Return Value

None

Restrictions

Errors

The arguments must conform to the following:

1. All the view objects must be valid.
2. `min_bin` < `max_bin`.

Notes/References

The first and last bins collect all the values less than `min_val`, and greater or equal to `max_val`, respectively. If these outlier values are not desired, create and bind a view of length *P*, and create a derived view (using the `vsip_vsubview_f` function) of the first view starting at index 1 and of length *P*-2. Collect the histogram into the larger view. Just the histogram values without the outliers are available in the derived view.

Examples

See Also

vsip_*ds*freqswap_*f* Frequency Swap

Swaps halves of a vector, or quadrants of a matrix, to remap zero frequencies from the origin to the middle.

Functionality

Swap:

$x_i \leftarrow x_{((\lfloor N/2 \rfloor + i) \bmod N)}$ for $j = 0, 1, \ldots, N\text{-}1$

$x_{i,j} \leftarrow x_{((\lfloor M/2 \rfloor + i) \bmod M)\,\text{ϒ}\,((\lfloor N/2 \rfloor + j) \bmod N)}$ for $i = 0, 1, \ldots, M\text{-}1$; for $j = 0, 1, \ldots, N\text{-}1$

Prototypes

```
void vsip_vfreqswap_f(
                      const vsip_vview_f *x);
void vsip_cvfreqswap_f(
                      const vsip_cvview_f *x);
void vsip_mfreqswap_f(
                      const vsip_mview_f *x);
void vsip_cmfreqswap_f(
                      const vsip_cmview_f *x);
```

Arguments

`*x`	Pointer to an input/output vector (matrix) view object

Return Value

Void.

Restrictions

Errors

The arguments must conform to the following:

1. The input/output object must be valid.

Notes/References

Examples

See Also

LINEAR ALGEBRA FUNCTIONS

Introduction

The following functions operate on or produce matrix results. They are currently only defined for floating point types.

In VSIPL, values in a block can be viewed as a vector (`vsip_`*d*`vview_`*p*) or as a matrix (`vsip_`*d*`mview_`*p*). For notational convenience, the matrix functions treat the vector view objects as column vectors.

Matrix objects may be stored in memory in either row major or column major order (C order or FORTRAN order) by the application programmers choice of matrix view strides.

Note: Many of the matrix functions that make up a family, such as matrix multiply, have been implemented in other libraries like BLAS as one function with many parameters. BLAS and other libraries historically made such choices due to the limitations of a seven letter (or short) subroutine names. Here they are handled as separately named functions. This eliminates some runtime checking by moving it to compile time, and may simplify some optimizations. (The namespace approach is also more closely matched to potential future object oriented bindings.) Both approaches have similar code development sizes. However, the separately named function approach reduces the size of the linked program by not including unnecessary functionality.

Matrix and Vector Operations

vsip_**cmherm**_*p*	Matrix Hermitian
vsip_**cvjdot**_*p*	Complex Vector Conjugate Dot Product
vsip_*d***gemp**_*p*	General Matrix Product
vsip_*d***gems**_*p*	General Matrix Sum
vsip_*ds***kron**_*p*	Kronecker Matrix Product
vsip_*d***mprod3**_*p*	3 by 3 Matrix Product
vsip_*d***mprod4**_*p*	4 by 4 Matrix Product
vsip_*d***mprod**_*p*	Matrix Product
vsip_**cmprodh**_*p*	Matrix Hermitian Product
vsip_**cmprodj**_*p*	Matrix Conjugate Product
vsip_*d***mprodt**_*p*	Matrix Transpose Product
vsip_*d***mvprod3**_*p*	3 by 3 Matrix Vector Product
vsip_*d***mvprod4**_*p*	4 by 4 Matrix Vector Product
vsip_*d***mvprod**_*p*	Matrix Vector Product
vsip_*d***mtrans**_*p*	Matrix Transpose
vsip_*d***vdot**_*p*	Vector Dot Product
vsip_*d***vmprod**_*p*	Vector Matrix Product
vsip_*d***vouter**_*p*	Vector Outer Product

vsip_cmherm_*p* Matrix Hermitian

Complex Hermitian (conjugate transpose) of a matrix

Functionality

Returns the *N* by *M* matrix **C**, which is the Hermitian (conjugate transpose) of an *M* by *N* matrix **A**.

$$\mathbf{C} \leftarrow \mathbf{A}^H$$

Prototypes

```
void vsip_cmherm_f(
                    const vsip_cmview_f *A,
                    const vsip_cmview_f *C);
```

Arguments

`*A`	View of input *M* by *N* matrix.
`*C`	View of output *N* by *M* matrix.

Return Value

None.

Restrictions

If the matrix A is square, the transpose is in place if A and C resolve to the same object, otherwise A and C must be disjoint.

Errors

The arguments must conform to the following:

1. All objects must be valid.
2. The matrices **A**, and **C** must be conformant.
3. If the matrix is not square, $M \neq N$, the input and output matrix views must not overlap. If the matrix is square, $M = N$, the input and output views must be identical views of the same block, or must not overlap.

Notes/References

Examples

See Also

`vsip_dmtrans_f`, and `vsip_dgems_f`

vsip_cvjdot_*p* Complex Vector Conjugate Dot Product

Compute the conjugate inner (dot) product of two complex vectors.

Functionality

Compute the conjugate dot product

$$r \leftarrow \mathbf{a}^T \mathbf{b}^*$$

$$r \leftarrow \sum_{j=0}^{N-1} a_j \cdot b_j^*$$

Where "*" denotes complex conjugate.

Prototypes

```
vsip_cscalar_f vsip_cvjdot_f(
                     const vsip_cvview_f *a,
                     const vsip_cvview_f *b);
```

Arguments

`*a`	View of input vector.
`*b`	View of input vector.

Return Value

This function returns a complex scalar of the same precision as the input vectors.

Restrictions

Overflow may occur. The result of overflow is implementation dependent.

Errors

The arguments must conform to the following:

1. Arguments for input must be the same size.
2. All view objects must be valid.

Notes/References

Examples

```
#include<stdio.h>
#include "vsip.h"

#define L 7                    /* length */

int main()
{
    int i;
    vsip_vview_d*  dataRe;
    vsip_vview_d*  dataIm;
    vsip_cvview_d* cvectorLeft;
    vsip_cvview_d* cvectorRight;
    vsip_cscalar_d cjdotpr,cLeft,cRight;

    vsip_init((void *)0);
    dataRe = vsip_vcreate_d(L, VSIP_MEM_NONE);
    dataIm = vsip_vcreate_d(L, VSIP_MEM_NONE);
    cvectorLeft = vsip_cvcreate_d(L, VSIP_MEM_NONE);
    cvectorRight = vsip_cvcreate_d(L, VSIP_MEM_NONE);
    /* Make up some data to use for a dot product*/
    vsip_vramp_d(1.0, 1.0 , dataRe);
```

```
    vsip_vramp_d(1.0, -2.0/(double)(L-1), dataIm);
    vsip_vcmplx_d(dataRe, dataIm, cvectorLeft);
    vsip_vcmplx_d(dataIm, dataRe, cvectorRight);
    cjdotpr = vsip_cvjdot_d(cvectorLeft, cvectorRight);
    for(i=0; i<L-1; i++){
        cLeft = vsip_cvget_d(cvectorLeft, i);
        cRight = vsip_cvget_d(cvectorRight, i);
        printf("(%7.4f + %7.4fi) * conj(%7.4f + %7.4fi) +\n",
               vsip_real_d(cLeft), vsip_imag_d(cLeft),
               vsip_real_d(cRight),vsip_imag_d(cRight));
    }
    cLeft = vsip_cvget_d(cvectorLeft, L-1);
    cRight = vsip_cvget_d(cvectorRight, L-1);
    printf("(%7.4f + %7.4fi) * conj(%7.4f + %7.4fi) = "
            "(%7.4f + %7.4fi)\n",
            vsip_real_d(cLeft),vsip_real_d(cLeft),
            vsip_real_d(cRight),vsip_imag_d(cRight),
            vsip_real_d(cjdotpr),vsip_imag_d(cjdotpr)); printf("\n");

    /* Do the same conjugate dot product with inputs reversed */
    cjdotpr = vsip_cvjdot_d(cvectorRight, cvectorLeft);
    /* now print out the data and the Result */
    for(i=0; i<L-1; i++){
        cLeft = vsip_cvget_d(cvectorLeft, i);
        cRight = vsip_cvget_d(cvectorRight, i);
        printf("(%7.4f + %7.4fi) * conj(%7.4f + %7.4fi) +\n",
            vsip_real_d(cRight),cRight.i,vsip_real_d(cLeft),cLeft.i);
    }
    cLeft = vsip_cvget_d(cvectorLeft, L-1);
    cRight = vsip_cvget_d(cvectorRight, L-1);
    printf("(%7.4f + %7.4fi) * conj(%7.4f + %7.4fi) = "
            "(%7.4f + %7.4fi)\n",
            vsip_real_d(cRight), vsip_imag_d(cRight),
            vsip_real_d(cLeft),  vsip_imag_d(cLeft),
            vsip_real_d(cjdotpr),vsip_imag_d(cjdotpr));
    /* destroy the vector views and any associated blocks */
    vsip_blockdestroy_d(vsip_vdestroy_d(dataRe));
    vsip_blockdestroy_d(vsip_vdestroy_d(dataIm));
    vsip_cblockdestroy_d(vsip_cvdestroy_d(cvectorLeft));
    vsip_cblockdestroy_d(vsip_cvdestroy_d(cvectorRight));
    vsip_finalize((void *)0);
    return 0;
}
/* (1.0000 +  1.0000i) * conj( 1.0000 +  1.0000i) +
(2.0000 +  0.6667i) * conj( 0.6667 +  2.0000i) +
(3.0000 +  0.3333i) * conj( 0.3333 +  3.0000i) +
(4.0000 +  0.0000i) * conj( 0.0000 +  4.0000i) +
(5.0000 + -0.3333i) * conj(-0.3333 +  5.0000i) +
(6.0000 + -0.6667i) * conj(-0.6667 +  6.0000i) +
(7.0000 +  7.0000i) * conj(-1.0000 +  7.0000i) = (-18.6667 + -136.8889i)
(1.0000 +  1.0000i) * conj( 1.0000 +  1.0000i) +
(0.6667 +  2.0000i) * conj( 2.0000 +  0.6667i) +
(0.3333 +  3.0000i) * conj( 3.0000 +  0.3333i) +
(0.0000 +  4.0000i) * conj( 4.0000 +  0.0000i) +
(-0.3333 +  5.0000i) * conj( 5.0000 + -0.3333i) +
(-0.6667 +  6.0000i) * conj( 6.0000 + -0.6667i) +
(-1.0000 +  7.0000i) * conj( 7.0000 + -1.0000i) = (-18.6667 + 136.8889i) */
```

See Also

`vsip_`*d*`vdot_`*p*

vsip_*d*gemp_*p* General Matrix Product

Calculate the general product of two matrices and accumulate.

Functionality

Computes the following matrix operation

$$\mathbf{C} \leftarrow \alpha \cdot \mathrm{op}(\mathbf{A}) \cdot \mathrm{op}(\mathbf{B}) + \beta \cdot \mathbf{C}$$

where op(**X**) is one of the following

$$\mathrm{op}(\mathbf{X}) = \mathbf{X},\ \mathrm{op}(\mathbf{X}) = \mathbf{X}^{\mathrm{T}},\ \mathrm{op}(\mathbf{X}) = \mathbf{X}^{H},\ \mathrm{op}(\mathbf{X}) = \mathbf{X}^{*},$$

α and β are scalars, **A**, **B**, **C** are matrices, op(**A**) is an *M* by *P* matrix, op(**B**) is a *P* by *N* matrix and **C** is an *M* by *N* matrix.

Prototypes

```
void vsip_gemp_f(
                    vsip_scalar_f alpha,
                    const vsip_mview_f *A,
                    vsip_mat_op OpA,
                    const vsip_mview_f *B,
                    vsip_mat_op OpB,
                    vsip_scalar_f beta,
                    const vsip_mview_f *C);
void vsip_cgemp_f(
                    vsip_cscalar_f alpha,
                    const vsip_cmview_f *A,
                    vsip_mat_op OpA,
                    const vsip_cmview_f *B,
                    vsip_mat_op OpB,
                    vsip_cscalar_f beta,
                    const vsip_cmview_f *C);
```

Arguments

`alpha` (Real/Complex) scalar.

`*A` View of input matrix **A.**

`OpA` Specifies the form of op(**A**).

```
typedef enum {
  VSIP_MAT_NTRANS = 0,
  VSIP_MAT_TRANS = 1,
  VSIP_MAT_HERM = 2,
  VSIP_MAT_CONJ = 3
} vsip_mat_op;
```

op(**A**) = **A**
op(**A**) = $\mathbf{A}^{T}$
op(**A**) = $\mathbf{A}^{H}$ (complex only)
op(**X**) = $\mathbf{A}^{*}$ (complex only)

`*B` View of input matrix **B.**

`OpB` Specifies the form of op(**B**).

```
typedef enum {
  VSIP_MAT_NTRANS = 0,
  VSIP_MAT_TRANS = 1,
  VSIP_MAT_HERM = 2,
  VSIP_MAT_CONJ = 3
} vsip_mat_op;
```

op(**B**) = **B**
op(**B**) = $\mathbf{B}^T$
op(**B**) = $\mathbf{B}^H$ (complex only)
op(**B**) = $\mathbf{B}^*$ (complex only)

`beta` (Real/Complex) scalar.

`*C` View of output *M* by *N* matrix.

Return Value

None.

Restrictions

The result matrix view, may not overlap either input matrix view.

Errors

The arguments must conform to the following:

1. All objects must be valid.
2. The matrices **A**, **B**, and **C** must be conformant.
3. The input and output matrix views must not overlap.
4. `OpA` and `OpB` must be valid.
   Real: `OpA, OpB ∈ {VSIP_MAT_NTRANS, VSIP_MAT_TRANS}`
   Complex: `OpA, OpB ∈ {VSIP_MAT_NTRANS, VSIP_MAT_TRANS, VSIP_MAT_HERM, VSIP_MAT_CONJ}`

Notes/References

Examples

See Also

`vsip_dmprodt_f`, `vsip_cmprodh_f`, `vsip_dmprod_f`, `vsip_cmprodj_f`, `vsip_dmprod3_f`, and `vsip_dmprod4_f`

vsip_*d*gems_*p* General Matrix Sum

Calculate a general matrix sum.

Functionality

Computes the following matrix operation

$\mathbf{C} \leftarrow \alpha \cdot \mathrm{op}(\mathbf{A}) + \beta \cdot \mathbf{C}$

where op(**X**) is one of the following

$\mathrm{op}(\mathbf{A}) = \mathbf{A}$, $\mathrm{op}(\mathbf{A}) = \mathbf{A}^{\mathrm{T}}$, $\mathrm{op}(\mathbf{A}) = \mathbf{A}^{H}$, $\mathrm{op}(\mathbf{A}) = \mathbf{A}^{*}$,

α and β are scalars, **A**, and **C** are matrices, op(**A**) is an *M* by *N* matrix, and **C** is an *M* by *N* matrix.

Prototypes

```
void vsip_gems_f(
                    vsip_scalar_f alpha,
                    const vsip_mview_f *A,
                    vsip_mat_op OpA,
                    vsip_scalar_f beta,
                    const vsip_mview_f *C);
void vsip_cgems_f(
                    vsip_cscalar_f alpha,
                    const vsip_cmview_f *A,
                    vsip_mat_op OpA,
                    vsip_cscalar_f beta,
                    const vsip_cmview_f *C);
```

Arguments

`alpha` (Real/Complex) scalar.

`*A` View of input matrix **A.**

`OpA` Specifies the form of op(**A**).

```
typedef enum {
  VSIP_MAT_NTRANS = 0,
  VSIP_MAT_TRANS = 1,
  VSIP_MAT_HERM = 2,
  VSIP_MAT_CONJ = 3
} vsip_mat_op;
```

op(**A**) = **A**
op(**A**) = $\mathbf{A}^{T}$
op(**A**) = $\mathbf{A}^{H}$ (complex only)
op(**A**) = $\mathbf{A}^{*}$ (complex only)

`beta` (Real/Complex) scalar.

`*C` View of output *M* by *N* matrix.

Return Value

None.

Restrictions

The result matrix view **C** may not overlap the input matrix view **A**.

Errors

The arguments must conform to the following:

1. All objects must be valid.
2. The matrices **A**, and **C** must be conformant.
3. The input and input/output matrix views must not overlap.
4. `OpA` must be valid.
   Real: `OpA ∈ {VSIP_MAT_NTRANS, VSIP_MAT_TRANS}`
   Complex: `OpA ∈ {VSIP_MAT_NTRANS, VSIP_MAT_TRANS, VSIP_MAT_HERM, VSIP_MAT_CONJ}`

Notes/References

Examples

See Also

`vsip_`*`d`*`mtrans_`*`f`*, and `vsip_`*`d`*`mherm_`*`f`*

vsip_*dskron*_*p* Kronecker Product

Calculate the Kronecker tensor product of two vectors or matrices.

Functionality

If **x** and **y** are vectors of length N and M respectively, then this function computes a scalar multiple of a Kronecker product of **x** and **y**. That is,

$$\mathbf{C} \leftarrow \alpha\cdot(\mathbf{x} \otimes \mathbf{y})$$

where

$$\mathbf{x} \otimes \mathbf{y} \equiv [\, x_0\mathbf{y}\; x_1\mathbf{y}\; \ldots\; x_{N\text{-}1}\mathbf{y}],$$

and α is a scalar. The resulting matrix, **C**, is an M by N matrix.

If **A** is an M by N matrix and **B** is an K by L matrix , then this function computes a scalar multiple of a Kronecker product of **A** and **B**. That is,

$$\mathbf{C} \leftarrow \alpha\cdot(\mathbf{A} \otimes \mathbf{B})$$

where

$$A \otimes B \equiv \begin{bmatrix} a_{0,0}B & a_{0,1}B & \cdots & a_{0,N-1}B \\ a_{1,0}B & a_{1,1}B & \cdots & a_{1,N-1}B \\ \vdots & \vdots & \ddots & \vdots \\ a_{M-1,0}B & a_{M-1,1}B & \cdots & a_{M-1,N-1}B \end{bmatrix}$$

and α is a scalar. The resulting matrix, **C**, is an $M{\cdot}K$ by $N{\cdot}L$ matrix.

Prototypes

```
void vsip_vkron_f(
                      vsip_scalar_f alpha,
                      const vsip_vview_f *x,
                      const vsip_vview_f *y,
                      const vsip_mview_f *C);
void vsip_cvkron_f(
                      vsip_cscalar_f alpha,
                      const vsip_cvview_f *x,
                      const vsip_cvview_f *y,
                      const vsip_cmview_f *C);
void vsip_mkron_f(
                      vsip_scalar_f alpha,
                      const vsip_mview_f *A,
                      const vsip_mview_f *B,
                      const vsip_mview_f *C);
void vsip_cmkron_f(
                      vsip_cscalar_f alpha,
                      const vsip_cmview_f *A,
                      const vsip_cmview_f *B,
                      const vsip_cmview_f *C);
```

Arguments

`alpha`	(Real/Complex) scalar.
`*x`	View of input vector.
`*y`	View of input vector.
`*A`	View of input matrix .
`*B`	View of input matrix.
`*C`	View of output matrix.

Return Value

None.

Restrictions

The result matrix view may not overlap either input matrix view.

Errors

The arguments must conform to the following:

1. All objects must be valid.
2. The vectors and matrix **x**, **y**, and **C** or the matrices **A**, **B**, and **C** must be conformant.
3. The output matrix view and the input vector/matrix views must not overlap.

Notes/References

Examples

See Also

`vsip_`*`d`*`vouter_`*`f`*

vsip_*d*mprod3_*p* 3 by 3 Matrix Product

Calculate the product of a 3 by 3 matrix and a 3 by *N* matrix.

Functionality

Computes the product of a 3 by 3 matrix **A** and a 3 by *N* matrix, **B**. The result of this operation, **C** ← **AB**, is a 3 by *N* matrix.

$$c_{i,j} \leftarrow \sum_{k=0}^{2} a_{i,k} b_{k,j}\text{, for } i = 0, 1, 2; \text{ for } j = 0, 1, \ldots, N\text{-}1$$

Prototypes

```
void vsip_mprod3_f(
                      const vsip_mview_f *A,
                      const vsip_mview_f *B,
                      const vsip_mviewW_f *C);
void vsip_cmprod3_f(
                      const vsip_cmview_f *A,
                      const vsip_cmview_f *B,
                      const vsip_cmview_f *C);
```

Arguments

`*A`	View of input 3 by 3 matrix.
`*B`	View of input 3 by *N* matrix.
`*C`	View of output 3 by *N* matrix.

Return Value

None.

Restrictions

The result matrix view may not overlap either input matrix view.

Errors

The arguments must conform to the following:

1. All objects must be valid.
2. The matrices **A**, **B**, and **C** must be conformant.
3. The input and output matrix views must not overlap.

Notes/References

Examples

See Also

`vsip_dmprodt_f`, `vsip_cmprodh_f`, `vsip_dmprod_f`, `vsip_cmprodj_f`, `vsip_dmprod4_f`, and `vsip_dgemp_f`

vsip_*d*mprod4_*p* 4 by 4 Matrix Product

Calculate the product of a 4 by 4 matrix and a 4 by *N* matrix.

Functionality

Computes the product of a 4 by 4 matrix, **A**, times a 4 by *N* matrix, **B**. The result of this operation, **C** ← **AB**, is a 4 by *N* matrix.

$$c_{i,j} \leftarrow \sum_{k=0}^{3} a_{i,k} b_{k,j}\text{, for } i = 0, 1, 2, 3\text{; for } j = 0, 1, \ldots, N\text{-}1$$

Prototypes

```
void vsip_mprod4_f(
                      const vsip_mview_f *A,
                      const vsip_mview_f *B,
                      const vsip_mview_f *C);
void vsip_cmprod4_f(
                      const vsip_cmview_f *A,
                      const vsip_cmview_f *B,
                      const vsip_cmview_f *C);
```

Arguments

`*A`	View of input 4 by 4 matrix.
`*B`	View of input 4 by *N* matrix.
`*C`	View of output 4 by *N* matrix.

Return Value

None.

Restrictions

The result matrix view may not overlap either input matrix view.

Errors

The arguments must conform to the following:

1. All objects must be valid.
2. The matrices **A**, **B**, and **C** must be conformant.
3. The input and output matrix views must not overlap.

Notes/References

Examples

See Also

`vsip_dmprodt_f`, `vsip_cmprodh_f`, `vsip_dmprod_f`, `vsip_cmprodj_f`, `vsip_dmprod3_f`, and `vsip_dgemp_f`

vsip_*d*mprod_*p* Matrix Product

Calculate the product of two matrices.

Functionality

Computes the product of an M by P matrix, **A**, and a P by N matrix, **B**. The result of this operation, **C** ← **AB**, is an M by N matrix.

$$c_{i,j} \leftarrow \sum_{k=0}^{P-1} a_{i,k} b_{k,j}\text{, for } i = 0, 1, \ldots, M\text{-}1\text{; for } j = 0, 1, \ldots, N\text{-}1$$

Prototypes

```
void vsip_mprod_f(
                      const vsip_mview_f *A,
                      const vsip_mview_f *B,
                      const vsip_mview_f *C);
void vsip_cmprod_f(
                      const vsip_cmview_f *A,
                      const vsip_cmview_f *B,
                      const vsip_cmview_f *C);
```

Arguments

`*A`	View of input M by P matrix.
`*B`	View of input P by N matrix.
`*C`	View of output M by N matrix.

Return Value

None.

Restrictions

The result matrix view may not overlap either input matrix view.

Errors

The arguments must conform to the following:

1. All objects must be valid.
2. The matrices **A**, **B**, and **C** must be conformant.
3. The input and output matrix views must not overlap.

Notes/References

Examples

See Also

`vsip_dmprodt_f`, `vsip_cmprodh_f`, `vsip_cmprodj_f`, `vsip_dmprod3_f`, `vsip_dmprod4_f`, and `vsip_dgemp_f`

vsip_cmprodh_*p* Matrix Hermitian Product

Calculate the product a complex matrix and the Hermitian of a complex matrix.

Functionality

Computes the product of an M by P complex matrix, **A**, and the Hermitian of a N by P complex matrix, **B**. The result of this operation, $\mathbf{C} \leftarrow \mathbf{AB}^H$, is an M by N complex matrix.

$$c_{i,j} \leftarrow \sum_{k=0}^{P-1} a_{i,k} b_{j,k}^{*} \text{ , for } i = 0, 1, \ldots, M\text{-}1; \text{ for } j = 0, 1, \ldots, N\text{-}1$$

Prototypes

```
void vsip_cmprodh_f(
                    const vsip_cmview_f *A,
                    const vsip_cmview_f *B,
                    const vsip_cmview_f *C);
```

Arguments

`*A`	View of input M by P matrix.
`*B`	View of input N by P matrix.
`*C`	View of output M by N matrix.

Return Value

None.

Restrictions

The result matrix view may not overlap either input matrix view.

Errors

The arguments must conform to the following:

1. All objects must be valid.
2. The matrices **A**, **B**, and **C** must be conformant.
3. The input and output matrix views must not overlap.

Notes/References

Examples

See Also

`vsip_dmprodt_f`, `vsip_dmprod_f`, `vsip_cmprodj_f`, `vsip_dmprod3_f`, `vsip_dmprod4_f`, and `vsip_dgemp_f`

vsip_cmprodj_*p* Matrix Conjugate Product

Calculate the product a matrix and the conjugate of a matrix.

Functionality

Computes the product of an M by P matrix, **A**, and the conjugate of a P by N matrix, **B**. The result of this operation, $\mathbf{C} \leftarrow \mathbf{AB}^*$, is an M by N matrix.

$$c_{i,j} \leftarrow \sum_{k=0}^{P-1} a_{i,k} b_{k,j}^* \text{, for } i = 0, 1, \ldots, M\text{-}1\text{; for } j = 0, 1, \ldots, N\text{-}1$$

Prototypes

```
void vsip_cmprodj_f(
                    const vsip_cmview_f *A,
                    const vsip_cmview_f *B,
                    const vsip_cmview_f *C);
```

Arguments

`*A`	View of input M by P matrix.
`*B`	View of input P by N matrix.
`*C`	View of output M by N matrix.

Return Value

None.

Restrictions

The result matrix view may not overlap either input matrix view.

Errors

The arguments must conform to the following:

1. All objects must be valid.
2. The matrices **A**, **B**, and **C** must be conformant.
3. The input and output matrix views must not overlap.

Notes/References

Examples

See Also

`vsip_dmprodt_f`, `vsip_cmprodh_f`, `vsip_dmprod_f`, `vsip_dmprod3_f`, `vsip_dmprod4_f`, and `vsip_dgemp_f`

vsip_*d*mprodt_*p* Matrix Transpose Product

Calculate the product of a matrix and the transpose of a matrix.

Functionality

Computes the product of an M by P matrix, **A**, times the transpose of an N by P matrix, **B**. The result of this operation, $\mathbf{C} \leftarrow \mathbf{AB}^T$, is an M by N matrix.

$$c_{i,j} \leftarrow \sum_{k=0}^{P-1} a_{i,k} b_{j,k} \text{ , for } i = 0, 1, \ldots, M\text{-}1; \text{ for } j = 0, 1, \ldots, N\text{-}1$$

Prototypes

```
void vsip_mprodt_f(
                      const vsip_mview_f *A,
                      const vsip_mview_f *B,
                      const vsip_mview_f *C);
void vsip_cmprodt_f(
                      const vsip_cmview_f *A,
                      const vsip_cmview_f *B,
                      const vsip_cmview_f *C);
```

Arguments

`*A`	View of input M by P matrix.
`*B`	View of input N by P matrix.
`*C`	View of output M by N matrix.

Return Value

None.

Restrictions

The result matrix view may not overlap either input matrix view.

Errors

The arguments must conform to the following:

1. All objects must be valid.
2. The matrices **A**, **B**, and **C** must be conformant.
3. The input and output matrix views must not overlap.

Notes/References

Examples

See Also

`vsip_cmprodh_f`, `vsip_dmprod_f`, `vsip_cmprodj_f`, `vsip_dmprod3_f`, `vsip_dmprod4_f`, and `vsip_dgemp_f`

vsip_*d*mvprod3_*p* 3 by 3 Matrix Vector Product

Calculate the product of a 3 by 3 matrix and a vector.

Functionality

Computes product of an 3 by 3 matrix and a vector.

$$\mathbf{y} \leftarrow \mathbf{A}\mathbf{x}$$

$$y_i \leftarrow \sum_{j=0}^{2} a_{i,j} x_j \text{, for } i = 0, 1, 2$$

Prototypes

```
void vsip_mvprod3_f(
                      const vsip_mview_f *A,
                      const vsip_vview_f *x,
                      const vsip_vview_f *y);
void vsip_cmvprod3_f(
                      const vsip_cmview_f *A,
                      const vsip_cvview_f *x,
                      const vsip_cvview_f *y);
```

Arguments

`*A`	View of input 3 by 3 matrix.
`*x`	View of input vector of length 3.
`*y`	View of output vector of length 3.

Return Value

None.

Restrictions

The result vector view may not overlap either input matrix/vector view.

Errors

The arguments must conform to the following:

1. All objects must be valid.
2. The matrix and vectors **A**, **x**, and **y** must be conformant.
3. The input and output matrix/vector views must not overlap.

Notes/References

These functions may be implemented as macros.

Examples

See Also

`vsip_dmvprod_f`, `vsip_dmvprod4_f`, and `vsip_dvmprod_f`

vsip_*d*mvprod4_*p* 4 by 4 Matrix Vector Product

Calculate the product of a 4 by 4 matrix times a vector.

Functionality

Computes product of an 4 by 4 matrix times a vector.

$\mathbf{y} \leftarrow \mathbf{A}\mathbf{x}$

$y_i \leftarrow \sum_{j=0}^{3} a_{i,j} x_j$, for $i = 0, 1, 2, 3$

Prototypes

```
void vsip_mvprod4_f(
                      const vsip_mview_f *A,
                      const vsip_vview_f *x,
                      const vsip_vview_f *y);
void vsip_cmvprod4_f(
                      const vsip_cmview_f *A,
                      const vsip_cvview_f *x,
                      const vsip_cvview_f *y);
```

Arguments

`*A`	View of input 4 by 4 matrix.
`*x`	View of input vector of length 4.
`*y`	View of output vector of length 4.

Return Value

None.

Restrictions

The result vector view may not overlap either input matrix/vector view.

Errors

The arguments must conform to the following:

1. All objects must be valid.
2. The matrix and vectors **A**, **x**, and **y** must be conformant.
3. The input and output matrix/vector views must not overlap.

Notes/References

These functions may be implemented as macros.

Examples

See Also

`vsip_dmvprod_f`, `vsip_dmvprod3_f`, and `vsip_dvmprod_f`

vsip_*d*mvprod_*p* Matrix Vector Product

Calculate a matrix - vector product.

Functionality

Computes product of an *M* by *N* matrix and a vector.

$$\mathbf{y} \leftarrow \mathbf{A}\mathbf{x}$$

$$y_i \leftarrow \sum_{j=0}^{N-1} a_{i,j} x_j \text{, for } i = 0, 1, \ldots, M\text{-}1$$

Prototypes

```
void vsip_mvprod_f(
                      const vsip_mview_f *A,
                      const vsip_vview_f *x,
                      const vsip_vview_f *y);
void vsip_cmvprod_f(
                      const vsip_cmview_f *A,
                      const vsip_cvview_f *x,
                      const vsip_cvview_f *y);
```

Arguments

`*A`	View of input *M* by *N* matrix.
`*x`	View of input vector of length *N*.
`*y`	View of output vector of length *M*.

Return Value

None.

Restrictions

The result vector view may not overlap either input matrix/vector view.

Errors

The arguments must conform to the following:

1. All objects must be valid.
2. The matrix and vectors **A**, **x**, and **y** must be conformant.
3. The input and output matrix/vector views must not overlap.

Notes/References

Examples

See Also

`vsip_dmvprod3_f`, `vsip_dmvprod4_f`, and `vsip_dvmprod_f`

vsip_*d*mtrans_*p* Matrix Transpose

Transpose a matrix

Functionality

Returns the N by M matrix **C**, which is the transpose of an M by N matrix **A**.

$$\mathbf{C} \leftarrow \mathbf{A}^T$$

Prototypes

```
void vsip_mtrans_f(
                    const vsip_mview_f *A,
                    const vsip_mview_f *C);
void vsip_cmtrans_f(
                    const vsip_cmview_f *A,
                    const vsip_cmview_f *C);
```

Arguments

`*A`	View of input M by N matrix.
`*C`	View of output N by M matrix.

Return Value

None.

Restrictions

If the matrix **A** is square, the transpose is in place if **A** and **C** resolve to the same object, otherwise **A** and **C** must not overlap.

Errors

The arguments must conform to the following:

1. All objects must be valid.
2. The matrices **A**, and **C** must be conformant.
3. If the matrix is not square, $M \neq N$, the input and output matrix views must not overlap.
4. If the matrix is square, $M = N$, the input and output views must be identical views of the same block, or must not overlap.

Notes/References

Examples

See Also

`vsip_cmherm_f`, and `vsip_dgems_f`

vsip_*d*vdot_*p* Vector Dot Product

Compute the inner (dot) product of two vectors.

Functionality

Compute the dot product

$r \leftarrow \mathbf{a}^T \mathbf{b}$.

$$r \leftarrow \sum_{j=0}^{N-1} a_j \cdot b_j$$

Prototypes

```
vsip_scalar_f vsip_vdot_f(
                      const vsip_vview_f *a,
                      const vsip_vview_f *b);
vsip_cscalar_f vsip_cvdot_f(
                      const vsip_cvview_f *a,
                      const vsip_cvview_f *b);
```

Arguments

`*a`	View of input vector.
`*b`	View of input vector.

Return Value

For real input vectors this function returns a real scalar of the same precision as the input vectors.

For complex input vectors, this function returns a complex scalar of the same precision as the input vectors.

Restrictions

Overflow may occur. The result of overflow is implementation specific.

Errors

The arguments must conform to the following:

1. Arguments for input must be the same size.
2. All view objects must be valid.

Notes/References

Examples

```
/* Example of vector dot product */
#include<stdio.h>
#include "vsip.h"
#define L 7                   / *length */
int main()
{
    int i;
    vsip_vview_d*  dataRe;
    vsip_vview_d*  dataIm;
    vsip_cvview_d* cvectorLeft;
    vsip_cvview_d* cvectorRight;
    vsip_cscalar_d cdotpr,cLeft,cRight;

    vsip_init((void *)0);
    dataRe = vsip_vcreate_d(L, VSIP_MEM_NONE);
```

```
    dataIm = vsip_vcreate_d(L, VSIP_MEM_NONE);
    cvectorLeft = vsip_cvcreate_d(L, VSIP_MEM_NONE);
    cvectorRight = vsip_cvcreate_d(L, VSIP_MEM_NONE);
    vsip_vramp_d(1.0, 1.0 , dataRe);
    vsip_vramp_d(1.0, -2.0/(double)(L-1), dataIm);
    vsip_vcmplx_d(dataRe, dataIm, cvectorLeft);
    vsip_vcmplx_d(dataIm, dataRe, cvectorRight);
    /* do a real vector dot product and print the data and results*/
    for(i=0; i<L-1; i++)
        printf("%7.4f * %7.4fi +\n",
                  vsip_vget_d(dataRe,i),
                  vsip_vget_d(dataIm,i));
    printf("%7.4f * %7.4fi = %7.4f\n\n",
                  vsip_vget_d(dataRe,i),vsip_vget_d(dataIm,i),
                  vsip_vdot_d(dataRe,dataIm));
    /* do a complex vector dot product and print the data and results*/
    cdotpr = vsip_cvdot_d(cvectorLeft,cvectorRight);
    for(i=0; i<L-1; i++){
        cLeft = vsip_cvget_d(cvectorLeft, i);
        cRight = vsip_cvget_d(cvectorRight, i);
        printf("(%7.4f + %7.4fi) * (%7.4f + %7.4fi) +\n",
                  vsip_real_d(cRight),vsip_imag_d(cRight),
                  vsip_real_d(cLeft), vsip_imag_d(cLeft));
    }
    cLeft = vsip_cvget_d(cvectorLeft, L-1);
    cRight = vsip_cvget_d(cvectorRight, L-1);
    printf("(%7.4f + %7.4fi) * (%7.4f + %7.4fi) = "
            "(%7.4f + %7.4fi)\n", vsip_real_d(cRight),vsip_imag_d(cRight),
            vsip_real_d(cLeft), vsip_imag_d(cLeft),
            vsip_real_d(cdotpr),vsip_imag_d(cdotpr));
    /* destroy the vector views and any associated blocks */
    vsip_blockdestroy_d(vsip_vdestroy_d(dataRe));
    vsip_blockdestroy_d(vsip_vdestroy_d(dataIm));
    vsip_cblockdestroy_d(vsip_cvdestroy_d(cvectorLeft));
    vsip_cblockdestroy_d(vsip_cvdestroy_d(cvectorRight));
    vsip_finalize((void *)0);
    return 0;
}
/* output */
/* 1.0000 *  1.0000i +
   2.0000 *  0.6667i +
   3.0000 *  0.3333i +
   4.0000 *  0.0000i +
   5.0000 * -0.3333i +
   6.0000 * -0.6667i +
   7.0000 * -1.0000i = -9.3333
   ( 1.0000 +  1.0000i) * ( 1.0000 +  1.0000i) +
   ( 0.6667 +  2.0000i) * ( 2.0000 +  0.6667i) +
   ( 0.3333 +  3.0000i) * ( 3.0000 +  0.3333i) +
   ( 0.0000 +  4.0000i) * ( 4.0000 +  0.0000i) +
   (-0.3333 +  5.0000i) * ( 5.0000 + -0.3333i) +
   (-0.6667 +  6.0000i) * ( 6.0000 + -0.6667i) +
   (-1.0000 +  7.0000i) * ( 7.0000 + -1.0000i) = ( 0.0000 + 143.1111i) */
```

See Also

vsip_cvjdot.

vsip_*d*vmprod_*p* Vector Matrix Product

Calculate a vector - matrix product.

Functionality

Computes product of a vector and an *M* by *N* matrix.

$$\mathbf{y} \leftarrow \mathbf{x}^T\mathbf{A}$$

$$y_i \leftarrow \sum_{j=0}^{M-1} a_{j,i} x_j \text{, for } i = 0, 1, \ldots, M\text{-}1$$

Prototypes

```
void vsip_vmprod_f(
                      const vsip_vview_f *x,
                      const vsip_mview_f *A,
                      const vsip_vview_f *y);
void vsip_cvmprod_f(
                      const vsip_cvview_f *x,
                      const vsip_cmview_f *A,
                      const vsip_cvview_f *y);
```

Arguments

`*x`	View of input vector of length *M*.
`*A`	View of input *M* by *N* matrix.
`*y`	View of output vector of length *N*.

Return Value

None.

Restrictions

The result vector view may not overlap either input matrix/vector view.

Errors

The arguments must conform to the following:

1. All objects must be valid.
2. The matrix and vectors **A**, **x**, and **y** must be conformant.
3. The input and output matrix/vector views must not overlap.

Notes/References

Examples

See Also

`vsip_dmvprod_f`, `vsip_dmvprod3_f`, and `vsip_dmvprod4_f`

vsip_*d*vouter_*p* Vector Outer Product

Calculate the outer product of two vectors.

Functionality

If **x** and **y** are vectors of length *M* and *N* respectively, then this function computes the scalar multiple of an outer product of **x** and **y**. That is,

$$\mathbf{C} \leftarrow \alpha \cdot \mathbf{x}\mathbf{y}^T$$

$$C \leftarrow \alpha \cdot \begin{bmatrix} x_0 y_0 & x_0 y_1 & \cdots & x_0 y_{N-1} \\ x_1 y_0 & x_1 y_1 & \cdots & x_1 y_{N-1} \\ \vdots & \vdots & \ddots & \vdots \\ x_{M-1} y_0 & x_{M-1} y_1 & \cdots & x_{M-1} y_{N-1} \end{bmatrix}$$

If **x** and **y** are complex vectors, then this function computes

$$\mathbf{C} \leftarrow \alpha \cdot \mathbf{x}\mathbf{y}^H$$

$$C \leftarrow \alpha \cdot \begin{bmatrix} x_0 y_0^* & x_0 y_1^* & \cdots & x_0 y_{N-1}^* \\ x_1 y_0^* & x_1 y_1^* & \cdots & x_1 y_{N-1}^* \\ \vdots & \vdots & \ddots & \vdots \\ x_{M-1} y_0^* & x_{M-1} y_1^* & \cdots & x_{M-1} y_{N-1}^* \end{bmatrix}$$

Prototypes

```
void vsip_vouter_f(
                    vsip_scalar_f alpha,
                    const vsip_vview_f *x,
                    const vsip_vview_f *y,
                    const vsip_mview_f *C);
void vsip_cvouter_f(
                    vsip_cscalar_f alpha,
                    const vsip_cvview_f *x,
                    const vsip_cvview_f *y,
                    const vsip_cmview_f *C);
```

Arguments

`alpha`	(Real/Complex) scalar.
`*x`	View of input vector of length *M*.
`*y`	View of input vector of length *N*.
`*C`	View of output *M* by *N* matrix.

Return Value

None.

Restrictions

The result matrix view may not overlap either input vector view.

Errors

The arguments must conform to the following:

1. All objects must be valid.
2. The vectors and matrix **x**, **y**, and **C** must be conformant
3. The output matrix view and the input vector views must not overlap.

Notes/References

Examples

See Also

`vsip_`*`ds`*`kron_`*`f`*

Special Linear System Solvers

`vsip_`***`d`***`covsol_`***`p`***	Solve Covariance System
`vsip_`***`d`***`llsqsol_`***`p`***	Solve Linear Least Squares Problem
`vsip_`***`d`***`toepsol_`***`p`***	Solve Toeplitz System

vsip_*d*covsol_*p* Solve Covariance System

Solve a covariance linear system problem.

Functionality

Solves a covariance linear system problem,

$\mathbf{A}^T\mathbf{AX} = \mathbf{B}$,

or

$\mathbf{A}^H\mathbf{AX} = \mathbf{B}$,

where **A** is a matrix of order *M* by *N* with rank *N*, $M \geq N$, and **B** is a matrix of order *N* by *K*.

Prototypes

```
int vsip_covsol_f(
                    const vsip_mview_f *A,
                    const vsip_mview_f *XB);
int vsip_ccovsol_f(
                    const vsip_cmview_f *A,
                    const vsip_cmview_f *XB);
```

Arguments

`*A`	On entry, view of input matrix **A,** of size *M* by *N*, $M \geq N$.
`*XB`	View of output **X**/input matrix **B**, of size *N* by *K*.

Return Value

0 if successful

-1 if memory allocation failure

Positive if **A** does not have full column rank, rank(**A**) = *N*

Restrictions

The matrix **A** may be overwritten.

Errors

The input and output/input objects must conform to the following:

1. All objects must be valid.
2. The matrices **A** and **XB** must be conformant and must not overlap.

Notes/References

This function may allocate and free temporary workspace, which may result in non-deterministic execution time. The more general QR routines may be used to solve a covariance problem and they support explicit creation and destruction.

The matrix **A** is assumed to be of full rank. This property is not checked. A positive return value indicates that an error occurred and a zero pivot element was encountered.

Examples

See Also

Section – Overdetermined Linear System

vsip_*d*llsqsol_*p* Solve Linear Least Squares Problem

Solve a linear least squares problem.

Functionality

Solves the linear least squares problem,

$$\min_x \|AX - B\|_2$$

where **A** is a matrix of order *M* by *N* with rank *N*, $M \geq N$, and **B** is a matrix of order *M* by *K*.

Prototypes

```
int vsip_llsqsol_f(
                    const vsip_mview_f *A,
                    const vsip_mview_f *XB);
int vsip_cllsqsol_f(
                    const vsip_cmview_f *A,
                    const vsip_cmview_f *XB);
```

Arguments

`*A`	On entry, view of input matrix **A,** of size *M* by *N*, $M \geq N$.
`*XB`	On entry view of input matrix **B**, of size *M* by *K*. The view is const; however on exit the first *N* rows starting at index zero are the output data. Contents of the view starting at index *N* are implementation dependent.

Return Value

0 if successful

-1 if memory allocation failure

Positive if **A** does not have full column rank, rank(**A**) = *N*

Restrictions

The matrix **A** may be overwritten.

Errors

The input and input/output objects must conform to the following:

1. All objects must be valid.
2. The matrices **A** and **XB** must be conformant.

Notes/References

This function may allocate and free temporary workspace, which may result in non-deterministic execution time. The more general QR routines may be used to solve a linear least squares problem and they support explicit creation and destruction.

The matrix **A** is assumed to be of full rank. This property is not checked. A positive return value indicates the matrix did not have full column rank and the algorithm failed to be completed.

Since the output data length may be smaller than the input data length it is recomended that a subview of the input vector be created which defines a vector view of the output data.

Examples

See Also

Section – Overdetermined Linear System

vsip_*d*toepsol_*p* Solve Toeplitz System

Solve a real symmetric or complex Hermitian positive definite Toeplitz linear system.

Functionality

Solve a real symmetric positive definite *N* by *N* Toeplitz linear system, **Tx** = **b**, where,

$$T = \begin{bmatrix} t_0 & t_1 & \cdots & t_{N-2} & t_{N-1} \\ t_1 & t_0 & t_1 & & t_{N-2} \\ \vdots & t_1 & \ddots & \ddots & \vdots \\ t_{N-2} & & \ddots & \ddots & t_1 \\ t_{N-1} & t_{N-2} & \cdots & t_1 & t_0 \end{bmatrix}$$

Solve a complex Hermitian positive definite *N* by *N* Toeplitz linear system, **Tx** = **b,** where,

$$T = \begin{bmatrix} t_0 & t_1 & \cdots & t_{N-2} & t_{N-1} \\ t_1^* & t_0 & t_1 & & t_{N-2} \\ \vdots & t_1^* & \ddots & \ddots & \vdots \\ t_{N-2}^* & & \ddots & \ddots & t_1 \\ t_{N-1}^* & t_{N-2}^* & \cdots & t_1^* & t_0 \end{bmatrix}$$

We only need a vector **t**, the first row of **T** to specify the system.

Prototypes

```
int vsip_toepsol_f(
                      const vsip_vview_f *t,
                      const vsip_vview_f *b,
                      const vsip_vview_f *w,
                      const vsip_vview_f *x);
int vsip_ctoepsol_f(
                      const vsip_cvview_f *t,
                      const vsip_cvview_f *b,
                      const vsip_cvview_f *w,
                      const vsip_cvview_f *x);
```

Arguments

`*t`	View of input vector, **t**, of length *N*, the first row of the Toeplitz matrix **T**.
`*b`	View of input vector, **b**, of length *N*.
`*w`	View of vector, **w**, of length *N* used for temporary workspace.
`*x`	View of output vector, **x**, of length *N*.

Return Value

0 if successful

-1 if memory allocation failure

Positive if **T** is not positive definite

Restrictions

The result vector view may not overlap either input vector view.

Errors

The arguments must conform to the following:

1. All objects must be valid.
2. The vectors **t**, **x**, **w**, and **b** must be conformant.
3. The input vector views and output vector view must not overlap.

Notes/References

The matrix **T** is assumed to be of full rank and positive definite. This property is not checked. A positive return value indicates that an error occurred and the algorithm failed to be completed.

Examples

See Also

General Square Linear System Solver

`vsip_`***`d`***`lud_`*`p`	Matrix LU Decomposition
`vsip_`***`d`***`lud_create_`*`p`	Create LU Decomposition Object
`vsip_`***`d`***`lud_destroy_`*`p`	Destroy LUD Object
`vsip_`***`d`***`lud_getattr_`*`p`	LUD Get Attributes
`vsip_`***`d`***`lusol_`*`p`	Solve General Linear System

vsip_*d*lud_*p* LU Decomposition

Compute an LU decomposition of a square matrix **A** using partial pivoting.

Functionality

Computes the LU decomposition of a general *N* by *N* matrix **A** using partial pivoting, with either row interchanges or column interchanges.

An example of an LU decomposition is a factorization using row interchanges that has the form,

A = **PLU**

or using column interchanges has the form,

A = **LUP**,

where **P** is a permutation matrix, **L** is lower triangular, and **U** is upper triangular.

The choice of the particular factorization and row or column interchanges is implementation dependent.

Prototypes

```
int vsip_lud_f(
                    vsip_lu_f *lud,
                    const vsip_mview_f *A);
int vsip_clud_f(
                    vsip_clu_f *lud,
                    const vsip_cmview_f *A);
```

Arguments

`*lud`	Pointer to an LU decomposition object, created by `vsip_dlud_create_f`.
`*A`	On entry, view of input matrix **A**, *N* by *N*.

Return Value

Returns zero if successful. This routine will fail and return non-zero if **A** does not have full rank.

Restrictions

The matrix **A** may be overwritten by the decomposition, and the matrix **A** must not be modified as long as the factorization is required.

Errors

The input and input/output objects must conform to the following:

1. All objects must be valid.
2. The matrix **A** and the LU decomposition object must be conformant.

Notes/References

The matrix **A** is assumed to be of full rank. This property is not checked. A positive return value indicates that an error occurred and a zero pivot element was encountered.

Examples

See Also

`vsip_dlud_create_p`, `vsip_dlusol_p`, `vsip_dlud_destroy_p`, and `vsip_dlud_getattr_p`

vsip_*d*lud_create_*p* Create LU Decomposition Object

Create an LU decomposition object.

Functionality

Creates an LU decomposition object. The LU decomposition object encapsulates the information concerning the properties of the decomposition and required workspace.

The LU decomposition object is used to compute the LU decomposition of a general *N* by *N* matrix **A** using partial pivoting with either row interchanges or column interchanges.

As an example an LU decomposition, using row interchanges has the form,

A = **PLU**

or using column interchanges has the form,

A = **LUP**,

where **P** is a permutation matrix, **L** is lower triangular, and **U** is upper triangular matrix.

The choice of the particular factorization and row or column interchanges in an implementation dependent feature.

Prototypes

```
vsip_lu_f *vsip_lud_create_f(
                    vsip_length N);
vsip_clu_f *vsip_clud_create_f(
                    vsip_length N);
```

Arguments

`N`	The number of rows in the matrix **A**.

Return Value

The return value is a pointer to an LU decomposition object, or *null* if it fails.

Restrictions

Errors

The input parameter must conform to the following:

1. *N* is greater than zero.

Notes/References

Examples

See Also

`vsip_dlud_p`, `vsip_dlusol_p`, `vsip_dlud_destroy_p`, and `vsip_dlud_getattr_p`

vsip_*d*lud_destroy_*p* Destroy LUD Object

Destroy an LU decomposition object.

Functionality

Destroys (frees memory) an LU decomposition object returning *zero* on success, and non-*zero* on failure.

Prototypes

```
int vsip_lud_destroy_f(
                    vsip_lu_f *lud);
int vsip_clud_destroy_f(
                    vsip_clu_f *lud);
```

Arguments

`*lud`	Pointer to an LU decomposition object, created by `vsip_dlud_create_f`.

Return Value

Returns zero if successful.

Restrictions

Errors

The input argument must conform to the following:

1. The LU decomposition object must be valid. An argument of *null* is not an error.

Notes/References

An argument of *null* is not an error.

Examples

See Also

`vsip_dlud_create_p`, `vsip_dlud_p`, `vsip_dlusol_p`, and `vsip_dlud_getattr_p`

vsip_*d*lud_getattr_*p* LUD Get Attributes

Returns the attributes of an LU decomposition object.

Functionality

Returns the attributes of an LU decomposition object in an LU attribute structure passed by reference.

The attributes are:

```
typedef struct {
   vsip_length n;          number of rows and columns in the matrix
} vsip_dlu_attr_f;
```

Prototypes

```
void vsip_lud_getattr_f(
                     const vsip_lu_f *lud,
                     vsip_lu_attr_f *attr);
void vsip_clud_getattr_f(
                     const vsip_clu_f *lud,
                     vsip_clu_attr_f *attr);
```

Arguments

`*lud`	Pointer to an LU decomposition object, created by `vsip_dlud_create_f`.
`*attr`	Pointer to output attribute structure.

Return Value

None.

Restrictions

Errors

The arguments must conform to the following:

1. The LU decomposition object must be valid.
2. The attribute pointer must be valid – non-*null*.

Notes/References

Examples

See Also

`vsip_dlud_create_p`, `vsip_dlud_p`, `vsip_dlusol_p`, and `vsip_dlud_destroy_p`

vsip_*d*lusol_*p* Solve General Linear System

Solve a square linear system.

Functionality

Solve the following linear system

op(**A**)**X** = **B**,

where op(**A**) is one of the following

op(**A**) = **A**, op(**A**) = $\mathbf{A}^T$, or op(**A**) = $\mathbf{A}^H$,

for a general matrix **A** using the decomposition computed by the routine `vsip_dlud_f`. **A** is a matrix of order *N* by *N* with rank *N*, and **B** is a matrix of order *N* by *K*.

Prototypes

```
int vsip_lusol_f(
                    const vsip_lu_f *lud,
                    vsip_mat_op OpA,
                    const vsip_mview_f *XB);
int vsip_clusol_f(
                    const vsip_clu_f *lud,
                    vsip_mat_op OpA,
                    const vsip_cmview_f *XB);
```

Arguments

`*lud` Pointer to an LU decomposition object for the *N* by *N* matrix, **A**, computed by the routine `vsip_dlud_f`.

`OpA` Specifies the form of op(**A**).

```
typedef enum {
  VSIP_MAT_NTRANS = 0,      op(A) = A
  VSIP_MAT_TRANS = 1,       op(A) = A^T (real only)
  VSIP_MAT_HERM = 2,        op(A) = A^H (complex only)
  VSIP_MAT_CONJ = 3         Not valid
} vsip_mat_op;
```

`*XB` View of output **X**/input **B**, matrix of size *N* by *K*.

Return Value

Returns zero if successful.

Restrictions

Errors

The input and input/output objects must conform to the following:

1. All objects must be valid.
2. The matrix **XB** (**X** and **B**) and the LU decomposition object must be conformant.
3. `OpA` must be valid.
   Real: `OpA ∈ {VSIP_MAT_NTRANS, VSIP_MAT_TRANS}`
   Complex: `OpA ∈ {VSIP_MAT_NTRANS, VSIP_MAT_HERM }`

Notes/References

It is okay to call `vsip_dlusol_f` after `vsip_dlud_f` fails. This will result in a non-zero unsuccessful return value.

Examples

See Also

`vsip_`*d*`lud_create_`*p*, `vsip_`*d*`lud_`*p*, `vsip_`*d*`lud_destroy_`*p*, and `vsip_`*d*`lud_getattr_`*p*

Symmetric Positive Definite Linear System Solver

`vsip_dchold_p`	Matrix Cholesky Decomposition
`vsip_dchold_create_p`	Create Cholesky Decomposition Object
`vsip_dchold_destroy_p`	Destroy CHOLD Object
`vsip_dchold_getattr_p`	CHOLD Get Attributes
`vsip_dcholsol_p`	Solve SPD Linear System

vsip_*d*chold_*p* Cholesky Decomposition

Compute a Cholesky decomposition of a symmetric (Hermitian) positive definite matrix **A**.

Functionality

The Cholesky decomposition of a symmetric (Hermitian) positive definite *N* by *N* matrix **A** is given by

$\mathbf{A} = \mathbf{L}\mathbf{L}^T$ ($\mathbf{A} = \mathbf{L}\mathbf{L}^H$), where **L** is a lower triangular matrix,

or

$\mathbf{A} = \mathbf{R}^T\mathbf{R}$ ($\mathbf{A} = \mathbf{R}^H\mathbf{R}$), where **R** is a upper triangular matrix,

The particular type of factorization is an implementation dependent feature. There is not a utility function for accessing the factors.

Prototypes

```
int vsip_chold_f(
                    vsip_chol_f *chold,
                    const vsip_mview_f *A);
int vsip_cchold_f(
                    vsip_cchol_f *chold,
                    const vsip_cmview_f *A);
```

Arguments

`*chold`	Pointer to a Cholesky decomposition object, created by `vsip_dchold_create_f`.
`*A`	On entry, view of input matrix **A**, *N* by *N*.

Return Value

Returns zero if successful. This routine will fail if a leading minor of **A** is not symmetric (Hermitian) positive definite and the algorithm could not be completed.

Restrictions

The matrix **A** may be overwritten by the decomposition, and the matrix **A** must not be modified as long as the decomposition is required.

Errors

The input and input/output objects must conform to the following:

1. All objects must be valid.
2. The matrix **A** and the Cholesky decomposition object must be conformant.

Notes/References

The matrix, **A**, is assumed to be symmetric (Hermitian). This property is not checked. Since VSIPL does not have a symmetric (Hermitian) object type storage for the full matrix must be specified. Only half of the matrix is referenced and modified; the other half is not modified.

Examples

See Also

`vsip_dchold_create_p`, `vsip_dcholsol_p`, `vsip_dchold_destroy_p`, and `vsip_dchold_getattr_p`

vsip_*d*chold_create_*p* Create Cholesky Decomposition Object

Create a Cholesky decomposition object.

Functionality

Create a Cholesky decomposition (CHOLD) object. The Cholesky decomposition object encapsulates the information concerning the properties of the decomposition and required workspace.

The Cholesky decomposition object is used to compute the Cholesky decomposition of a symmetric positive definite N by N matrix $\mathbf{A}$.

The Cholesky decomposition of a symmetric (Hermitian) positive definite N by N matrix $\mathbf{A}$ is given by

$\mathbf{A} = \mathbf{L}\mathbf{L}^T$ ($\mathbf{A} = \mathbf{L}\mathbf{L}^H$), where $\mathbf{L}$ is a lower triangular matrix,

or

$\mathbf{A} = \mathbf{R}^T\mathbf{R}$ ($\mathbf{A} = \mathbf{R}^H\mathbf{R}$), where $\mathbf{R}$ is a upper triangular matrix,

The particular type of factorization is an implementation dependent feature. There is not a utility function for accessing the factors.

Prototypes

```
vsip_chol_f *vsip_chold_create_f(
                    vsip_mat_uplo uplo,
                    vsip_length N);
vsip_cchol_f *vsip_cchold_create_f(
                    vsip_mat_uplo uplo,
                    vsip_length N);
```

Arguments

`N`	The number of row or columns in the input matrix.
`uplo`	Specifies if the upper or lower triangular half of the matrix is stored.

```
typedef enum {
  VSIP_TR_LOW = 0,   only the lower half of the matrix is stored
  VSIP_TR_UPP = 1    only the upper half of the matrix is stored
} vsip_mat_uplo;
```

Return Value

The return value is a pointer to a Cholesky decomposition object, or *null* if it fails.

Restrictions

Errors

The input parameters must conform to the following:

1. N is greater than zero.
2. `uplo` is valid: `uplo` ∈ {`VSIP_TR_LOW`, `VSIP_TR_UPP`}

Notes/References

Examples

See Also

`vsip_dchold_p`, `vsip_dcholsol_p`, `vsip_dchold_destroy_p`, and `vsip_dchold_getattr_p`

vsip_*d*chold_destroy_*p* Destroy CHOLD Object

Destroy a Cholesky decomposition object.

Functionality

Destroy (free memory) a Cholesky decomposition object returning *zero* on success, and non-*zero* on failure.

Prototypes

```
int vsip_chold_destroy_f(
                     vsip_chol_f *chold);
int vsip_cchold_destroy_f(
                     vsip_cchol_f *chold);
```

Arguments

`*chold` Pointer to a Cholesky decomposition object, created by `vsip_dchold_create_f`.

Return Value

Returns *zero* on success, and non-*zero* on failure.

Restrictions

Errors

The input object must conform to the following:

1. The Cholesky decomposition object must be valid. An argument of *null* is not an error.

Notes/References

An argument of *null* is not an error.

Examples

See Also

`vsip_dchold_p`, `vsip_dchold_create_p`, `vsip_dcholsol_p`, and `vsip_dchold_getattr_p`

vsip_*d*chold_getattr_*p* CHOLD Get Attributes

Returns the attributes of a Cholesky decomposition object.

Functionality

Returns the attributes of a Cholesky decomposition object in structure passed by reference.

The attributes are:

```
typedef struct {
  vsip_mat_uplo uplo;        Upper or lower triangular matrix
  vsip_length n;             Number of rows and columns in the matrix
} vsip_dchol_attr_f;
```

Prototypes

```
void vsip_chold_getattr_f(
                      const vsip_chol_f *chold,
                      vsip_chol_attr_f *attr);
void vsip_cchold_getattr_f(
                      const vsip_cchol_f *chold,
                      vsip_cchol_attr_f *attr);
```

Arguments

`*chold`	Pointer to a Cholesky decomposition object, created by `vsip_dchold_create_f`.
`*attr`	Pointer to output attribute structure.

Return Value

None.

Restrictions

Errors

The input and output arguments must conform to the following:

1. The Cholesky decomposition object must be valid.
2. The attribute pointer must be valid – non-*null*.

Notes/References

Examples

See Also

`vsip_dchold_p`, `vsip_dchold_create_p`, `vsip_dcholsol_p`, and `vsip_dchold_destroy_p`

vsip_*d*cholsol_*p* Solve SPD Linear System

Solve a symmetric (Hermitian) positive definite linear system.

Functionality

Solve the following linear system

AX = **B**,

for a symmetric (Hermitian) positive definite matrix **A** using the decomposition computed by the routine `vsip_dchold_f`. **A** is a matrix of order *N* by *N*, and **B** is a matrix of order *N* by *K*.

Prototypes

```
int vsip_cholsol_f(
                    const vsip_chol_f *chold,
                    const vsip_mview_f *XB);
int vsip_ccholsol_f(
                    const vsip_cchol_f *chold,
                    const vsip_cmview_f *XB);
```

Arguments

`*chold`	Pointer to a Cholesky decomposition object for the *N* by *N* matrix, **A**, computed by the routine `vsip_dchold_f`.
`*XB`	View of input **B**/output **X**, matrix of size *N* by *K*.

Return Value

Returns zero if successful.

Restrictions

Errors

The input and input/output objects must conform to the following:

1. All objects must be valid.
2. The matrix **XB** (**X** and **B**) and the Cholesky decomposition object must be conformant.

Notes/References

It is okay to call `vsip_dcholsol_f` after `vsip_dchold_f` fails. This will result in a non-zero, unsuccessful, return value.

Examples

See Also

`vsip_dchold_p`, `vsip_dchold_create_p`, `vsip_dchold_destroy_p`, and `vsip_dchold_getattr_p`

Over-determined Linear System Solver

`vsip_`***`d`***`qrd_`***`p`***	Matrix QR Decomposition
`vsip_`***`d`***`qrd_create_`***`p`***	Create QR Decomposition Object
`vsip_`***`d`***`qrd_destroy_`***`p`***	Destroy QRD Object
`vsip_`***`d`***`qrd_getattr_`***`p`***	QRD Get Attributes
`vsip_`***`d`***`qrdprodq_`***`p`***	Product with Q from QR Decomposition
`vsip_`***`d`***`qrdsolr_`***`p`***	Solve Linear System Based on R from QR Dec.
`vsip_`***`d`***`qrsol_`***`p`***	Solve Covariance or LLSQ System

vsip_*d*qrd_*p* QR Decomposition

Compute a QR decomposition of a matrix .

Functionality

Compute a QR decomposition of a matrix. It is a requirement that $M \geq N$. The QR decomposition of an M by N matrix **A** is given by

$$\mathbf{A} = \mathbf{QR},$$

where **Q** is an M by N orthogonal matrix ($\mathbf{Q}^T\mathbf{Q} = \mathbf{I}$) or **Q** is an M by N unitary matrix ($\mathbf{Q}^H\mathbf{Q} = \mathbf{I}$) and **R** is an upper triangular matrix. If **A** has full rank, then **R** is a nonsingular matrix. This routine does not perform any column interchanges.

Prototypes

```
int vsip_qrd_f(
                    vsip_qr_f *qrd,
                    const vsip_mview_f *A);
int vsip_cqrd_f(
                    vsip_cqr_f *qrd,
                    const vsip_cmview_f *A);
```

Arguments

`*qrd`	Pointer to a QR decomposition object, created by `vsip_dqrd_create_f`.
`*A`	On entry, view of input matrix **A**, M by N.

Return Value

Returns zero on success. This routine will fail and return non-zero if **A** does not have full column rank, rank(**A**) = N.

Restrictions

The matrix **A** may be overwritten by the decomposition, and matrix **A** must not be modified as long as the factorization is required.

Errors

The input and input/output objects must conform to the following:

1. All objects must be valid.
2. The matrix **A** and the QR decomposition object must be conformant.

Notes/References

The matrix **A** is assumed to be of full rank. This property is not checked. A positive return value indicates that an error occurred and an exactly zero diagonal element of R was encountered.

Examples

See Also

`vsip_dqrd_create_p`, `vsip_dqrsol_p`, `vsip_dqrdprodq_p`, `vsip_dqrdsolr_p`, `vsip_dqrd_destroy_p`, and `vsip_dqrd_getattr_p`

vsip_*d*qrd_create_*p* Create QR Decomposition Object

Create a QR decomposition object.

Functionality

Create a QR decomposition (QRD) object. The QR decomposition object encapsulates the information concerning the properties of the decomposition and required workspace.

For example, given that $M \geq N$, the QR decomposition of an M by N matrix **A** is given by

$$\mathbf{A} = \mathbf{QR},$$

where **Q** is an M by N orthogonal matrix ($\mathbf{Q}^T\mathbf{Q} = \mathbf{I}$) or **Q** is an M by N unitary matrix ($\mathbf{Q}^H\mathbf{Q} = \mathbf{I}$) and **R** is a N by N upper triangular matrix. This type of decomposition is often referred to as the "skinny" QR factorization. If **A** has full rank, then **R** is a nonsingular matrix. This routine does not perform any column interchanges.

The **R** matrix will be generated and retained for later usage. However, there is a flag to indicate if the **Q** matrix is retained. It is an option to either retain the "skinny" Q or a full Q, where Q is an M by M orthogonal (unitary) matrix.

Prototypes

```
vsip_qr_f *vsip_qrd_create_f(
                     vsip_length M,
                     vsip_length N,
                     vsip_qrd_qopt qopt
                     );
vsip_cqr_f *vsip_cqrd_create_f(
                     vsip_length M,
                     vsip_length N
                     vsip_qrd_qopt qopt
                     );
```

Arguments

`M` The number of rows for the input matrix **A.**

`N` The number of columns for the input matrix **A.**

`qopt` Indicates if the matrix **Q** is retained.

```
typedef enum {
  VSIP_QRD_NOSAVEQ = 0,        Do not save Q
  VSIP_QRD_SAVEQ = 1,          Save full Q
..VSIP_QRD_SAVEQ1 = 2          Save skinny Q
} vsip_qrd_qopt;
```

Return Value

The return value is a pointer to a QR decomposition object, or *null* if it fails.

Restrictions

Errors

The input arguments must conform to the following:

1. M and N positive with $N \leq M$.
2. `Qopt` is valid: `qopt` ∈{`VSIP_QRD_NOSAVEQ, VSIP_QRD_SAVEQ, VSIP_QRD_SAVEQ1`}.

Notes/References

Examples

See Also

`vsip_`*d*`qrd_`*p*, `vsip_`*d*`qrsol_`*p*, `vsip_`*d*`qrdprodq_`*p*, `vsip_`*d*`qrdsolr_`*p*, `vsip_`*d*`qrd_destroy_`*p*, and `vsip_`*d*`qrd_getattr_`*p*

vsip_*d*qrd_destroy_*p* Destroy QRD Object

Destroy a QR decomposition object.

Functionality

Destroy (free memory) a QR decomposition object returning *zero* on success, and non-*zero* on failure.

Prototypes

```
int vsip_qrd_destroy_f(
                    vsip_qr_f *qrd);
int vsip_cqrd_destroy_f(
                    vsip_cqr_f *qrd);
```

Arguments

`*qrd`	Pointer to a QR decomposition object, created by `vsip_dqrd_create_f`.

Return Value

Returns *zero* on success, and non-*zero* on failure.

Restrictions

Errors

The input object must conform to the following:

1. The QR decomposition object must be valid. An argument of *null* is not an error.

Notes/References

An argument of *null* is not an error.

Examples

See Also

`vsip_dqrd_p`, `vsip_dqrd_create_p`, `vsip_dqrsol_p`, `vsip_dqrdprodq_p`, `vsip_dqrdsolr_p`, and `vsip_dqrd_getattr_p`

vsip_*d*qrd_getattr_*p* QRD Get Attributes

Returns the attributes of a QR decomposition object.

Functionality

Returns the attributes of a QR decomposition object in structure passed by reference.

The attributes are:

```
typedef struct {
  vsip_length m;          The number of rows for the input matrix A
  vsip_length n;          The number of columns for the input matrix A
  vsip_qrd_opt Qopt;      Indicates if the matrix Q is retained or not
} vsip_dqr_attr_f;
```

```
typedef enum {
  VSIP_QRD_NOSAVEQ = 0,     Do not save Q
  VSIP_QRD_SAVEQ = 1,       Save Q
  VSIP_QRD_SAVEQ1 = 2       Save Skinny Q
} vsip_qrd_qopt;
```

Prototypes

```
void vsip_qrd_getattr_f(
                    const vsip_qr_f *qrd,
                    vsip_qr_attr_f *attr);
void vsip_cqrd_getattr_f(
                    const vsip_cqr_f *qrd,
                    vsip_cqr_attr_f *attr);
```

Arguments

`*qrd`	Pointer to a QR decomposition object, created by `vsip_dqrd_create_f`.
`*attr`	Pointer to output attribute structure.

Return Value

None.

Restrictions

Errors

The arguments must conform to the following:

1. The QR decomposition object must be valid.
2. The attribute pointer must be valid – non-*null*.

Notes/References

Examples

See Also

`vsip_dqrd_p`, `vsip_dqrd_create_p`, `vsip_dqrsol_p`, `vsip_dqrdprodq_p`, `vsip_dqrdsolr_p`, and `vsip_dqrd_destroy_p`

vsip_*d*qrdprodq_*p* Product with Q from QR Decomposition

Multiply a matrix by the matrix **Q** from a QR decomposition.

Functionality

This function overwrites a *R* by *S* matrix **C** with

	MAT_LSIDE	MAT_RSIDE
MAT_NTRANS	Q C	C Q
MAT_TRANS	Q^T C	C Q^T
MAT_HERM	Q^H C	C Q^H

Where the matrix **Q** was generated by the routine `vsip_dqrd_f`. If a *M* by *N* matrix was the input matrix for the function `vsip_dqrd_f`, then **Q** is either a *M* by *M* or *M* by *N* matrix, depending which option was used to generate **Q**.

If Q was computed using the `QRD_SAVEQ1` option, then the following table lists the possible dimensions of the matrix C before and after this operation:

	Input		Output	
	MAT_LSIDE	**MAT_RSIDE**	**MAT_LSIDE**	**MAT_RSIDE**
MAT_NTRANS	*N* by *S*	*R* by *M*	*M* by *S*	*R* by *N*
MAT_TRANS	*M* by *S*	*R* by *N*	*N* by *S*	*R* by *M*
MAT_HERM	*M* by *S*	*R* by *N*	*N* by *S*	*R* by *M*

Given that $M \geq N$, then for some options the result of this operation is a matrix that is larger than the input matrix. The matrix view object used for the input/output data is const. The first element of the input and the first element of the output are stored at element location (0,0) of the input/output matrix. Other elements are stored in their natural location in the block determined by the row stride and column stride of the input/output view.

If Q was computed using the `QRD_SAVEQ` option, then the following table lists the possible dimensions of the matrix C before and after this operation:

	Input & Output	
	MAT_LSIDE	**MAT_RSIDE**
MAT_NTRANS	*M* by *S*	*R* by *M*
MAT_TRANS	*M* by *S*	*R* by *M*
MAT_HERM	*M* by *S*	*R* by *M*

Prototypes

```
int vsip_qrdprodq_f(
                    const vsip_qr_f *qrd,
                    vsip_mat_op opQ,
                    vsip_mat_side apQ,
                    const vsip_mview_f *C);
int vsip_cqrdprodq_f(
                    const vsip_cqr_f *qrd,
                    vsip_mat_op opQ,
                    vsip_mat_side apQ,
                    const vsip_cmview_f *C);
```

Arguments

`*qrd` Pointer to a QR decomposition object, generated by `vsip_dqrd_f`.

`opQ` Specifies the form of op(**Q**).

```
typedef enum {
  VSIP_MAT_NTRANS = 0,
  VSIP_MAT_TRANS = 1,
  VSIP_MAT_HERM = 2
} vsip_mat_op;
```

	Op(**Q**) = **Q**	
	Op(**Q**) = $\mathbf{Q}^T$	(real only)
	Op(**Q**) = $\mathbf{Q}^H$	(complex only)

`apQ` Indicates if op(**Q**) is applied on the left or right of **C**.

```
typedef enum {
  VSIP_MAT_LSIDE = 0,
  VSIP_MAT_RSIDE = 1
} vsip_mat_side;
```

`*C` On entry, view of input matrix **C**, *R* by *S*. On output the data is stored in natural order in the block determined by the offset, row stride, and column stride of the input matrix view. See restrictions below.

Return Value

Returns *zero* if successful.

Restrictions

Since the output data space may be larger than the input data space it is required that the input data view allow storage in the block for the output data. This means the row stride and column stride must be calculated to accommodate the larger data space, whether it be input or output.

Errors

The arguments must conform to the following:

1. All objects must be valid.
2. `opQ` is valid:
   Real: `opQ` ∈{`VSIP_MAT_NTRANS, VSIP_MAT_TRANS`}
   Complex: `opQ` ∈{`VSIP_MAT_NTRANS, VSIP_MAT_HERM`}
3. `apQ` is valid: `apQ` ∈{ `VSIP_MAT_LSIDE, VSIP_MAT_RSIDE`}
4. The matrix **C** and the QR decomposition object must be conformant.
5. The QR decomposition object must have specified retaining the **Q** matrix when it was created.

Notes/References

It is okay to call `vsip_`*`d`*`qrdprodq_`*`f`* after `vsip_`*`d`*`qrd_`*`f`* fails. This will result in a non-zero, unsuccessful, return value.

One way to ensure the input/output data space is proper is to calculate the size of the output data space either using the tables under functionality above, or directly given knowledge of the input matrix sizes. If the output data space is larger than the input data space create a matrix view large enough to hold the output data. Create a subview of this with index offset at (0,0) of proper size to hold the input data. The new (sub) view is then the input to the function, and the original view will hold the output data.

Examples

See Also

`vsip_`*`d`*`qrd_`*`p`*, `vsip_`*`d`*`qrd_create_`*`p`*, `vsip_`*`d`*`qrsol_`*`p`*, `vsip_`*`d`*`qrdsolr_`*`p`*, `vsip_`*`d`*`qrd_destroy_`*`p`*, and `vsip_`*`d`*`qrd_getattr_p`

vsip_*d*qrdsolr_*p* Solve Linear System Based on R from QR Dec.

Solve linear system based on the matrix **R**, from QR decomposition of the matrix **A**.

Functionality

Solve a triangular linear system of the form,

$$\mathrm{op}(\mathbf{R})\ \mathbf{X} = \alpha \cdot \mathbf{B}$$

where op(**R**) is one of the following

$$\mathrm{op}(\mathbf{R}) = \mathbf{R},\ \mathrm{op}(\mathbf{R}) = \mathbf{R}^T,\ \mathrm{op}(\mathbf{R}) = \mathbf{R}^H,$$

R is an upper triangular *N* by *N* matrix ,and **X** and **B** are *N* by *K* matrices.

Prototypes

```
int vsip_qrdsolr_f(
                    const vsip_qr_f *qrd,
                    vsip_mat_op OpR,
                    vsip_scalar_f alpha,
                    const vsip_mview_f *XB);
int vsip_cqrdsolr_f(
                    const vsip_cqr_f *qrd,
                    vsip_mat_op OpR,
                    vsip_cscalar_f alpha,
                    const vsip_cmview_f *XB);
```

Arguments

`*qrd` Pointer to a QR decomposition object, generated by `vsip_dqrd_f`.

`OpR` Specifies the form of op(**R**).

```
typedef enum {
  VSIP_MAT_NTRANS = 0,
  VSIP_MAT_TRANS = 1,
  VSIP_MAT_HERM = 2,
} vsip_mat_op;
```

op(**R**) = **R**
op(**R**) = $\mathbf{R}^T$ (real only)
op(**R**) = $\mathbf{R}^H$ (complex only)

`alpha` (Real/Complex) scalar.

`*XB` View of input/output matrix of size *N* by *K*.

Return Value

Returns *zero* if successful.

Restrictions

Errors

The arguments must conform to the following:

1. All of the objects must be valid.
2. `OpR` is valid:
   Real: `OpR` ∈ `{VSIP_MAT_NTRANS, VSIP_MAT_TRANS }`
   Complex: `OpR` ∈ `{VSIP_MAT_NTRANS, VSIP_MAT_HERM }`
3. The matrix **XB** (**X** and **B**) and the QR decomposition object must be conformant.

Notes/References

It is okay to call `vsip_`*`d`*`qrdsolr_`*`f`* after `vsip_`*`d`*`qrd_`*`f`* fails. This will result in a non-zero, unsuccessful, return value.

Examples

See Also

`vsip_`*`d`*`qrd_`*`p`*, `vsip_`*`d`*`qrd_create_`*`p`*, `vsip_`*`d`*`qrsol_`*`p`*, `vsip_`*`d`*`qrdprodq_`*`p`*, `vsip_`*`d`*`qrd_destroy_`*`p`*, and `vsip_`*`d`*`qrd_getattr_`*`p`*

vsip_*d*qrsol_*p* Solve Covariance or LLSQ System

Solve either a linear covariance or linear least squares problem.

Functionality

Assume that **A** is a matrix of order M by N with rank N, $M \geq N$ and **B** is a matrix of order N by K for the covariance problem or M by K for the least squares problem. This routine solves one of the following problems using the decomposition computed by the routine `vsip_dqrd_f`:

A covariance linear system problem,

$$\mathbf{A}^T\mathbf{A}\mathbf{X} = \mathbf{B},$$

or

$$\mathbf{A}^H\mathbf{A}\mathbf{X} = \mathbf{B},$$

or a linear least squares problem,

$$\min_x \|AX - B\|_2$$

Prototypes

```
int vsip_qrsol_f(
                    const vsip_qr_f *qrd,
                    vsip_qrd_prob prob,
                    const vsip_mview_f *XB);
int vsip_cqrsol_f(
                    const vsip_qr_f *qrd,
                    vsip_qrd_prob prob,
                    const vsip_cmview_f *XB);
```

Arguments

`*qrd` Pointer to QR decomposition object for the M by N matrix, **A**, computed by the routine `vsip_dqrd_f`.

`prob` Selects between the covariance and linear least squares problem.

```
typedef enum {
  VSIP_COV = 0,      Solve a covariance linear system problem
  VSIP_LLS = 1       Solve a linear least squares problem
} vsip_qrd_prob;
```

`*XB` On input view of input matrix B of size N by K for the covariance problem or M by K for the least squares problem. The view is const. The output data overwrites the input data starting at index zero. For the least squares problem elements of the input/output view starting at index N are vendor dependent on output.

Return Value

Returns *zero* if successful.

Restrictions

This routine will fail if rank(**A**) < N.

Errors

The arguments must conform to the following:

1. All objects must be valid.
2. The matrix **XB** (**X** and **B**) and the QR decomposition object must be conformant and must not overlap with the input matrix.
3. `prob` is valid: `prob` ∈ `{VSIP_COV, VSIP_LLS }`

Notes/References

It is okay to call `vsip_`*`d`*`qrsol_`*`f`* after `vsip_`*`d`*`qrd_`*`f`* fails. This will result in a non-zero, unsuccessful, return value.

Examples

See Also

`vsip_`*`d`*`qrd_`*`p`*, `vsip_`*`d`*`qrd_create_`*`p`*, `vsip_`*`d`*`qrdprodq_`*`p`*, `vsip_`*`d`*`qrdsolr_`*`p`*, `vsip_qrd_destroy_`*`p`*, and `vsip_`*`d`*`qrd_getattr_`*`p`*, `vsip_`*`d`*`llsqsol_`*`p`*, `vsip_`*`d`*`covsol_`*`p`*.

Singular Value Decomposition

This section defines the singular value decomposition (SVD) routines. Unlike other matrix decomposition routines the SVD routines do not include a linear equation solver. However, systems of linear equations can be solved by using the matrix multiplication routines `vsip_dsvdprodu_p` and `vsip_dsvdprodv_p`.

`vsip_dsvd_p`	Matrix Singular Value Decomposition
`vsip_dsvd_create_p`	Create Singular Value Decomposition Object
`vsip_dsvd_destroy_p`	Destroy SVD Object
`vsip_dsvd_getattr_p`	SVD Get Attributes
`vsip_dsvdprodu_p`	Product with U from SV Decomposition
`vsip_dsvdprodv_p`	Product with V from SV Decomposition
`vsip_dsvdmatu_p`	Return with U from SV Decomposition
`vsip_dsvdmatv_p`	Return with V from SV Decomposition

vsip_*d*svd_*p* Singular Value Decomposition

Compute the singular value decomposition of a matrix.

Functionality

Computes singular value decomposition of a matrix. The singular value decomposition of an M by N real matrix **A** is given by

$$\mathbf{A} = \mathbf{U}\mathbf{S}\mathbf{V}^T$$

where **U** is an M by M orthogonal matrix ($\mathbf{U}^T\mathbf{U} = \mathbf{I}$), **S** is an M by N zero matrix except for its min(M, N) diagonal elements, **V** is an N by N orthogonal matrix ($\mathbf{V}^T\mathbf{V} = \mathbf{I}$).

The singular value decomposition of an M by N complex matrix **A** is given by

$$\mathbf{A} = \mathbf{U}\mathbf{S}\mathbf{V}^H$$

where **U** is an M by M unitary matrix ($\mathbf{U}^H\mathbf{U} = \mathbf{I}$), **S** is an M by N zero matrix except for its min(M, N) diagonal elements, **V** is an N by N unitary matrix ($\mathbf{V}^H\mathbf{V} = \mathbf{I}$).

The diagonal elements of **S** are called 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 called the singular vectors of **A**.

Prototypes

```
int vsip_svd_f(
                    vsip_sv_f *svd,
                    const vsip_mview_f *A,
                    vsip_vview_f *s);
int vsip_csvd_f(
                    vsip_csv_f *svd,
                    const vsip_cmview_f *A,
                    vsip_vview_f *s);
```

Arguments

`*svd`	Pointer to an SVD object, created by `vsip_dsvd_create_f`.
`*A`	On entry, view of input matrix A, M by N.
`*s`	A real vector of length min(M, N), containing the output singular values of **A** in descending order.

Return Value

Returns *zero* if successful. Returns a nonzero if the algorithm failed to be completed.

Restrictions

The matrix **A** is overwritten by the decomposition, and matrix **A** must not be modified as long as the factorization is required.

Errors

The input and input/output objects must conform to the following:

1. All objects must be valid.
2. The matrix **A**, vector **s**, and the SVD object must be conformant.

Notes/References

Examples

See Also

`vsip_`*d*`svd_create_`*p*, `vsip_`*d*`svdprodu_`*p*, `vsip_`*d*`svdprodv_`*p*, `vsip_`*d*`svd_destroy_`*p*, and `vsip_`*d*`svd_getattr_`*p*

vsip_*d*svd_create_*p* Create Singular Value Decomposition Object

Create the singular value decomposition (SVD) object.

Functionality

Creates a singular value decomposition (SVD) object. The singular value decomposition of an M by N real matrix A is given by

$$\mathbf{A} = \mathbf{U}\mathbf{S}\mathbf{V}^T$$

where **U** is an M by M orthogonal matrix ($\mathbf{U}^T\mathbf{U} = \mathbf{I}$), **S** is an M by N zero matrix except for its min(M,N) diagonal elements, **V** is an N by N orthogonal matrix ($\mathbf{V}^T\mathbf{V} = \mathbf{I}$).

The singular value decomposition of an M by N complex matrix **A** is given by

$$\mathbf{A} = \mathbf{U}\mathbf{S}\mathbf{V}^H$$

where **U** is an M by M unitary matrix ($\mathbf{U}^H\mathbf{U} = \mathbf{I}$), **S** is an M by N zero matrix except for its min(M,N) diagonal elements, **V** is an N by N unitary matrix ($\mathbf{V}^H\mathbf{V} = \mathbf{I}$).

The diagonal elements of **S** are called 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 called the singular vectors of **A**.

Prototypes

```
vsip_sv_f *vsip_svd_create_f(
                    vsip_length M,
                    vsip_length N,
                    vsip_svd_uv Usave,
                    vsip_svd_uv Vsave);
vsip_csv_f *vsip_csvd_create_f(
                    vsip_length M,
                    vsip_length N
                    vsip_svd_uv Usave,
                    vsip_svd_uv Vsave);
```

Arguments

`M` The number of rows for the input matrix **A.**

`N` The number of columns for the input matrix **A.**

`Usave` Specifies the options for computing **U.**

```
typedef enum {
  VSIP_SVD_UVNOS = 0,     no columns of U are computed
  VSIP_SVD_UVFULL = 1,    all columns of U are computed
  VSIP_SVD_UVPART = 2     first min(M,N) columns of U are computed
} vsip_svd_uv;
```

`Vsave` Specifies the options for computing $\mathbf{V}^T$ ($\mathbf{V}^H$).

```
typedef enum {
  VSIP_SVD_UVNOS = 0,     no rows of V^T (V^H) are computed
  VSIP_SVD_UVFULL = 1,    all rows of V^T (V^H) are computed
  VSIP_SVD_UVPART = 2     first min(M,N) rows of V^T (V^H) are computed
} vsip_svd_uv;
```

Return Value

The return value is a pointer to an SVD object, or *null* if it fails.

Restrictions

Errors

The arguments must conform to the following:

1. *M* and *N* must be positive.
2. `Usave` and `Vsave` valid:
   `Usave, Vsave` ∈ `{VSIP_SVD_UVNOS, VSIP_SVD_UVFULL, VSIP_SVD_UVPART }`.

Notes/References

Examples

See Also

`vsip_`***d***`svd_`***p***, `vsip_`***d***`svdprodu_`***p***, `vsip_`***d***`svdprodv_`***p***, `vsip_`***d***`svd_destroy_`***p***, and `vsip_`***d***`svd_getattr_`***p***

vsip_*d*svd_destroy_*p* Destroy SVD Object

Destroy an SVD object.

Functionality

Destroy (free memory) an SVD object.

Prototypes

```
int vsip_svd_destroy_f(
                    vsip_sv_f *svd);
int vsip_csvd_destroy_f(
                    vsip_csv_f *svd);
```

Arguments

`*svd`	Pointer to an SVD object, created by `vsip_dsvd_create_f`.

Return Value

Return *zero* on success, and non-*zero* on failure.

Restrictions

Errors

The input object must conform to the following:

1. The SVD object must be valid. An argument of *null* is not an error.

Notes/References

An argument of *null* is not an error.

Examples

See Also

`vsip_dsvd_create_p`, `vsip_dsvdprodu_p`, `vsip_dsvdprodv_p`, and `vsip_dsvd_getattr_p`

vsip_*d*svd_getattr_*p* SVD Get Attributes

Returns the attributes of an SVD object.

Functionality

Returns the attributes of an SVD object in structure passed by reference.

The attributes are:

```
typedef struct {
   vsip_length m;        The number of rows for the input matrix A
   vsip_length n;        The number of columns for the input matrix A
   vsip_svd_uv Usave;    Specifies the options for computing U
   vsip_svd_uv Vsave;    Specifies the options for computing V^T (V^H)
} vsip_dsv_attr_f;
```

Where:

```
typedef enum {
  VSIP_SVD_UVNOS = 0,     no columns of U/rows of V^T (V^H) are computed
  VSIP_SVD_UVFULL = 1,    all columns of U/rows of V^T (V^H) are computed
  VSIP_SVD_UVPART = 2     first min(M,N) columns of U/rows of V^T (V^H) are computed
} vsip_svd_uv;
```

Prototypes

```
void vsip_svd_getattr_f(
                      const vsip_sv_f *svd,
                      vsip_sv_attr_f *attr);
void vsip_csvd_getattr_f(
                      const vsip_csv_f *svd,
                      vsip_csv_attr_f *attr);
```

Arguments

`*svd`	Pointer to an SVD object, created by `vsip_dsvd_create_f`.
`*attr`	Pointer to output attribute structure.

Return Value

None.

Restrictions

Errors

The arguments must conform to the following:

1. The SVD object must be valid.
2. The attribute pointer must be valid – non-*null.*

Notes/References

Examples

See Also

`vsip_dsvd_create_p`, `vsip_dsvdprodu_p`, `vsip_dsvdprodv_p`, and `vsip_dsvd_destroy_p`

vsip_*d*svdprodu_*p* Product with U from SV Decomposition

Multiply a matrix by the matrix **U** from a singular value decomposition.

Functionality

This function overwrites a *R* by *S* matrix **C** with

	MAT_LSIDE	MAT_RSIDE
MAT_NTRANS	U C	C U
MAT_TRANS	U^T C	C U^T
MAT_HERM	U^H C	C U^H

Where the matrix **U** was generated by the routine `vsip_dsvd_f`. If a *M* by *N* matrix was the input matrix for the function `vsip_dsvd_f`, then **U** is either a *M* by *M* or *M* by min(*M,N*) matrix, depending which option was used to generate **U**.

If **U** was computed using the `SVD_UVPART` option, then the following table lists the possible dimensions of the matrix **C** before and after this operation:

	Input		Output	
	MAT_LSIDE	**MAT_RSIDE**	**MAT_LSIDE**	**MAT_RSIDE**
MAT_NTRANS	min*(M,N)* by *S*	*R* by *M*	*M* by *S*	*R* by min*(M,N)*
MAT_TRANS	*M* by *S*	*R* by min*(M,N)*	min*(M,N)* by *S*	*R* by *M*
MAT_HERM	*M* by *S*	*R* by min*(M,N)*	min*(M,N)* by *S*	*R* by *M*

For some options, the result of this operation is a matrix that is larger than the input matrix. The matrix view object used for the input/output data is const. The first element of the input and the first element of the output are stored at element location (0,0) of the input/output matrix. Other elements are stored in their natural location in the block determined by the row stride and column stride of the input/output view.

If **U** was computed using the `SVD_UVFULL` option, then the following table lists the possible dimensions of the matrix **C** before and after this operation:

	Input & Output	
	MAT_LSIDE	**MAT_RSIDE**
MAT_NTRANS	*M* by *S*	*R* by *M*
MAT_TRANS	*M* by *S*	*R* by *M*
MAT_HERM	*M* by *S*	*R* by *M*

Prototypes

```
int vsip_svdprodu_f(
                    const vsip_sv_f *svd,
                    vsip_mat_op OpU,
                    vsip_mat_side ApU,
                    const vsip_mview_f *C);
int vsip_csvdprodu_f(
                    const vsip_csv_f *svd,
                    vsip_mat_op OpU,
                    vsip_mat_side ApU,
                    const vsip_cmview_f *C);
```

Arguments

`*svd` Pointer to an SVD object, generated by `vsip_dsvd_f`.

`OpU` Specifies the form of op(**U**).

```
typedef enum {
  VSIP_MAT_NTRANS = 0,
  VSIP_MAT_TRANS = 1,
  VSIP_MAT_HERM = 2
} vsip_mat_op;
```

op(**U**) = **U**
op(**U**) = $\mathbf{U}^T$ (real only)
op(**U**) = $\mathbf{U}^H$ (complex only)

`ApU` Indicates if op(**U**) is applied on the left or right of **C**.

```
typedef enum {
  VSIP_MAT_LSIDE = 0,
  VSIP_MAT_RSIDE = 1
} vsip_mat_side;
```

op(**U**) **C**
C op(**U**)

`*C` On entry, view of input matrix **C**, *R* by *S*. On output the data is stored in natural order in the block determined by the offset, row stride, and column stride of the input matrix view. See restrictions below.

Return Value

Returns zero if successful.

Restrictions

Since the output data space may be larger than the input data space it is required that the input data view allow storage in the block for the output data. This means the row stride and column stride must be calculated to accommodate the larger data space, whether it be input or output.

Errors

The input arguments must conform to the following:

1. All objects are valid.
2. `OpU` is valid:
   - Real: `OpU` ∈ `{VSIP_MAT_NTRANS, VSIP_MAT_TRANS }`
   - Complex: `OpU` ∈ `{VSIP_MAT_NTRANS, VSIP_MAT_HERM }`
3. `ApU` is valid: `ApU` ∈ `{VSIP_MAT_LSIDE, VSIP_MAT_RSIDE }`.
4. The SVD object must have been created with the argument "`Usave`" set to `VSIP_SVD_UVFULL` or `VSIP_SVD_UVPART`.
5. The matrix **C**, and the SVD object must be conformant.

Notes/References

It is okay to call `vsip_`*`d`*`svdprodu_`*`f`* after `vsip_`*`d`*`svd_`*`f`* fails. This will result in a non-zero, unsuccessful, return value.

One way to ensure the input/output data space is proper is to calculate the size of the output data space either using the tables under functionality above, or directly given knowledge of the input matrix sizes. If the output data space is larger than the input data space create a matrix view large enough to hold the output data. Create a subview of this with index offset at (0,0) of proper size to hold the input data. The new (sub) view is then the input to the function, and the original view will hold the output data.

Examples

See Also

`vsip_`*`d`*`svd_create_`*`p`*, `vsip_`*`d`*`svdprodv_`*`p`*, `vsip_`*`d`*`svd_destroy_`*`p`*, and `vsip_`*`d`*`svd_getattr_`*`p`*

vsip_*d*svdprodv_*p* Product with V from SV Decomposition

Multiply a matrix by the matrix **V** from a singular value decomposition.

Functionality

This function overwrites a *R* by *S* matrix **C** with

	MAT_LSIDE	MAT_RSIDE
MAT_NTRANS	V C	C V
MAT_TRANS	V^T C	C V^T
MAT_HERM	V^H C	C V^H

Where the matrix **V** was generated by the routine `vsip_dsvd_f`. If a *M* by *N* matrix was the input matrix for the function `vsip_dsvd_f`, then **V** is either a *N* by *N* or *N* by min(*M,N*) matrix, depending which option was used to generate **V**.

If **V** was computed using the `SVD_UVPART` option, then the following table lists the possible dimensions of the matrix **C** before and after this operation:

	Input		Output	
	MAT_LSIDE	**MAT_RSIDE**	**MAT_LSIDE**	**MAT_RSIDE**
MAT_NTRANS	min*(M,N)* by *S*	*R* by *N*	*N* by *S*	*R* by min*(M,N)*
MAT_TRANS	*N* by *S*	*R* by min*(M,N)*	min*(M,N)* by *S*	*R* by *N*
MAT_HERM	*N* by *S*	*R* by min*(M,N)*	min*(M,N)* by *S*	*R* by *N*

For some options the result of this operation is a matrix that is larger than the input matrix. The matrix view object used for the input/output data is const. The first element of the input and the first element of the output are stored at element location (0,0) of the input/output matrix. Other elements are stored in their natural location in the block determined by the row stride and column stride of the input/output view.

If **U** was computed using the `SVD_UVFULL` option, then the following table lists the possible dimensions of the matrix **C** before and after this operation:

	Input & Output	
	MAT_LSIDE	**MAT_RSIDE**
MAT_NTRANS	*N* by *S*	*R* by *N*
MAT_TRANS	*N* by *S*	*R* by *N*
MAT_HERM	*N* by *S*	*R* by *N*

Prototypes

```
int vsip_svdprodv_f(
                    const vsip_sv_f *svd,
                    vsip_mat_op OpV,
                    vsip_mat_side ApV,
                    const vsip_mview_f *C);
int vsip_csvdprodv_f(
                    const vsip_csv_f *svd,
                    vsip_mat_op OpV,
                    vsip_mat_side ApV,
                    const vsip_cmview_f *C);
```

Arguments

`*svd` Pointer to an SVD object, generated by `vsip_dsvd_f`.

`OpV` Specifies the form of op(**V**).

```
typedef enum {
  VSIP_MAT_NTRANS = 0,
  VSIP_MAT_TRANS = 1,
  VSIP_MAT_HERM = 2
} vsip_mat_op;
```

op(**V**) = **V**
op(**V**) = $\mathbf{V}^T$ (real only)
op(**V**) = $\mathbf{V}^H$ (complex only)

`ApV` Indicates if op(**V**) is applied on the left or right of **C**.

```
typedef enum {
  VSIP_MAT_LSIDE = 0,
  VSIP_MAT_RSIDE = 1
} vsip_mat_side;
```

op(**V**) **C**
C op(**V**)

`*C` On entry, view of input matrix **C**, *R* by *S*. On output the data is stored in natural order in the block determined by the offset, row stride, and column stride of the input matrix view. See restrictions below.

Return Value

Returns zero if successful.

Restrictions

Since the output data space may be larger than the input data space it is required that the input data view allow storage in the block for the output data. This means the row stride and column stride must be calculated to accommodate the larger data space, whether it be input or output.

Errors

The input arguments must conform to the following:

1. All objects are valid.
2. `OpV` is valid:
   Real: `OpV` ∈ `{VSIP_MAT_NTRANS, VSIP_MAT_TRANS }`.
   Complex: `OpV` ∈ `{VSIP_MAT_NTRANS, VSIP_MAT_HERM }`.
3. `ApV` is valid: `ApV` ∈ `{VSIP_MAT_LSIDE, VSIP_MAT_RSIDE }`.
4. The SVD object must have been created with the argument "`Vsave`" set to `VSIP_SVD_UVFULL` or `VSIP_SVD_UVPART`.
5. The matrix **C**, and the SVD object must be conformant.

Notes/References

It is okay to call `vsip_`***d***`svdprodv_`***f*** after `vsip_`***d***`svd_`***f*** fails. This will result in a non-zero, unsuccessful, return value.

One way to ensure the input/output data space is proper is to calculate the size of the output data space either using the tables under functionality above, or directly given knowledge of the input matrix sizes. If the output data space is larger than the input data space create a matrix view large enough to hold the output data. Create a subview of this with index offset at (0,0) of proper size to hold the input data. The new (sub) view is then the input to the function, and the original view will hold the output data.

Examples

See Also

`vsip_`***d***`svd_create_`***p***, `vsip_`***d***`svdprodu_`***p***, `vsip_`***d***`svd_destroy_`***p***, and `vsip_`***d***`svd_getattr_`***p***

vsip_*d*svdmatu_*p* Return with U from SV Decomposition

Returns consecutive columns in the matrix **U** from a singular value decomposition.

Functionality

Returns consecutive columns in the matrix **U** from a singular value decomposition of an *M* by *N* matrix, starting with the column *low* and finishing with column *high*. Let u_j denote the jth column in the matrix **U.** This functions returns the following matrix:

$$C = \left[U_{low}, U_{low+1}, U_{low+2}, \cdots, U_{high}\right]$$

Prototypes

```
int vsip_svdmatu_f(
                    const vsip_sv_f *svd,
                    vsip_scalar_vi low,
                    vsip_scalar_vi high,
                    const vsip_mview_f *C);
int vsip_csvdmatu_f(
                    const vsip_csv_f *svd,
                    vsip_scalar_vi low,
                    vsip_scalar_vi high,
                    const vsip_cmview_f *C);
```

Arguments

`*svd`	Pointer to an SVD object, generated by `vsip_dsvd_f`.
`low`	Specifies the first column in U
`high`	Specifies the last column in U
`*C`	On output the data is stored in natural order in the block determined by the offset, row stride, and column stride of the input matrix view. See restrictions below.

Return Value

Returns zero if successful.

Restrictions

Low and high are required to be less than the number of columns in U.

Errors

The input arguments must conform to the following:

1. All objects are valid.
2. `low` must be less than or equal to `high`
3. The SVD object must have been created with the argument "`Usave`" set to `VSIP_SVD_UVFULL` or `VSIP_SVD_UVPART`.
4. If the SVD object was created with the argument "`Usave`" set to `VSIP_SVD_UVPART`, then `high` must be less than or equal to the number of columns for the matrix U.

Notes/References

It is okay to call `vsip_dsvdmatu_f` after `vsip_dsvd_f` fails. This will result in a non-zero, unsuccessful, return value.

Examples

See Also

`vsip_dsvd_create_p`, `vsip_dsvdprodv_p`, `vsip_dsvd_destroy_p`, and `vsip_dsvd_getattr_p`

vsip_*d*svdmatv_*p* Return with V from SV Decomposition

Returns consecutive columns in the matrix **V** from a singular value decomposition.

Functionality

Returns consecutive columns in the matrix **V** from a singular value decomposition of an *M* by *N* matrix, starting with the column *low* and finishing with column *high*. Let v_j denote the jth column in the matrix **V.** This function returns the following matrix:

$$C = \left[V_{low}, V_{low+1}, V_{low+2}, \cdots, V_{high}\right]$$

Prototypes

```
int vsip_svdmatv_f(
                    const vsip_sv_f *svd,
                    vsip_scalar_vi low,
                    vsip_scalar_vi high,
                    const vsip_mview_f *C);
int vsip_csvdmatv_f(
                    const vsip_csv_f *svd,
                    vsip_scalar_vi low,
                    vsip_scalar_vi high,
                    const vsip_cmview_f *C);
```

Arguments

`*svd`	Pointer to an SVD object, generated by `vsip_dsvd_f.`
`low`	Specifies the first column in V
`high`	Specifies the last column in V
`*C`	On output the data is stored in natural order in the block determined by the offset, row stride, and column stride of the input matrix view. See restrictions below.

Return Value

Returns zero if successful.

Restrictions

Low and high are required to be less than the number of columns in V

Errors

The input arguments must conform to the following:

1. All objects are valid.
2. `low` must be less than or equal to `high`
3. The SVD object must have been created with the argument "`Vsave`" set to `VSIP_SVD_UVFULL` or `VSIP_SVD_UVPART`.
4. If the SVD object was created with the argument "`Vsave`" set to `VSIP_SVD_UVPART`, then `high` must be less than or equal to the number of columns for the matrix V.

Notes/References

It is okay to call `vsip_dsvdmatv_f` after `vsip_dsvd_f` fails. This will result in a non-zero, unsuccessful, return value.

Examples

See Also

`vsip_dsvd_create_p`, `vsip_dsvdprodv_p`, `vsip_dsvd_destroy_p`, and `vsip_dsvd_getattr_p`

IMPLEMENTATION DEPENDENT INPUT AND OUTPUT

Introduction

Unlike VSIPL block objects that have a well defined counterpart in user memory, other opaque VSIPL objects have a totally implementation dependent content. For instance the information stored in FFT objects is dependent on the implementation and is not defined by the specification. Only the characteristics of the FFT object are defined in VSIPL.

It is desirable to be able to copy these objects into user memory so they may be stored for later use or sent to another process. The purpose of this chapter is to define sufficient functionality so that all VSIPL objects have a counterpart in user memory. Because these objects are implementation dependent the content of the user memory is not defined. Objects from different implementations are not required to be compatible.

Methodology

The general method for implementation dependent I/O will be to query the implementation dependent object for its size and then use that size to allocate user memory. The user memory is then passed to an *export* function that fills the memory with implementation dependent information suitable to recreate the exported object. The user memory can be stored for later use, or it can be passed as a message to another process where it is used to recreate the functionality of the original object.

To recreate the object the user memory is associated with an object of the proper type using an *import* function. The *import* function uses the exported data and does whatever is necessary to return an object which is a copy of the original object.

The import function takes either a pointer to a created object of the proper object type or a *null* pointer as an input parameter. If a *null* pointer is input the import function returns a new object of the proper type. If an object of the proper type is input then the import function is free to make use of the object. The import function's use of the input object is implementation dependent.

Only objects created initially with import can be used as an input to import. The method used to create the object is entirely implementation dependent. There is no requirement for import to return the same object as that input.

The import function is required to destroy any memory associated with the input object if that memory is not required for use in the returned object. If an object is returned it must be destroyed by the user using the normal destroy functions. The implementation is always responsible for cleaning up after itself. If import returns `NULL` because of an allocation failure or some other reason then any input object must be destroyed by the implementation.

Any memory allocated by the user for storage of the exported object must be destroyed by the user. The memory should not be destroyed until it is no longer associated with an imported object. This will happen after the imported object is destroyed or when the imported object is re-imported using another exported object.

Functionality and Naming

There are many functions associated with all the VSIPL implementation dependent objects, however there are only a few *root* names to remember for I/O functions. The root name of the I/O function goes with the root name (or object name) of the object to create the functions names for I/O. So the function name will look like `vsip_d<object>_<root>_p`.

Three new root names are required which are not used elsewhere in VSIPL. The first is *size* which indicates a query for the amount of user memory required for a VSIPL object to be stored in. The second is *export* which indicates an export of the VSIPL object to the user memory. The final is *import* which is used to create and/or modify the VSIPL objects and import the data.

In addition to these three new functions we also define a *find* function to return a pointer to user memory associated with an imported object.

Root	Functionality
`export`	Export a VSIPL object to user memory.
`import`	Import a VSIPL object stored in user memory. Create the object if necessary. Allocate or free VSIPL memory as necessary for use by the VSIPL object.
`size`	Return the amount of user memory required to export a VSIPL object.
`find`	Return a pointer to user memory.

I/O Functionality

The specification pages for all implementation dependent input/output functions are essentially identical and so are overloaded into a single set of specification pages. To create valid prototypes using the overloaded prototypes replace the *<object>* place holder with a valid object root and the *<object_type>* placeholder with a valid object root type. For instance for the QRD object the name is `qrd` and the type is `qr` so the proper prototype resulting from the overloaded prototype

```
size_t vsip_d<object>_export_p(
                    vsip_d<object_type>_p *obj,
                    size_t N,
                    void *mem_ptr);
```

would be

```
size_t vsip_dqrd_export_p(
                    vsip_dqr_p *obj,
                    size_t N,
                    void *mem_ptr);
```

Of course only I/O functions for which a valid create function are defined in the VSIPL specification are defined for I/O. All specified VSIPL I/O functions are listed below.

Signal Processing Prototypes

FFT objects

Since all FFT objects are of a single type we only require one set of I/O functions.

<object> `fft`

<object_type> `fft`

`size_t vsip_`**`fft_export_`**`f(`
`vsip_fft_f *obj,`
`size_t N,`
`void *mem_ptr);`
	FFT export

`vsip_fft_f *vsip_`**`fft_import_`**`f(`
`vsip_fft_f *obj,`
`void *mem_ptr);`
	FFT import

`size_t vsip_`**`fft_size_`**`f(`
`vsip_fft_f`
`*obj);`
	FFT size

`void *vsip_`**`fft_find_`**`f(`
`vsip_fft_f`
`*obj);`
	Find a pointer to the user data associated with an FFT object

Multiple FFT objects

Since all FFTM objects are of a single type we only require one set of I/O functions.

<object> `fftm`

<object_type> `fftm`

`size_t vsip_`**`fftm_export_`***`f`*`(`
`vsip_fftm_`*`f`*` *obj,`
`size_t N,`
`void *mem_ptr);`
	Multiple FFT export

`vsip_fftm_`*`f`*` *vsip_`**`fftm_import_`***`f`*`(`
`vsip_fftm_`*`f`*` *obj,`
`void *mem_ptr);`
	Multiple FFT import

`size_t vsip_`**`fftm_size_`***`f`*`(` `vsip_fftm_`*`f`*` *obj);`	Multiple FFT size
`void *vsip_`**`fftm_find_`***`f`*`(` `vsip_fftm_`*`f`*` *obj);`	Find a pointer to the user data associated with a multiple FFT object

Two Dimensional FFT Objects

Since all two dimensional FFT objects are of a single type we only require one set of I/O functions.

<object> `fft2d`

<object_type> `fft2d`

`size_t vsip_`**`fft2d_export_`***`f`*`(`
`vsip_fft2d_`*`f`*` *obj,`
`size_t N,`
`void *mem_ptr);`
	Two dimensional FFT export

`vsip_fft2d_`*`f`*` *vsip_`**`fft2d_import_`***`f`*`(`
`vsip_fft2d_`*`f`*` *obj,`
`void *mem_ptr);`
	Two dimensional FFT import

`size_t vsip_`**`fft2d_size_`***`f`*`(` `vsip_fft2d_`*`f`*` *obj);`	Two dimensional FFT size
`void *vsip_`**`fft2d_find_`***`f`*`(` `vsip_fft2d_`*`f`*` *obj);`	Find a pointer to the user data associated with a two dimensional FFT object

Three Dimensional FFT Objects

Since all three dimensional FFT objects are of a single type we only require one set of I/O functions.

<object> `fft3d`

<object_type> `fft3d`

`size_t vsip_`**`fft3d_export_`***`f`*`(`
`vsip_fft3d_`***`f`*` *obj,`
`size_t N,`
`void *mem_ptr);`
	Three dimensional FFT export

`vsip_fft3d_`***`f`*` *vsip_`**`fft3d_import_`***`f`*`(`
`vsip_fft3d_`***`f`*` *obj,`
`void *mem_ptr);`
	Three dimensional FFT import

`size_t vsip_`**`fft3d_size_`***`f`*`(` `vsip_fft3d_`***`f`*` *obj);`	Three dimensional FFT size
`void *vsip_`**`fft3d_find_`***`f`*`(` `vsip_fft3d_`***`f`*` *obj);`	Find a pointer to the user data associated with an three dimensional FFT object

One Dimensional Convolution

Only real float convolution objects are defined in the current VSIPL specification.

<object> `conv1d`

<object_type> `conv1d`

`size_t vsip_`**`conv1d_export_`***`f`*`(`
`vsip_conv1d_`***`f`*` *obj,`
`size_t N,`
`void *mem_ptr);`
	One dimensional convolution export

`vsip_conv1d_`***`f`*` *vsip_`**`conv1d_import_`***`f`*`(`
`vsip_conv1d_`***`f`*` *obj,`
`void *mem_ptr);`
	One dimensional convolution import

`size_t vsip_`**`conv1d_size_`***`f`*`(`
`vsip_conv1d_`***`f`*
`*obj);`
	One dimensional convolution size

`void *vsip_`**`conv1d_find_`***`f`*`(`
`vsip_conv1d_`***`f`*
`*obj);`
	Find a pointer to the user data associated with an one dimensional convolution object

Two Dimensional Convolution

Only real float convolution objects are defined in the current VSIPL specification.

<object> `conv2d`

<object_type> `conv2d`

`size_t vsip_conv2d_export_f(`
`vsip_conv2d_f *obj,`
`size_t N,`
`void *mem_ptr);`
	Two dimensional convolution export

`vsip_conv2d_f *vsip_conv2d_import_f(`
`vsip_conv2d_f *obj,`
`void *mem_ptr);`
	Two dimensional convolution import

`size_t vsip_conv2d_size_f(`
`vsip_conv2d_f`
`*obj);`
	Two dimensional convolution size

`void *vsip_conv2d_find_f(`
`vsip_conv2d_f`
`*obj);`
	Find a pointer to the user data associated with an two dimensional convolution object

One Dimensional Correlation

Real and complex float correlation objects are defined in the specification

<object> `corr1d`

<object_type> `corr1d`

`size_t vsip_corr1d_export_f(`
`vsip_corr1d_f *obj,`
`size_t N,`
`void *mem_ptr);`
	One dimensional correlation export

`vsip_corr1d_f *vsip_corr1d_import_f(`
`vsip_corr1d_f *obj,`
`void *mem_ptr);`
	One dimensional correlation import

`size_t vsip_corr1d_size_f(`
`vsip_corr1d_f`
`*obj);`
	One dimensional correlation size

`void *vsip_corr1d_find_f(`
`vsip_corr1d_f`
`*obj);`
	Find a pointer to the user data associated with an one dimensional correlation object

`vsip_ccorr1d_export_f(`
`vsip_ccorr1d_f`
`*obj,`
`size_t N,`
`void *mem_ptr);`
	One dimensional complex correlation export

`vsip_ccorr1d_f *vsip_ccorr1d_import_f(`
`vsip_corr1d_f *obj,`
`void *mem_ptr);`
	One dimensional complex correlation import

`size_t vsip_ccorr1d_size_f(`
`vsip_corr1d_f`
`*obj);`
	One dimensional complex correlation size

`void *vsip_ccorr1d_find_f(`
`vsip_corr1d_f`
`*obj);`
	Find a pointer to the user data associated with an one dimensional complex correlation object

Two Dimensional Correlation

Real and complex float correlation objects are defined in the specification.

<object> `corr2d`

<object_type> `corr2d`

`size_t vsip_corr2d_export_f(`
`vsip_corr2d_f *obj,`
`size_t N,`
`void *mem_ptr);`
	Two dimensional correlation export

`vsip_corr2d_f *vsip_corr2d_import_f(`
`vsip_corr2d_f *obj,`
`void *mem_ptr);`
	Two dimensional correlation import

`vsip_corr2d_size_f(` `vsip_corr2d_f *obj);`	Two dimensional correlation size
`void *vsip_corr2d_find_f(` `vsip_corr2d_f *obj);`	Find a pointer to the user data associated with an two dimensional correlation object

`size_t vsip_ccorr2d_export_f(`
`vsip_ccorr2d_f *obj,`
`size_t N,`
`void *mem_ptr);`
	Two dimensional complex correlation export

`vsip_ccorr2d_f *vsip_ccorr2d_import_f(`
`vsip_ccorr2d_f *obj,`
`void *mem_ptr);`
	Two dimensional complex correlation import

`size_t vsip_ccorr2d_size_f(`
`vsip_ccorr2d_f`
`*obj);`
	Two dimensional complex correlation size

`void *vsip_ccorr2d_find_f(`
`vsip_ccorr2d_f`
`*obj);`
	Find a pointer to the user data associated with an two dimensional complex correlation object

FIR Filter Object

Real and complex float FIR filter objects are defined in the specification.

<object> `fir`

<object_type> `fir`

`size_t vsip_fir_export_f(`
`vsip_fir_f *obj,`
`size_t N,`
`void *mem_ptr);`
	FIR export

`vsip_fir_f *vsip_fir_import_f(`
`vsip_fir_f *obj,`
`void *mem_ptr);`
	FIR import

`size_t vsip_fir_size_f(` `vsip_fir_f *obj);`	FIR size
`void *vsip_fir_find_f(` `vsip_fir_f *obj);`	Find a pointer to the user data associated with a FIR filter object

`size_t vsip_cfir_export_f(`
`vsip_cfir_f *obj,`
`size_t N,`
`void *mem_ptr);`
	Complex FIR export

`vsip_cfir_f *vsip_cfir_import_f(`
`vsip_cfir_f *obj,`
`void *mem_ptr);`
	Complex FIR import

`size_t vsip_cfir_size_f(` `vsip_cfir_f *obj);`	Complex FIR size
`void *vsip_cfir_find_f(` `vsip_cfir_f *obj);`	Find a pointer to the user data associated with a complex FIR filter object

IIR Filter Object

Only real float IIR objects are in the specification.

<object> `iir`

<object_type> `iir`

`size_t vsip_iir_export_f(`
`vsip_iir_f *obj,`
`size_t N,`
`void *mem_ptr);`
	IIR export

`vsip_iir_f *vsip_iir_import_f(`
`vsip_iir_f *obj,`
`void *mem_ptr);`
	IIR import

`size_t vsip_iir_size_f(` `vsip_iir_f *obj);`	IIR size
`void *vsip_iir_find_f(` `vsip_iir_f *obj);`	Find a pointer to the user data associated with an IIR filter object

Linear Algebra Prototypes

Linear algebra objects are associated with a decomposition of a matrix which is a separate operation from the create function. It is erroneous to export a decomposition object without an associated decomposed matrix.

LUD Object I/O

Float and complex LUD objects are defined in the specification

<object> `lud`

<object_type> `lu`

`size_t vsip_`**`lud_export_`***`f`*`(`
`vsip_lu_`***`f`*` *obj,`
`size_t N,`
`void *mem_ptr);`
	LUD export

`vsip_lu_`***`f`*` *vsip_`**`lud_import_`***`f`*`(`
`vsip_lu_`***`f`*` *obj,`
`void *mem_ptr);`
	LUD import

`size_t vsip_`**`lud_size_`***`f`*`(` `vsip_lu_`***`f`*` *obj);`	LUD size
`void *vsip_`**`lud_find_`***`f`*`(` `vsip_lu_`***`f`*` *obj);`	Find a pointer to the user data associated with a LUD object

`size_t vsip_`**`clud_export_`***`f`*`(`
`vsip_clu_`***`f`*` *obj,`
`size_t N,`
`void *mem_ptr);`
	Complex LUD export

`vsip_clu_`***`f`*` *vsip_`**`clud_import_`***`f`*`(`
`vsip_clu_`***`f`*` *obj,`
`void *mem_ptr);`
	Complex LUD import

`size_t vsip_`**`clud_size_`***`f`*`(` `vsip_clu_`***`f`*` *obj);`	Complex LUD size
`void *vsip_`**`clud_find_`***`f`*`(` `vsip_clu_`***`f`*` *obj);`	Find a pointer to the user data associated with a Complex LUD object

Cholesky Object I/O

Float and complex cholesky objects are defined in the specification

<object> `chold`

<object_type> `chol`

`size_t vsip_chold_export_f(`
`vsip_chol_f *obj,`
`size_t N,`
`void *mem_ptr);`
	Cholesky export

`vsip_chol_f *vsip_chold_import_f(`
`vsip_chol_f *obj,`
`void *mem_ptr);`
	Cholesky import

`size_t vsip_chold_size_f(` `vsip_chol_f *obj);`	Cholesky size
`void *vsip_chold_find_f(` `vsip_chol_f *obj);`	Find a pointer to the user data associated with a Cholesky object

`size_t vsip_cchold_export_f(`
`vsip_cchol_f *obj,`
`size_t N,`
`void *mem_ptr);`
	Complex Cholesky export

`vsip_cchol_f *vsip_cchold_import_f(`
`vsip_cchol_f *obj,`
`void *mem_ptr);`
	Complex Cholesky import

`size_t vsip_cchold_size_f(` `vsip_cchol_f *obj);`	Complex Cholesky size
`void *vsip_cchold_find_f(` `vsip_cchol_f *obj);`	Find a pointer to the user data associated with a complex Cholesky object

QRD Object I/O

Float and complex QRD objects are defined in the specification

<object> `qrd`

<object_type> `qr`

`size_t vsip_qrd_export_f(`
`vsip_qr_f *obj,`
`size_t N,`
`void *mem_ptr);`
	QRD export

`vsip_qr_f *vsip_qrd_import_f(`
`vsip_qr_f *obj,`
`void *mem_ptr);`
	QRD import

`size_t vsip_qrd_size_f(` `vsip_qr_f *obj);`	QRD size
`void *vsip_qrd_find_f(` `vsip_qr_f *obj);`	Find a pointer to the user data associated with a QRD object
`size_t vsip_cqrd_export_f(` `vsip_cqr_f *obj,`	Complex QRD export

size_t N, void *mem_ptr);	
vsip_cqr_*f* *vsip_**cqrd_import_***f*(vsip_cqr_*f* *obj, void *mem_ptr);	Complex QRD import
size_t vsip_**cqrd_size_***f*(vsip_cqr_*f* *obj);	Complex QRD size
void *vsip_**cqrd_find_***f*(vsip_cqr_*f* *obj);	Find a pointer to the user data associated with a complex QRD object

SVD Object I/O

Float and complex SVD objects are defined in the specification

<object> svd

<object_type> sv

size_t vsip_**svd_export_***f*(vsip_sv_*f* *obj, size_t N, void *mem_ptr);	SVD export
vsip_sv_*f* *vsip_**svd_import_***f*(vsip_sv_*f* *obj, void *mem_ptr);	SVD import
size_t vsip_**svd_size_***f*(vsip_sv_*f* *obj);	SVD size
void *vsip_**svd_find_***f*(vsip_sv_*f* *obj);	Find a pointer to the user data associated with a SVD object
size_t vsip_**csvd_export_***f*(vsip_csv_*f* *obj, size_t N, void *mem_ptr);	Complex SVD export
vsip_csv_*f* *vsip_**csvd_import_***f*(vsip_csv_*f* *obj, void *mem_ptr);	Complex SVD import
size_t vsip_**csvd_size_***f*(vsip_csv_*f* *obj);	Complex SVD size
void *vsip_**csvd_find_***f*(vsip_csv_*f* *obj);	Find a pointer to the user data associated with a complex SVD object

vsip_*d<object>*_export_*p* Export

Export an object and any associated data to user memory.

Functionality

This function allows an object to export information to user memory that can be used to create a new object of identical functionality as the original object. If the exported object is normally associated with array decomposition then the object must include the decomposition before the export is done.

Prototypes

```
size_t vsip_d<object>_export_p(
                    vsip_d<object_type>_p *obj,
                    size_t N,
                    void *mem_ptr);
```

Arguments

`obj`	Valid object.
`N`	Size of memory in bytes referenced by `mem_ptr`.
`mem_ptr`	User data array of sufficient size to hold the exported object and its data. This pointer must be byte aligned on a long word.

Return Value

Size of memory in bytes used to store the exported information.

Restrictions

If the object is associated with a matrix decomposition, the decomposition must be performed before the export function is called.

Errors

The argument must conform to the following:

1. The object must be valid.

Notes/References

If the memory required to export the object is not sufficient the export will fail and the returned value will be zero.

The result of using a memory pointer which is not long word aligned is implementation dependent.

Any valid object of the proper data type may be exported.

Examples

See example at the end of this chapter.

See Also

vsip_*d<object>*_find_*p* Find

Return the memory pointer of a user memory associated with an object.

Functionality

Return the pointer to the beginning of user memory associated with an object. If the object has no user memory associated with it then the function returns a *null* pointer.

Prototypes

```
void *vsip_d<object>_find_p(
                    vsip_d<object_type>_p *obj);
```

Arguments

`obj` Valid object.

Return Value

Pointer to user memory associated with an object, or `NULL` if the object is not associated with any user memory.

Restrictions

Errors

The argument must conform to the following:

1. The object must be valid.

Notes/References

Even though a pointer to the user memory is returned, any user memory associated with an object is owned by VSIPL until the object is either destroyed, or reused by the `vsip_d<object>_import_p` function. The results of modifying memory associated with an object is implementation dependent and is erroneous in a compliant VSIPL program.

Examples

See example at the end of this chapter.

See Also

vsip_*d<object>*_import_*p* Import

Import an object.

Functionality

This function will re-create an object exported using `vsip_d<object>_export_p`.

The import function is called with a *null* pointer if no object is available. Objects used in the import function must be created by the import function.

The import function may be used in a loop to import objects. It is the responsibility of the import function to allocate or free memory in the input object as necessary. The object returned by the import function must be destroyed after it is no longer needed using the `vsip_d<object>_destroy_p` function.

User memory associated with the import is not destroyed by VSIPL and must be destroyed by the user after the object is destroyed or associated with other user memory (used in another import). The object owns the associated user memory until either the object is destroyed, or other user memory is imported to the object. User memory associated with the object must not be destroyed until the object is destroyed.

If the import fails returning a *null* pointer, the input argument is also destroyed.

Prototypes

```
vsip_d<object_type>_p *vsip_d<object>_import_p(
                    vsip_d<object_type>_p *obj,
                    void *mem_ptr);
```

Arguments

`obj`	A valid object or `NULL`.
`mem_ptr`	Pointer to user memory initialized with `vsip_d<object>_export_p`.

Return Value

A valid object or a *null* pointer if the import fails.

Restrictions

Errors

The argument must conform to the following:

1. The imported user memory must be a valid pointer to an exported object if the proper type.
2. The input object must either be a *null* pointer or an object previously created with import.

Notes/References

Since import allows the reuse of VSIPL objects it is important that the implementation query the input VSIPL object to determine its state and destroy any VSIPL memory which may not be needed by the implementation. The methods for importing an object are implementation dependent. The intent of this note is to prevent memory leaks.

Only objects created with the import function may be used as input to import.

Examples

See example at the end of this chapter.

See Also

vsip_*d<object>*_size_*p* Size

Return the minimum memory size of *user data* required to hold an object and any data associated with the object.

Functionality

Calculate the amount of memory required to store an object exported with the `vsip_d<object>_export_p` function. The size includes all the data or other information associated with the object.

Prototypes

```
size_t vsip_d<object>_size_p(
                    vsip_d<object_type>_p *obj);
```

Arguments

`obj` Valid object.

Return Value

Memory size in bytes required to hold user data exported by `vsip_d<object>_export_p`.

Restrictions

Errors

The argument must conform to the following:

1. The object must be valid.

Notes/References

Examples

See example at the end of this chapter.

See Also

`vsip_d<object>_export_p`

Examples

This section shows VSIPL example code for I/O functions. The Example usea a QR decompostion to solve the problem

$\mathbf{A}^T\mathbf{A}\mathbf{x} = \mathbf{b}$

for **x** given a float matrix **A** of size (*aRows*, *aCols*) and a right-hand side vector **b**.

```
/* Example 1:
    VSIPL QR Export code */

/* Declarations */
#define AROWS   20
#define ACOLS   10
#define NDECOMP 15
#define QR_BIG_BUFFER_SIZE 1000000

vsip_mview_f *A;
int iD;
void      *qrBuffer;
vsip_qr_f *qrdObject;
size_t    qrBytes,
          qrCheck;

/* Construction: allocate memory for the matrix and for its decomposition
   The 'Q' matrix from the factorization will not be saved */

A = vsip_mcreate_f(A_ROWS, A_COLS, VSIP_ROW, VSIP_MEM_NONE);
qrdObject = vsip_qrd_create_f(A_ROWS, A_COLS, VSIP_QRD_NOSAVEQ);

/* Allocate memory for an export buffer of the maximum anticipated size */
qrBuffer = (void *) malloc(QR_BIG_BUFFER_SIZE);

/* Loop over the number of decompositions to be done */
for (iD = 0; iD < NDECOMP; iD++){
   /* Acquire the matrix A somehow */

   /* Compute the QR decomposition of the matrix A */
   vsip_qrd_f(qrdObject, A);

   /* Export the decomposition: find out how large it is,
     call the export function, and verify that everything
     worked properly */
   qrBytes = vsip_qrd_size_f(qrdObject);
   if (qrBytes > QR_BIG_BUFFER_SIZE)
   {
      printf("QR decomposition too large for buffer.\n");
      exit(1);
   }

   qrCheck = vsip_qrd_export_f(qrdObject, QR_BIG_BUFFER_SIZE, qrBuffer);
   if (qrCheck == 0)
   {
      printf("Failed to properly export QR decomposition.\n");
      exit(1);
   }
```

```
   /* At this point, send the QR buffer to the other process (for example
      with MPI_SEND or with a file write operation). It is possible to
      send the size of the buffer in a separate message so that the
      other process can allocate a buffer of exactly the right size
      In this example, the other process is required to allocate
      buffers that are ``sufficiently large'' */
}

/* Clean-up */
vsip_malldestroy_f(A);
vsip_qrd_destroy_f(qrdObject);
free(qrBuffer);

/*****************************************************************/
/* Example 2: */
/* VSIPL QR Import code */
/* Declarations */

#define A_ROWS  20
#define A_COLS  10
#define NDECOMP 15
#define QR_BIG_BUFFER_SIZE 1000000

vsip_vview_f *bx;
int       iD;
void      *recvBuffer;
void      *qrBufferA,
          *qrBufferB;
size_t     qrBytes;
vsip_qr_f *qrdObject = NULL;

/* Construction: allocate memory for the vectors and the buffers */
bx = vsip_vcreate_f(A_COLS, VSIP_MEM_NONE);
qrBufferA = (void *) malloc(QR_BIG_BUFFER_SIZE);
qrBufferB = (void *) malloc(QR_BIG_BUFFER_SIZE);

/* Loop over the number of decompositions to be done */
for (iD = 0; iD < NDECOMP; iD++){
   /* Choose the buffer to be used by this loop
   /* Perform double-buffering: on even iterations use buffer A,
      on odd iterations use buffer B */
   if ((iD % 2) == 0){
        recvBuffer = qrBufferA;
   } else {
       recvBuffer = qrBufferB;
   }

   /* Presume that the exported QR decomposition information is
      acquired (for example, via an MPI_RECV or a file read operation)
      into recvBuffer at this point. Notice that a single buffer would
      not work here because the buffer currently underlying qrdObject
      is owned by VSIPL until the import operation is performed
      Destroying qrdObject would also allow the buffer to be reused,
      but destroy may be expensive in an inner loop */

    /* Import the QR decomposition of the matrix A
```

```
        Notice that no create or destroy operation on the qrdObject is
        needed: indeed, using the 'create' operation to create qrdObject
        first would have resulted in a memory leak
        The 'create' operation happens in the first loop iteration because
        qrdObject was initialized to be a NULL pointer.
        On subsequent calls to vsip_qrd_import, the memory associated with
        the qrdObject is re-used or re-allocated by the implementation */
    qrdObject = vsip_qrd_import(qrdObject, recvBuffer);

    /* Presume the right-hand side b is acquired here */
    /* Use the decomposition to solve for x
        First line solves R^T c = b for the vector c;
        Second line solves R x = c for the vector x */
    vsip_qrsolr_f(qrdObject, VSIP_MAT_TRANS, 1.0, bx);
    vsip_qrsolr_f(qrdObject, VSIP_MAT_NONE, 1.0, bx);

    }
/* Clean-up */
/* Destroy the last QR object imported */
vsip_qrd_destroy_f(qrdObject);
vsip_valldestroy_f(bx);
free(qrBufferA);
free(qrBufferB);
```

NOTES TO IMPLEMENTORS

VSIPL Implementation Notes

Incomplete Type Definitions

VSIPL objects are implemented via *incomplete type definitions*. All members of the resulting objects are opaque. Incomplete type definitions are accomplished in the following manner.

Example: The library implementation uses a header file that contains the following definition of the type `vsip_bar`:

```
struct foo {
   int x;
   int y;
   float *x;
   …
};
typedef struct foo vsip_bar;
```

The application programmer is provided a related header file (but not given access to the implementation header files) that contains the following definition of the type `vsip_bar`:

```
struct foo;
typedef struct foo vsip_bar;
```

This allows the application programmer to declare pointers to objects of type `vsip_bar`. Functions are provided by the implementation to create the vsip_bar structure and return the pointer to the structure. for instance

```
vsip_bar *bar =vsip_bar_create(arg);
```

This returned pointer is then used as an argument to functions. Since the user has no knowledge of the private structures the implementor must supply all functions which operate on the structure. The purpose here is to prevent the user from producing non-portable applications, and to allow the implementor maximum ability to optimize their code in a portable library.

The compiler provides strict type checking. It does not allow the application programmer to access any of the elements of the structure.

Checking for Object Validity

All VSIPL objects in development mode should be implemented with a "*magic number*" data tag. When an object is destroyed the magic number should be set to a value that indicates an *invalid* object. This is so that development mode can detect attempts to operate on destroyed objects. This is what is implied by an *errors* requirement such as, "All the objects must be valid."

It is further suggested that the magic number be unique for each object type. This is a convenience for debugging. The application programmer relies on the compiler to enforce type correctness of arguments

Change Notes

Introduction

This section was added after version 1.01 of the specification. Its purpose is to provide guidance to implementors as to what changes were made between one version of the specification and the next. The method for indicating the changes will be a table of errata if the change is a result of an error in the earlier specification document, and as a table of changes if the change is the result of additions to the specification.

The forum decided not to reconstruct errata for earlier versions so there is no table for changes between 1.0 and 1.01.

Errata

Version 1.01	Version 1.02	Error correction
Entire Document	Entire Document	Many symbols in equations were incorrectly rendered when converting from the word processing format to portable document format. The conversion was moved to another computer and the problem seems to have been corrected.
Title Page	Title Page	Corrected Copyright for proper years
P 4	P 4	Changed the sentence “The process of moving data from the logical VSIPL data space to the logical user data space is called admission;” to “The process of moving data from the logical user data space to the logical VSIPL data space is called admission;”
P 13	P 13	Removed extraneous “,” from unsigned long int description.
P 17	P 17	Changed “No, preferred storage layout” to “No preferred storage layout”.
P 26, P 607, P 614, P 617, P 621	P 26, P 583, P 588,P 591, P 595	The description of the `STATE_NO_SAVE` and `STATE_SAVE` enumerated types was reveresed in the type definitions.
P 52	P 50	The “Prototype” heading is missing on the block destroy man page.
P 109	P 101	The matrix bind function prototypes had incorrect type definitions for the stride and bind arguments.
P 113	P 105	The matrix create functionality section incorrectly uses the matrix bind function. The usage is corrected.
P 151, P 152	P 140, P 141	The tensor bind function prototype section had incorrect type definitions for the stride and bind arguments.
P 154	P 143	The tensor create functionality section incorrectly uses the tensor bind function. The usage is corrected.

Version 1.01	Version 1.02	Error correction
P 515	P 495	The example for vsip_ccfft incorrectly uses 2 * N in cblockbind. VSIPL blockbind functions always use the number of elements, whatever the type, not the length of the user data array.
P 539	P 519	The crfft2dop functionality paragraph incorrectly describes the transform as real to complex.
P 629	P 603	In the list of matrix and vector operations the description of `vsip_`***`d`***`vmprod_`***`p`*** is as “Matrix Vector Product”. It should be “Vector Matrix Product”.
P 629, P 642, P 643	P 603, P 616, P 617	The functions names for matrix conjugate product and matrix hermitian product are incorrectly overloaded with the *depth* operator ”***d***”. Since these are only complex this should be a “c”
P 702	P 673	In man page for `vsip_`***`d`***`svdprodv_`***`p`***, the sentence “If U was computed using the `SVD_UVFULL` option” changed to “If V was computed using the `SVD_UVFULL` option”.

Version 1.02	Version 1.1	Error correction
P 73	P 74	In the functionality section for `vsip_`***`d`***`vcreate_`***`p`*** an argument `VSIP_ROW` is (incorrectly) included. The argument is removed.
P 101	P 103	Complex matrix bind prototype incorrectly shows a bind to a real block. This was changed to bind to a complex block.
P 105	P 107	Corrections to the matrix bind functionality (descriptions) in the matrix create function.
P 198	P 202	Corrected hypot function to return `vsip_scalar_`***`f`***.
P 357	P 363	The last sentence of Error #4 of `vsip_`***`d`***`v`***`d`***`mmul_`***`p`*** was changed from “The input vector view and output vector view must not overlap” to “The input vector view and output matrix view must not overlap”.
P 430	436	For Vector/Matrix NOT change the description from “Computes the NOT (one's complement), by element, of two vectors/matrices” to “Computes the NOT (one's complement), by element, of a vector/matrix.”
P 442	P 448	For `vsip_`***`ds`***`fill_`***`p`*** there is only one view object so errors one and three are not applicable, and were removed.
P 438 –	P 444 –	The copy man page showed a matrix and tensor view for

Version 1.02	Version 1.1	Error correction
P 441	P 446	index types. There are only vector views of index types supported in VSIPL. The page was modified to reflect this.
P 441	P446	The example of copies for the Core profile was wrong. The example was updated to meet the requirements in the Core profile document.
P 460	P 470	In the scatter man page under **Notes/References** changed "The view of the destination vector/matrix is not modified." to "The attributes of the destination vector/matrix are not modified."
P 519	P 527	In vsip_crfft2dop_f functionality changed "Computes a complex-to-real (*inverse*) 2D Fast Fourier Transform (FFT) of the real M by N matrix $X = (x_{m,n})$, and stores the results in the complex matrix $Y = (y_{u,v})$." to " Computes a complex-to-real (*inverse*) 2D Fast Fourier Transform (FFT) of the complex matrix $X = (x_{m,n})$, and stores the results in the real matrix $Y = (y_{u,v})$."
P 575	P 585	Added definition for τ_p to the functionality section of the Chebyshev window. Added note about reference for the window in the introduction to the chapter in the *See Also*.section.
P 588	P 599	changed $\lceil out_len/D \rceil$ to $\lceil in_len/D \rceil$
P 599	P 611	Removed "and of positive length" from the first error in the histogram function. View objects must all be of positive length and no view object is created in this function so checking for this error here is incorrect.
P 600	P 612	The equation in the functionality section for `vsip_`***ds***`freqswap_`***f*** was incorrect and has been corrected.
P 615, P 617	P 627, P 629	The size of the second argument in the arguments section for `vsip_cmprodj_`***p*** and `vsip_`***d***`mprod_`***p*** was listed as N by P. This is corrected to P by N.
Examples	Examples	All examples were reviewed for errors. Formatting was standardized. Many historical problems (examples written before a specification change) were corrected. It should be noted that this does not guarantee all the examples are now correct.
Various	Various	In several *examples* where the `vsip_mem_hint` type is used, and various argument lists where the `vsip_mem_hint` is explained, were found the memory

Version 1.02	Version 1.1	Error correction
		hints `VSIP_MEM_CONST_RDONLY_RDONLY` or `VSIP_MEM_CONST_RDONLY`. There are no such memory hints and these were all changed to `VSIP_MEM_CONST`.
Various	Various	When referring to a *length* replaced "positive" with "greater than zero" to make it clearer that zero length views are not allowed.
Various	Various	Changed "users specifies" to "user specifies" several places.

Version 1.1	Version 1.2	Error correction
`vsip_asin_`***p*** `vsip_`***s***`asin_`***p***	`vsip_asin_`***p*** `vsip_`***s***`asin_`***p***	The output is incorrectly stated to be between [0, π]. Changed to [-π/2, π/2].

In example for `vsip_ccfftm`***x_f***		Changed
		`sum=vsip_cvcreate_d(ROWS,VSIP_MEM_NONE);`
		to
		`sum=vsip_cvcreate_d(N,VSIP_MEM_NONE;)`
		Changed typedef `vsip_rand_state` to `vsip_randstate`

In `vsip_`***d***`covsol_`***p***		Added to the *errors* that **A** and **XB** must not overlap
In `vsip_`***d***`qrsol_`***p***		Added to the errors that **XB** must not overlap with the input matrix.

Version 1.2	Version 1.3	Error correction
In `vsip_copyto_user_`***p*** `and` `vsip_copyfrom_user_`***p***		Changed type `VSIP_MAJOR` to `vsip_major`
In Changes section		Added aditional information on change to complex storage funtion. This change was new in verstion 1.2.
Other changes		The mechanics of producing a new document cause numerous editing tweaks which are impossible to keep track of. If a proofreader finds something they feel should be in this list feel free to email the group for consideration.

Changes

Version	Change
1.02	Added section for change notes to the document.
1.1	Added complex to sum value (`vsip_dssumval_p`).
1.1	Added complex to elementwise logical equal (`vsip_dsleq_p`).
1.1	Added complex to elementwise logical not equal (`vsip_dslne_p`).
1.1	Added `vsip_fft_setwindow_f` and `vsip_fftm_setwindow_f` routines to FFT section. Modified functionality table in FFT section and signal processing introduction to reflect additions.
1.1	Added SVD routines `vsip_dsvdmatu_f` and `vsip_dsvdmatv_f.` Modified functionality table in SVD section to reflect additions.
1.1	Added chapter for implementation dependent input and output.
1.2	Added functions `vsip_dscopyto_user_p` and `vsip_dscopyfrom_user_p` to support copying data from user memory to the data space referenced by a view, and copying data referenced by a view directly to user memory.
1.2	Added functionality to support vector and matrix elementwise floor (`vsip_sfloor_p_p`), ceiling (`vsip_sceil_p_p`) and round (`vsip_sround_p_p`) operations. Although these functions may be done in place or copied to a float output, sometimes the output is desired to be integer. The naming convention is used to support a type change in the output view to integer.
1.2	Added type vector index (`_vi`) as a standard (unsigned) integer type. This means that any defined function supporting integer calculations may be implemented to support vector index view types. This makes vector index manipulations easier and eliminates the need for copies to and from other integer types.
1.2	Added section in Change Notes for Deprecated Functions
1.2	Added `vsip_cstorage_p.` Deprecated `vsip_cstorage`. Allows the use of different complex storage for different types. Also note changes to public header file defined on the new complex storage page.
1.3	Added short section allowing new functionality to be added to VSIPL in companion documents not part of this main document. (See additonal functions below).
1.3	Added gather and scatter functionality for matrix index vectors.
1.3	Added scalar vector/matrix compares to Logical Operations

Additional Functions

As time goes on additional functionality is requested to be added to VSIPL; however, as the specification has grown it has become unwieldly to change. Starting with version 1.3 of the specification additional functionality that is not already part of VSIPL will be added in standalone documents that reference VSIPL as the parent specification.

It also should be noted that any vendor might add VSIPL compliant functionality which is not part of the specification. This, of course, would produce non-portable code if an application programer used those functions; however there is always a need for speed and new functionality and the forum is very slow. There is also no way to keep vendors from doing this.

Vendors who supply functionality not defined by the VSIPL forum should use a naming convention which does not conflict with VSIPL by using a different prefix. For instance `acme_myfunction_f`. If the vendor desires there functionality to become part of the VSIPL specification they should write a suitable API document and submit it to the forum for discussion. Note that VSIPL is an open API and any additions need to be non-propriatary and open.

Deprecated Functions

A goal of any standards body is to write it right the first time and never do anything to it that would invalidate a previous versions specifications. The reality is somewhat less. This section includes functions that have been deprecated and may require some code changes if the functions were used in legacy code.

`vsip_cstorage`	Complex Storage Deprecated

vsip_cstorage Complex Storage

This function is deprecated and new code should use `vsip_cstorage_p`. Returns the preferred complex storage format, *interleaved, split*, or *none* for the system.

Functionality

Prototypes

```
vsip_cmplx_mem vsip_cstorage (
                     void);
```

Where:

```
typedef enum {
  VSIP_CMPLX_INTERLEAVED = 0,        Interleaved
  VSIP_CMPLX_SPLIT = 1,              Split, separate real and imaginary
  VSIP_CMPLX_NONE = 2                No, preferred storage layout
} vsip_cmplx_mem;
```

Arguments

None.

Return Value

Returns the preferred complex storage format.

Restrictions

Errors

Notes/References

It is also possible to determine the preferred storage format at compile time. The *include* file `vsip.h` defines the value of `VSIP_CMPLX_MEM` to be one of: {`VSIP_CMPLX_INTERLEAVED` | `VSIP_CMPLX_SPLIT` | `VSIP_CMPLX_NONE`}.

e.g.

```
#define VSIP_CMPLX_MEM (VSIP_CMPLX_INTERLEAVED)
```

Examples

See Also

`vsip_cstorage_p`

THE END

VSIPL Addendum

Addendum to Version 1.3 of VSIPL specification

Approved by VSIPL Forum

January 31, 2008

The three documents following this page define VSIPL APIs supporting data sorting , data interpolation, and matrix permutation functionality.

VSIPL Interpolation API

Version 1.0

Addendum to Version 1.3 or later of VSIPL specification

Approved by VSIPL Forum

January 31, 2008

© 2008 Georgia Tech Research Corporation, all rights reserved.

A non-exclusive, non-royalty bearing license is hereby granted to all persons to copy, modify, distribute and produce derivative works for any purpose, provided that this copyright notice and following disclaimer appear on all copies:

THIS LICENSE INCLUDES NO WARRANTIES, EXPRESSED OR IMPLIED, WHETHER ORAL OR WRITTEN, WITH RESPECT TO THE SOFTWARE OR OTHER MATERIAL INCLUDING, BUT NOT LIMITED TO, ANY IMPLIED WARRANTIES OF MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE, OR ARISING FROM A COURSE OF PERFORMANCE OR DEALING, OR FROM USAGE OR TRADE, OR OF NON-INFRINGEMENT OF ANY PATENTS OF THIRD PARTIES. THE INFORMATION IN THIS DOCUMENT SHOULD NOT BE CONSTRUED AS A COMMITMENT OF DEVELOPMENT BY ANY PARTY.

The US Government has a license under these copyrights, and this material may be reproduced by or for the US Government.

Introduction

This document defines interpolation functionality for VSIPL.

For VSIPL interpolation an initial set of data points `(x,y)` are obtained either through a measurement process (such as collecting data from a set of sensors), or calculated from some functional relationship. Each `y` data point is associated with an `x` data point. The `x` data points are ordered from smallest to largest. The term *node* is used to describe a particular `(x,y)` pair.

The data `x` is stored in a vector **`x`** and the data y is stored in a vector **`y`**. If **`x`** and **`y`** are of length n then the range of **`x`** is [**`x`**(0),**`x`**(n-1)].

Users should note that node values contained in **`x`** must be ordered from smallest to largest and must not be the same. This means that **`x`**`[i]` is strictly less than **`x`**`[i+1]` and may not be equal.

Interpolation is the process of creating a function $y = f(x)$ such that if $x = \mathrm{x(i)}$ then $y = \mathrm{y(i)}$ for for any x residing in the range of **`x`**. Values of y calculated for values of x which are not nodes are interpolated values. Of course there is no guarantee that interpolated values correspond to any physical or mathematical truth. The usefulness of interpolated values depends upon the selection of the interpolation method and the physical data being interpolated.

Interpolation methods are well known[1][2] and it would be pointless to try to define interpolation in this document. For VSIPL the interpolation methods for which function APIs are defined are nearest, piecewise linear, and natural cubic spline.

VSIPL does not exactly define the algorithm and functionality for each interpolation method. Library implementors methods might differ although linear and natural cubic spline methods are fairly standard and there is not much room for variance.

Nearest interpolation methods do not appear to be well defined although this method is easiest to implement. In particular the method for deciding the interpolation value if an x value lies exactly between two node points is problematic.

The user should always test an interpolation algorithm if exact numerical equivalence between two library vendors is a requirement. If needed users should write their own routine to meet their numerical requirement. Interpolation is not an exact science.

1 Cheny and Kincaid. *Numerical Mathematics And Computing,* 1985
2 Atkinson and Han. *Elementary Numerical Analysis,2004*

Interpolation Fundamentals

VSIPL interpolating functions will take as input data the node set described above in the introduction, and a vector of x values for which interpolated values are desired.

Three interpolation functions are defined;

`vsip_vinterp_nearest_p`,

`vsip_vinterp_linear_p`

and

`vsip_vinterp_spline_p`.

For the spline method a vendor dependent interpolation object is defined to allow for early binding of work space which may be needed for best performance using the cubic spline interpolation. This spline object requires the definition of create and destroy support functions as well as the spline function.

The nearest and linear methods require no interpolation object so have no additional support functionality.

This document also defines matrix interpolation; however matrix interpolation is basically an iteration of vector interpolation over rows or columns of the matrix.

Interpolation Type Definitions

```
struct vsip_splinestruct_p; /* vendor dependent */

typedef struct vsip_splinestruct_p vsip_spline_p;
```

Interpolation

The following represent interpolation operations defined in VSIPL.

`vsip_`**`spline_create`**`_`*`p`*	Create a (cubic) spline object
`vsip_`**`spline_destroy`**`_`*`p`*	Destroy a spline object
`vsip_`**`vinterp_spline`**`_`*`p`*	Perform a (cubic) spline interpolation
`vsip_`**`minterp_spline`**`_`*`p`*	Perform a (cubic) spline interpolation on a matrix (by row or by column).
`vsip_`**`vinterp_nearest`**`_`*`p`*	Perform a nearest neighbor interpolation.
`vsip_`**`minterp_nearest`**`_`*`p`*	Perform a nearest neighbor interpolation on a matrix (by row or by column).
`vsip_`**`vinterp_linear`**`_`*`p`*	Perform a linear interpolation.
`vsip_`**`minterp_linear`**`_`*`p`*	Perform a linear interpolation on a matrix (by row or by column)

`vsip_spline_create_p` Spline Object Create

Create a spline object.

Functionality

Create a spline object. The spline object is available to allow early binding for any data space the library implementor might need for the spline interpolation function.

Prototypes

```
vsip_spline_p* vsip_spline_create_p(
     vsip_length max);
```

Arguments

`max` Maximum number of known input data points. N must be greater than 2.

Return Value

Pointer to created spline object or null on create failure.

Restrictions

Errors

1. The length argument for the longest spline supported must be greater than 2.

Notes/References

The object must be created with a size equal to or greater than the size of the known data input to the interpolation function. For a vector spline fit `max` must be greater than or equal to the length of the input vector of known (`y`) values. For a matrix spline fit `max` must be greater than or equal to the product of the row length and the column length of the input matrix of known (`y`) values.

This object is reused as needed. It has no history requirement.

Implementors should note that a development mode implementation must have an attribute which indicates a valid object. The spline object is set to valid after a successful create. This allows error code to check for a valid object.

Examples

See example in `vsip_vinterspline_p` API page.

See Also

`vsip_spline_destroy_p`, `vsip_vinterp_spline_p`, `vsip_minterp_spline_p`

`vsip_spline_destroy_p` Spline Object Destroy

Destroy a spline object.

Functionality

Free any memory allocated when the spline object was created.

Prototypes

```
void vsip_spline_destroy_p(
     vsip_spline_p *spl);
```

Arguments

`spl` Spline object to be destroyed or null.

Return Value

None.

Restrictions

Errors

1. The spline object must be valid or `NULL`.

Notes/References

It is not an error to destroy a null spline object.

A development mode implementation will set the object to invalid before freeing the object.

Examples

See example in `vsip_vinterspline_p` API page.

See Also

`vsip_spline_create_p`, `vsip_vinterp_spline_p`, `vsip_minterp_spline_p`.

`vsip_vinterp_spline_p` Vector Interpolate Spline

Calculate interpolated values using the cubic spline method.

Functionality

This function implements a natural cubic spline interpolation method. A natural cubic spline assumes a cubic polynomial of degree less than or equal to three. The polynomial constants are calculated piecewise between each of the consecutive node points using boundary conditions that the first and second derivative of the polynomial is continuous at each node point. For a natural cubic spline the second derivative is defined as zero at the first and last node.

Prototypes

```
void vsip_vinterp_spline_p(
     const vsip_vview_p *x0,
     const vsip_vview_p *y0,
     vsip_spline_p *spl,
     const vsip_vview_p *x,
     const vsip_vview_p *y);
```

Arguments

`x0`	Ordered (smallest to largest) vector of input x values.
`y0`	Vector of input `y` values.
`spl`	spline object
`x`	Input vector of `x` values for which an interpolated `y` value is requested. This vector is ordered from smallest to largest.
`y`	Output `y` vector of the same length as `x`.

Return Value

None.

Restrictions

Errors

1. The view sizes must be conformant.
2. The view objects must be valid.

3. The spline object must be valid.
4. The spline object must be conformant.

Notes/References

No in-place functionality is defined for this function.

A conformant spline object will have been created with enough space. If enough space is not available a development mode implementation will fail. For a production mode library using a non-conformant spline object the result is implementation dependent.

Examples

```
#include <stdio.h>
#include <vsip.h>

/* Below we implement example from "help interp1" in octave 2.9.9
*              xf=[0:0.05:10]; yf = sin(2*pi*xf/5);
*              xp=[0:10];      yp = sin(2*pi*xp/5);
*              spl=interp1(xp,yp,xf,'spline');
*/

#define VPRINT(_x) { vsip_length L = vsip_vgetlength_f(_x); \
    vsip_index i; printf("[\n");\
    for(i=0; i< L; i++) printf("%5.4f;\n",vsip_vget_f(_x,i));\
    printf("];\n"); }

int main (int argc, const char * argv[]) {
   int retval = vsip_init((void*)0);
   vsip_length N0 = 11;
   vsip_spline_f *spl = vsip_spline_create_f(N0);
   vsip_length N  = 201;
   vsip_vview_f *yf = vsip_vcreate_f(N,VSIP_MEM_NONE);
   vsip_vview_f *xf = vsip_vcreate_f(N,VSIP_MEM_NONE);
   vsip_vview_f *xp = vsip_vcreate_f(N0,VSIP_MEM_NONE);
   vsip_vview_f *yp = vsip_vcreate_f(N0,VSIP_MEM_NONE);
   vsip_vramp_f(0.0,1.0,xp);
   vsip_vramp_f(0.0,0.05,xf);
   vsip_svmul_f(2.0/5.0 * M_PI,xp,yp);
   vsip_vsin_f(yp,yp);
   printf("xp = ");VPRINT(xp);
   printf("yp = ");VPRINT(yp);
```

```
    printf("xf = ");VPRINT(xf);
    vsip_vinterp_spline_f(xp,yp,spl,xf,yf);
    printf("spline = ");VPRINT(yf);

    vsip_spline_destroy_f(spl);
    vsip_valldestroy_f(xf);
    vsip_valldestroy_f(xp);
    vsip_valldestroy_f(yp);
    vsip_valldestroy_f(yf);
    vsip_finalize((void*)0);
    return retval;
}
```

Output from this code is too long to include in this document. To test redirect the output to an octave file (<file_name>.m) and run the file in octave. This will input the data into the octave environment allowing easy plotting of input and output data for comparison.

See Also

`vsip_spline_create_p`, `vsip_spline_destroy_p`, `vsip_minterp_spline_p`

`vsip_minterp_spline_p` Matrix Interpolate Spline

Calculate interpolated values by row or by column using the cubic spline method.

Functionality

Given a measured initial data set contained in vector `x0` and a matrix `y0` where `x0` is ordered from smallest to largest. The vector `x0` is associated with `y0` by rows or by columns. If `x0` is associated with `y0` by rows then a vector cubic spline interpolation is done for each row in the matrix. If `x0` is associated with `y0` by columns then a cubic spline interpolation is done for each column in the matrix.

Prototypes

```
void vsip_minterp_spline_p(
     const vsip_vview_p *x0,
     const vsip_mview_p *y0,
     vsip_spline_p *spl,
     vsip_major dim,
     const vsip_vview_p *x,
     const vsip_mview_p *y);
```

Arguments

`x0`	Ordered (smallest to largest) vector of input node values.
`y0`	Matrix of input node `y` values.
`spl`	spline object
`dim`	Indicates interpolation by row or by column.
`x`	Input vector of `x` values for which an interpolated `y` value is requested. This vector is ordered from smallest to largest.
`y`	Output `y` matrix.

Return Value

Restrictions

Errors

1. The view sizes must be conformant.
2. The view objects must be valid.

3. The spline object must be valid.
4. The spline object must be conformant.

Notes/References

No in-place functionality is defined for this function.

Note a conformant spline object will have been created with enough space. If enough space is not available a development mode implementation will fail. A production mode implementations action is implementation dependent.

For a dim argument of `VSIP_ROW` (`VSIP_COL`) the row (column) size of the output `y` matrix is equal to the length of the `x` vector and the column (row) size is equal to the column (row) size of the input `y0` matrix.

Examples

```
#include <stdio.h>
#include <vsip.h>

#define VPRINT(_x) { vsip_length L = vsip_vgetlength_f(_x); \
    vsip_index i; printf("[\n");\
    for(i=0; i< L; i++) printf("%5.4f;\n",vsip_vget_f(_x,i));\
    printf("];\n"); }

#define MPRINT(_x) { \
    vsip_length N = vsip_mgetrowlength_f(_x); \
    vsip_length M = vsip_mgetcollength_f(_x);\
    vsip_index i,j; printf("[\n");\
    for(i=0; i< M; i++) { for(j=0; j<N; j++){ \
       printf("%5.4f ",vsip_mget_f(_x,i,j));}\
    printf(";\n");}\
    printf("];\n"); }

/* Below we implement example from "help interp1" in octave 2.9.9
*              xf=[0:0.05:10]; yf = sin(2*pi*xf/5);
*              xp=[0:10];      yp = sin(2*pi*xp/5);
*              lin=interp1(xp,yp,xf);
*/

int main (int argc, const char * argv[]) {
   int retval = vsip_init((void*)0);
```

```
    vsip_length N0 = 11;
    vsip_length N  = 201;
    vsip_length M = 3;
    vsip_spline_f *spl = vsip_spline_create_f(N0 * N);
    vsip_mview_f *yf = vsip_mcreate_f(M,N,VSIP_ROW,VSIP_MEM_NONE);
    vsip_vview_f *xf = vsip_vcreate_f(N,VSIP_MEM_NONE);
    vsip_vview_f *xp = vsip_vcreate_f(N0,VSIP_MEM_NONE);
    vsip_mview_f *yp = vsip_mcreate_f(M,N0,VSIP_COL,VSIP_MEM_NONE);
    vsip_vview_f *yp0 = vsip_mrowview_f(yp,0);
    vsip_vview_f *yp1 = vsip_mrowview_f(yp,1);
    vsip_vview_f *yp2 = vsip_mrowview_f(yp,2);
    vsip_vramp_f(0.0,1.0,xp);
    vsip_vramp_f(0.0,0.05,xf);
    vsip_svmul_f(2.0/5.0 * M_PI,xp,yp0);
    vsip_svadd_f(M_PI/8.0,yp0,yp1);
    vsip_svadd_f(M_PI/8.0,yp1,yp2);
    vsip_vsin_f(yp0,yp0);
    vsip_vsin_f(yp1,yp1);
    vsip_vsin_f(yp2,yp2);
    printf("xp = ");VPRINT(xp);
    printf("yp = ");MPRINT(yp);
    printf("xf = ");VPRINT(xf);
    vsip_minterp_spline_f(xp,yp,spl,VSIP_ROW,xf,yf);
    printf("spline = "); MPRINT(yf);

    vsip_spline_destroy_f(spl);
    vsip_vdestroy_f(yp0);
    vsip_vdestroy_f(yp1);
    vsip_vdestroy_f(yp2);
    vsip_valldestroy_f(xf);
    vsip_valldestroy_f(xp);
    vsip_malldestroy_f(yp);
    vsip_malldestroy_f(yf);
    vsip_finalize((void*)0);
    return retval;
}
```

Output from this code is too long to include in this document. To test redirect the output to an octave file (<file_name>.m) and run the file in octave. This will input the

data into the octave environment allowing easy plotting of input and output data for comparison.

See Also

`vsip_spline_create_p`, `vsip_spline_destroy_p`, `vsip_vinterp_spline_p`

`vsip_vinterp_linear_p` Vector Interpolate Linear

Calculate interpolated values using the linear method.

Functionality

The vector `x0` is an input vector of length `n` and associated with an equal length vector `y0` of known values. The vectors x0 and y0 form the node pairs. A vector `x` of values is supplied and a y vector of interpolated values is calculated.

A slope

$$s_i \leftarrow \frac{y0_{i+1} - y0_i}{x0_{i+1} - x0_i} \quad \text{for} \quad i = 0,1,\ldots,n-1$$

is calculated between consecutive node pairs. The interpolated value is then given by

$$y_j \leftarrow y0_i + s_i * (x_j - x0_i) \quad \text{for} \quad x0_i \le x_j < x0_{i+1}$$

the range of `x` is less than or equal to the range of `x0`.

Prototypes

```
void vsip_vinterp_linear_p(
     const vsip_vview_p *x0,
     const vsip_vview_p *y0,
     const vsip_vview_p *x,
     const vsip_vview_p *y);
```

Arguments

`x0`	Vector of known `x` data points.
`y0`	Vector of know `y` data points
`x`	Input vector of `x` values for which interpolated `y` values are calculated. This vector is ordered from smallest to largest.
`y`	Output `y` vector of the same length as `x`.

Return Value

Restrictions

Errors

1. The view sizes must be conformant.
2. The view objects must be valid.

Notes/References

No in-place functionality is defined for this function.

Examples

```
#include <stdio.h>
#include <vsip.h>

#define VPRINT(_x) { vsip_length L = vsip_vgetlength_f(_x); \
    vsip_index i; printf("[\n");\
    for(i=0; i< L; i++) printf("%5.4f;\n",vsip_vget_f(_x,i));\
    printf("];\n"); }


/* Below we implement example from "help interp1" in octave 2.9.9
*              xf=[0:0.05:10]; yf = sin(2*pi*xf/5);
*              xp=[0:10];      yp = sin(2*pi*xp/5);
*              lin=interp1(xp,yp,xf);
*/

int main (int argc, const char * argv[]) {
   int retval = vsip_init((void*)0);
   vsip_length N0 = 11;
   vsip_length N  = 201;
   vsip_vview_f *yf = vsip_vcreate_f(N,VSIP_MEM_NONE);
   vsip_vview_f *xf = vsip_vcreate_f(N,VSIP_MEM_NONE);
   vsip_vview_f *xp = vsip_vcreate_f(N0,VSIP_MEM_NONE);
   vsip_vview_f *yp = vsip_vcreate_f(N0,VSIP_MEM_NONE);
   vsip_vramp_f(0.0,1.0,xp);
   vsip_vramp_f(0.0,0.05,xf);
   vsip_svmul_f(2.0/5.0 * M_PI,xp,yp);
   vsip_vsin_f(yp,yp);
   printf("xp = ");VPRINT(xp);
   printf("yp = ");VPRINT(yp);
   printf("xf = ");VPRINT(xf);
   vsip_vinterp_linear_f(xp,yp,xf,yf);
   printf("linear = "); VPRINT(yf);

   vsip_valldestroy_f(xf);
```

```
    vsip_valldestroy_f(xp);
    vsip_valldestroy_f(yp);
    vsip_valldestroy_f(yf);
    vsip_finalize((void*)0);
    return retval;
}
```

Output from this code is too long to include in this document. To test redirect the output to an octave file (<file_name>.m) and run the file in octave. This will input the data into the octave environment allowing easy plotting of input and output data for comparison.

See Also

`vsip_minterp_linear_p` Matrix Interpolate Linear

Calculate interpolated values using the linear method.

Functionality

The functionality for matrix interpolate linear is the same as for vector interpolation except that the interpolation is repeated over each row or column as selected by the `dim` argument.

Prototypes

```
void vsip_minterp_linear_p(
     const vsip_vview_p *x0;
     const vsip_mview_p *y0;
     vsip_major dim,
     const vsip_vview_p *x,
     const vsip_mview_p *y);
```

Arguments

`x0`	Vector of known x data points.
`y0`	Vector of know y data points.
`dim`	Indicates interpolation by row or by column.
`x`	Input vector of `x` values for which interpolated `y values` are calculated. This vector is ordered from smallest to largest.
`y`	Output y matrix.

Return Value

Restrictions

Errors

1. The view sizes must be conformant.
2. The view objects must be valid.

Notes/References

No in-place functionality is defined for this function.

For a `dim` argument of `VSIP_ROW` (`VSIP_COL`) the row (column) size of the output `y` matrix is equal to the length of the `x` vector and the column (row) size is equal to the column (row) size of the input `y0` matrix.

Examples

```
#include <stdio.h>
#include <vsip.h>

#define VPRINT(_x) { vsip_length L = vsip_vgetlength_f(_x); \
    vsip_index i; printf("[\n");\
    for(i=0; i< L; i++) printf("%5.4f;\n",vsip_vget_f(_x,i));\
    printf("];\n"); }

#define MPRINT(_x) { \
    vsip_length N = vsip_mgetrowlength_f(_x); \
    vsip_length M = vsip_mgetcollength_f(_x);\
    vsip_index i,j; printf("[\n");\
    for(i=0; i< M; i++) { for(j=0; j<N; j++){ \
       printf("%5.4f ",vsip_mget_f(_x,i,j));}\
    printf(";\n");}\
    printf("];\n"); }

/* Below we implement example from "help interp1" in octave 2.9.9
*              xf=[0:0.05:10]; yf = sin(2*pi*xf/5);
*              xp=[0:10];      yp = sin(2*pi*xp/5);
*              lin=interp1(xp,yp,xf);
*/

int main (int argc, const char * argv[]) {
   int retval = vsip_init((void*)0);
   vsip_length N0 = 11;
   vsip_length N  = 201;
   vsip_length M = 3;
   vsip_mview_f *yf = vsip_mcreate_f(M,N,VSIP_ROW,VSIP_MEM_NONE);
   vsip_vview_f *xf = vsip_vcreate_f(N,VSIP_MEM_NONE);
   vsip_vview_f *xp = vsip_vcreate_f(N0,VSIP_MEM_NONE);
   vsip_mview_f *yp = vsip_mcreate_f(M,N0,VSIP_COL,VSIP_MEM_NONE);
   vsip_vview_f *yp0 = vsip_mrowview_f(yp,0);
   vsip_vview_f *yp1 = vsip_mrowview_f(yp,1);
   vsip_vview_f *yp2 = vsip_mrowview_f(yp,2);
   vsip_vramp_f(0.0,1.0,xp);
   vsip_vramp_f(0.0,0.05,xf);
   vsip_svmul_f(2.0/5.0 * M_PI,xp,yp0);
   vsip_svadd_f(M_PI/8.0,yp0,yp1);
   vsip_svadd_f(M_PI/8.0,yp1,yp2);
   vsip_vsin_f(yp0,yp0);
```

```
    vsip_vsin_f(yp1,yp1);
    vsip_vsin_f(yp2,yp2);
    printf("xp = ");VPRINT(xp);
    printf("yp = ");MPRINT(yp);
    printf("xf = ");VPRINT(xf);
    vsip_minterp_linear_f(xp,yp,VSIP_ROW,xf,yf);
    printf("linear = "); MPRINT(yf);

    vsip_vdestroy_f(yp0);
    vsip_vdestroy_f(yp1);
    vsip_vdestroy_f(yp2);
    vsip_valldestroy_f(xf);
    vsip_valldestroy_f(xp);
    vsip_malldestroy_f(yp);
    vsip_malldestroy_f(yf);
    vsip_finalize((void*)0);
    return retval;
}
```

Output from this code is too long to include in this document. To test redirect the output to an octave file (<file_name>.m) and run the file in octave. This will input the data into the octave environment allowing easy plotting of input and output data for comparison.

See Also

`vsip_vinterp_nearest_p` Vector Interpolate Nearest

Calculate interpolated values using the nearest neighbor method.

Functionality

The vector `x0` is an input vector of length `n` and associated with an equal length vector `y0` of known values. The vectors x0 and y0 form the node pairs. A vector `x` of values is supplied and a y vector of interpolated values is calculated.

For an `x0[i] <= x[j] < x0[i+1]` the interpolated value `y[j]` is equal to `y0[i]` if `x[j]` is closer to `x0[i]` otherwise `y[j]` is equal to `y0[j+1]`.

Prototypes

```
void vsip_vinterp_nearest_p(
     const vsip_vview_p *x0,
     const vsip_vview_p *y0,
     const vsip_vview_p *x,
     const vsip_vview_p *y);
```

Arguments

`x0`	Vector of known `x` data points.
`y0`	Vector of know `y` data points.
`x`	Input vector of `x` values for which an interpolated `y` value is requested.
`y`	Output `y` vector of the same length as `x`.

Return Value

None.

Restrictions

Errors

1. The view sizes must be conformant.
2. The view objects must be valid.

Notes/References

This function is done out-of-place and no in-place functionality exists.

Examples

```
#include <stdio.h>
#include <vsip.h>

/* Below we implement example from "help interp1" in octave 2.9.9
*              xf=[0:0.05:10]; yf = sin(2*pi*xf/5);
*              xp=[0:10];      yp = sin(2*pi*xp/5);
*              near=interp1(xp,yp,xf,'nearest');
*/

#define VPRINT(_x) { vsip_length L = vsip_vgetlength_f(_x); \
    vsip_index i; printf("[\n");\
    for(i=0; i< L; i++) printf("%5.4f;\n",vsip_vget_f(_x,i));\
    printf("];\n"); }

int main (int argc, const char * argv[]) {
   int retval = vsip_init((void*)0);
   vsip_length N0 = 11;
   vsip_length N  = 201;
   vsip_vview_f *yf = vsip_vcreate_f(N,VSIP_MEM_NONE);
   vsip_vview_f *xf = vsip_vcreate_f(N,VSIP_MEM_NONE);
   vsip_vview_f *xp = vsip_vcreate_f(N0,VSIP_MEM_NONE);
   vsip_vview_f *yp = vsip_vcreate_f(N0,VSIP_MEM_NONE);
   vsip_vramp_f(0.0,1.0,xp);
   vsip_vramp_f(0.0,0.05,xf);
   vsip_svmul_f(2.0/5.0 * M_PI,xp,yp);
   vsip_vsin_f(yp,yp);
   printf("xp = ");VPRINT(xp);
   printf("yp = ");VPRINT(yp);
   printf("xf = ");VPRINT(xf);
   vsip_vinterp_nearest_f(xp,yp,xf,yf);
   printf("nearest = "); VPRINT(yf);

   vsip_valldestroy_f(xf);
   vsip_valldestroy_f(xp);
   vsip_valldestroy_f(yp);
   vsip_valldestroy_f(yf);
   vsip_finalize((void*)0);
   return retval;
}
```

Output from this code is too long to include in this document. To test redirect the output to an octave file (<file_name>.m) and run the file in octave. This will input the

data into the octave environment allowing easy plotting of input and output data for comparison.

See Also

`vsip_minterp_nearest_p` Matrix Interpolate Nearest

Calculate interpolated values using the nearest method.

Functionality

The functionality for matrix interpolate nearest is the same as for vector interpolation except that the interpolation is repeated over each row or column as selected by the `dim` argument.

Prototypes

```
void vsip_minterp_nearest_p(
     const vsip_vview_p *x0,
     const vsip_mview_p *y0,
     vsip_major dim,
     const vsip_vview_p *x,
     const vsip_mview_p *y);
```

Arguments

`x0`	Vector of known `x` data points.
`y0`	matrix of know `y` data points.
`dim`	Indicates interpolation by row or by column.
`x`	Input vector of `x` values for which an interpolated `y` value is requested This vector is ordered from smallest to largest.
`y`	Output `y` matrix.

Return Value

Restrictions

Errors

1. The view sizes must be conformant.
2. The view objects must be valid.

Notes/References

This function is done out-of-place and no in-place functionality exists.

For a `dim` argument of `VSIP_ROW` (`VSIP_COL`) the row (column) size of the output `y` matrix is equal to the length of the `x` vector and the column (row) size is equal to the column (row) size of the input `y0` matrix.

Examples

```
#include <stdio.h>
#include <vsip.h>

#define VPRINT(_x) { vsip_length L = vsip_vgetlength_f(_x); \
    vsip_index i; printf("[\n");\
    for(i=0; i< L; i++) printf("%5.4f;\n",vsip_vget_f(_x,i));\
    printf("];\n"); }

#define MPRINT(_x) { \
    vsip_length N = vsip_mgetrowlength_f(_x); \
    vsip_length M = vsip_mgetcollength_f(_x);\
    vsip_index i,j; printf("[\n");\
    for(i=0; i< M; i++) { for(j=0; j<N; j++){ \
       printf("%5.4f ",vsip_mget_f(_x,i,j));}\
    printf(";\n");}\
    printf("];\n"); }

/* Below we implement example from "help interp1" in octave 2.9.9
*              xf=[0:0.05:10]; yf = sin(2*pi*xf/5);
*              xp=[0:10];      yp = sin(2*pi*xp/5);
*              lin=interp1(xp,yp,xf);
*/

int main (int argc, const char * argv[]) {
   int retval = vsip_init((void*)0);
   vsip_length N0 = 11;
   vsip_length N  = 201;
   vsip_length M = 3;
   vsip_mview_f *yf = vsip_mcreate_f(M,N,VSIP_ROW,VSIP_MEM_NONE);
   vsip_vview_f *xf = vsip_vcreate_f(N,VSIP_MEM_NONE);
   vsip_vview_f *xp = vsip_vcreate_f(N0,VSIP_MEM_NONE);
   vsip_mview_f *yp = vsip_mcreate_f(M,N0,VSIP_COL,VSIP_MEM_NONE);
   vsip_vview_f *yp0 = vsip_mrowview_f(yp,0);
   vsip_vview_f *yp1 = vsip_mrowview_f(yp,1);
   vsip_vview_f *yp2 = vsip_mrowview_f(yp,2);
   vsip_vramp_f(0.0,1.0,xp);
```

```
    vsip_vramp_f(0.0,0.05,xf);
    vsip_svmul_f(2.0/5.0 * M_PI,xp,yp0);
    vsip_svadd_f(M_PI/8.0,yp0,yp1);
    vsip_svadd_f(M_PI/8.0,yp1,yp2);
    vsip_vsin_f(yp0,yp0);
    vsip_vsin_f(yp1,yp1);
    vsip_vsin_f(yp2,yp2);
    printf("xp = ");VPRINT(xp);
    printf("yp = ");MPRINT(yp);
    printf("xf = ");VPRINT(xf);
    vsip_minterp_nearest_f(xp,yp,VSIP_ROW,xf,yf);
    printf("nearest = "); MPRINT(yf);
    vsip_vdestroy_f(yp0);
    vsip_vdestroy_f(yp1);
    vsip_vdestroy_f(yp2);
    vsip_valldestroy_f(xf);
    vsip_valldestroy_f(xp);
    vsip_malldestroy_f(yp);
    vsip_malldestroy_f(yf);
    vsip_finalize((void*)0);
    return retval;
}
```

Output from this code is too long to include in this document. To test redirect the output to an octave file (<file_name>.m) and run the file in octave. This will input the data into the octave environment allowing easy plotting of input and output data for comparison.

See Also

VSIPL Permute API

Version 1.0

Addendum to Version 1.3 or later of VSIPL specification

Approved by VSIPL Forum

January 31, 2008

© 2008 Georgia Tech Research Corporation, all rights reserved.

A non-exclusive, non-royalty bearing license is hereby granted to all persons to copy, modify, distribute and produce derivative works for any purpose, provided that this copyright notice and following disclaimer appear on all copies:

THIS LICENSE INCLUDES NO WARRANTIES, EXPRESSED OR IMPLIED, WHETHER ORAL OR WRITTEN, WITH RESPECT TO THE SOFTWARE OR OTHER MATERIAL INCLUDING, BUT NOT LIMITED TO, ANY IMPLIED WARRANTIES OF MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE, OR ARISING FROM A COURSE OF PERFORMANCE OR DEALING, OR FROM USAGE OR TRADE, OR OF NON-INFRINGEMENT OF ANY PATENTS OF THIRD PARTIES. THE INFORMATION IN THIS DOCUMENT SHOULD NOT BE CONSTRUED AS A COMMITMENT OF DEVELOPMENT BY ANY PARTY.

The US Government has a license under these copyrights, and this material may be reproduced by or for the US Government.

Introduction

This document defines functionality for a permute operation on VSIPL matrices. The permute is done either by row or by column. This function may be done in place.

Permute Fundamentals

We define both a single function for permute without a permute object, and a function set which uses a permute object. The single function is simpler to use but does not maintain the state of the permute vector.

The more complicated permute defines a reusable object allowing for early binding of information and data space to optimize the permutation operation.

The permute function set consists of a permute object, a create function, an initialize function, a permute operation, and a destroy function. The permute object and method are implementation dependent; however, the implementor must support in-place, as well as out of place permutes for both permute functions.

To keep the interface simple permute is only done in one dimension at a time. To permute both the rows and columns you must permute twice, once for the row permute and once for the column permute.

Functionality

A permutation matrix P is an identity matrix with rows re-ordered. Note the permute matrix will have exactly one “1” in each row and in each column. All other elements in the row and column where a one resides will be zero, and every row and column will contain a one.

The matrix product

$$C \leftarrow P\,A$$

will permute the rows of and A place the result in C, and the matrix product

$$C \leftarrow A\,P$$

will permute the columns of A and place the result in C.

For example (row permute)

$$\begin{bmatrix} a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} \\ a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} \\ a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} \\ a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} \end{bmatrix} \leftarrow \begin{bmatrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} \\ a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} \end{bmatrix}$$

and (column permute)

$$\begin{bmatrix} a_{0,1} & a_{0,3} & a_{0,0} & a_{0,2} \\ a_{1,1} & a_{1,3} & a_{1,0} & a_{1,2} \\ a_{2,1} & a_{2,3} & a_{2,0} & a_{2,2} \\ a_{3,1} & a_{3,3} & a_{3,0} & a_{3,2} \end{bmatrix} \leftarrow \begin{bmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} \\ a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} \end{bmatrix} \begin{bmatrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{bmatrix}$$

Although permutation can be done with a matrix multiply, this is not an efficient mechanism. In addition the permute matrix P is sparse and not very handy to work with for the actual permute operation. Note that P is always square and of size column length for row permutes and of size row length for column permutes. The common mechanism is to create an index vector of the proper size (size of permute matrix) and to store the information contained in the P matrix by storing the location of the ones in a vector P. Basically if a one is located at P matrix location (i, j) then set $p(i)$ equal to j. For the example above P would be indicated by

$$p \leftarrow [2\ 0\ 3\ 1]$$

Note the permutation vector is defined the same for both row and column permutations.

Type Definitions for Permute

The implementation dependent type `vsip_permute` is defined for the permute object. Note that the permute object name does not contain depth, precision, or shape. Having a single type for permute means only a single destroy and a single initialize function are needed; however the create function and the permute function have depth, precision and shape to allow the implementor freedom to properly allocate any temporary space in the permute object.

```
struct vsip_permuteattributes;/* vendor dependent */

typedef struct vsip_permuteattributes vsip_permute;
```

Permutation

The following man pages represent permutation operations defined in VSIPL.

`vsip_`*d*`mpermute_create_`*p*	Create a permute object.
`vsip_permute_init`	Initialize a permute object
`vsip_permute_destroy`	Free memory associated with a permute object
`vsip_`*d*`mpermute_`*p*	Permute a matrix using a reusable permute object.
`vsip_`*d*`mpermute_once_`*p*	Permute a matrix using a permutation vector. The vector is modified in the permutation so may only be used once unless reinitialized.

`vsip_`*d*`mpermute_create_`*p* — Permute Create

Create a permute object.

Functionality

Allocate memory for a permute object and any necessary memory needed for all permute functionality. At create an object is created that can permute either by row or by column. The permute direction (row or column) may not be changed after the permute object is created, and the size of the matrix which can be permuted is also set at create time. All permute objects have the same type; however every permute create is keyed to the type of matrix that will be permuted.

Prototypes

```
vsip_permute* vsip_dmpermute_create_p(
     vsip_length m, /* number of rows */
     vsip_length n, /* number of columns */
     vsip_major major); /* permute dimension */
```

Arguments

`m` Number of rows in matrix to be permuted.

`n` Number of columns in matrix to be permuted.

`major` Permute direction. VSIP_ROW implies that rows are permuted and VSIP_COL implies that columns are permuted.

Return Value

Pointer to permute object or `NULL` on failure.

Restrictions

Errors

1. The lengths `m` and `n` must be greater than zero.
2. The `major` argument must be valid.

Notes/References

The permute direction is set with the create function and may not be reset. If permutes in both directions are required then they must be done with two different permute objects; one for row permutes and one for column permutes.

This function is to provide early binding for permute functionality. The returned permute object must be initialized before use using `vsip_permute_init`.

A development mode permute create function must set state information indicating a valid object on create. This allows error checking for a valid permute object. After this call the object will either be valid or `NULL`.

Examples

See example in `vsip_`*d*`mpermute_`*p* description page.

See Also

`vsip_permute_init` Permute Initialize

Initialize (or re-initialize) permute object.

Functionality

This function initializes a permute object with a permute vector. The permute vector is not attached to the permute object and is not modified by the initialization phase. A permute object may be re-initialized by calling this function with a new permute vector. The information stored in a permute object is not changed when `vsip_`*`dm`*`-permute_`*`p`* is called.

Prototypes

```
vsip_permute* vsip_permute_init(
     vsip_permute* perm, /* permute object */
     const vsip_vview_vi* p); /* permute index */
```

Arguments

`perm`	Permute object.
p	Index vector of rows or columns to permute.

Return Value

Convenience pointer to permute object.

Restrictions

The index vector p must be a valid permute vector. A valid permute vector will have one index entry for each indexed element of the permute vector. There will be exactly one 0 index and exactly one index $n-1$ where n is the length of the vector p. Every element in the index vector will have a value less than n, and no value will be used more than once. The result of an invalid index vector are implementation dependent.

Errors

1. The vector p must match the size of matrix and associated permute direction used in the permute create function (conformant).
2. The vector p must be valid.
3. The permute object must be valid or `NULL`.

Notes/References

If this function is called with a `NULL` permute object it does nothing and returns `NULL`.

Examples

See example in `vsip_`*d*`mpermute_`*p* description page.

See Also

`vsip_permute_destroy` Permute Destroy

Destroy a permute object.

Functionality

Free memory associated with a permute object.

Prototypes

```
void vsip_permute_destroy(
     vsip_permute *perm);
```

Arguments

perm	Valid permute object to destroy or `NULL`.

Return Value

`None.`

Restrictions

The permute object must be valid or `NULL`.

Errors

1.The permute object must be valid or `NULL`

Notes/References

A development mode permute destroy function must set the permute object to invalid before destroying the permute object. Since the object may remain in memory after destruction setting it to invalid will allow error code to determine it is invalid even if the object is not overwritten.

Examples

See example in `vsip_`*d*`mpermute_`*p* description page.

See Also

vsip_*d*mpermute_*p* — Matrix Permute

Permute a matrix.

Functionality

Using an initialized permute object permute an input matrix and place the output in an output matrix. If the input matrix and the output matrix is the same as the input matrix then an in-place operation occurs. If the output matrix is not the same as the input matrix then elements that are not permuted are copied into the corresponding element of the output matrix.

Prototypes

```
void vsip_dmpermute_p(
     const vsip_dmview_p *A,
     const vsip_permute *perm,
     const vsip_dmview_p *C);
```

Arguments

`A`	Matrix view of input matrix.
`perm`	Conforming permute object created with permute create function `vsip_dmpermute_create_p`
`C`	Matrix view of input matrix.

Return Value

`None.`

Restrictions

The permute object must have been made with the proper create function. The result of using a permute object which was not created for the proper matrix type is implementation dependent.

The size of the input matrix must equal the size used when the permute object was created.

If A and C reference the same data space (in-place operation) they must reference the exact same data space.

Errors

1. The matrix A and C must be conformant with each other.
2. The permute object must be conformant.
3. The permute object must be valid.

4. The views must be valid.

Notes/References

For the permute object to be conformant it must have been created with proper sizes and initialized.

Examples

```
#include<vsip.h>

#define PRINTM(_A)  {for(i=0; i<10; i++){ for(j=0; j<10; j++){ \
         printf("%3.1f, ",vsip_mget_f(_A,i,j)); } printf("\n");}}

#define INIT_DATA {for(i=0; i<10; i++){for(j=0; j<10; j++){ \
         vsip_mput_f(indta,i,j,(float)j/10.0 + (float)i ); } }}

int main(){
   int retval = vsip_init((void*)0);
   int i,j;
   vsip_vview_vi *p = vsip_vcreate_vi(10,VSIP_MEM_NONE);
   /* permute data for p */
   vsip_scalar_vi vi[10]={4, 3, 2, 1, 6, 5, 0, 7, 8, 9};
   vsip_mview_f *indta =\
                  vsip_mcreate_f(10,10,VSIP_ROW,VSIP_MEM_NONE);
   vsip_mview_f *outdta =\
                  vsip_mcreate_f(10,10,VSIP_COL,VSIP_MEM_NONE);
   vsip_permute* perm;
   /* Example of by row */
   INIT_DATA
   for(i=0; i<10; i++){ /* initialize vector p */
      vsip_vput_vi(p,i,vi[i]);
   }
   perm = vsip_mpermute_create_f(10,10,VSIP_ROW);
   vsip_permute_init(perm,p);  /* initialize the object with p */
   printf("permute vector p\n"); /* print vector p */
   for(i=0; i<10; i++){
      printf("%2d,",(int)vi[i]);
   }

   printf("\ninput\n"); PRINTM(indta);

   vsip_mpermute_f(indta, perm, outdta); /* permute out of place */
   printf("\noutput (by row out-of-place)\n"); PRINTM(outdta);
```

```
    vsip_mpermute_f(indta, perm, indta);
    printf("\noutput (by row in-place)\n"); PRINTM(indta);

    /* re-init input matrix and destroy and create new perm object */
    INIT_DATA
    vsip_permute_destroy(perm); /* destroy old permutation object */
    perm = vsip_mpermute_create_f(10,10,VSIP_COL);
    vsip_permute_init(perm,p);/* initialize the object with p */

    vsip_mpermute_f(indta, perm, outdta); /* permute out of place */
    printf("\noutput (by column out-of-place)\n"); PRINTM(outdta);

    vsip_mpermute_f(indta, perm, indta);
    printf("\noutput (by column in-place)\n"); PRINTM(indta);

    vsip_permute_destroy(perm);
    vsip_valldestroy_vi(p);
    vsip_malldestroy_f(indta);
    vsip_malldestroy_f(outdta);
    vsip_finalize((void*)0);
    return retval;
}
```

Output of the example should be as follows:

```
permute vector p
 4, 3, 2, 1, 6, 5, 0, 7, 8, 9,
input
0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9,
1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 1.8, 1.9,
2.0, 2.1, 2.2, 2.3, 2.4, 2.5, 2.6, 2.7, 2.8, 2.9,
3.0, 3.1, 3.2, 3.3, 3.4, 3.5, 3.6, 3.7, 3.8, 3.9,
4.0, 4.1, 4.2, 4.3, 4.4, 4.5, 4.6, 4.7, 4.8, 4.9,
5.0, 5.1, 5.2, 5.3, 5.4, 5.5, 5.6, 5.7, 5.8, 5.9,
6.0, 6.1, 6.2, 6.3, 6.4, 6.5, 6.6, 6.7, 6.8, 6.9,
7.0, 7.1, 7.2, 7.3, 7.4, 7.5, 7.6, 7.7, 7.8, 7.9,
8.0, 8.1, 8.2, 8.3, 8.4, 8.5, 8.6, 8.7, 8.8, 8.9,
9.0, 9.1, 9.2, 9.3, 9.4, 9.5, 9.6, 9.7, 9.8, 9.9,

output (by row out-of-place)
4.0, 4.1, 4.2, 4.3, 4.4, 4.5, 4.6, 4.7, 4.8, 4.9,
3.0, 3.1, 3.2, 3.3, 3.4, 3.5, 3.6, 3.7, 3.8, 3.9,
2.0, 2.1, 2.2, 2.3, 2.4, 2.5, 2.6, 2.7, 2.8, 2.9,
1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 1.8, 1.9,
6.0, 6.1, 6.2, 6.3, 6.4, 6.5, 6.6, 6.7, 6.8, 6.9,
```

```
5.0, 5.1, 5.2, 5.3, 5.4, 5.5, 5.6, 5.7, 5.8, 5.9,
0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9,
7.0, 7.1, 7.2, 7.3, 7.4, 7.5, 7.6, 7.7, 7.8, 7.9,
8.0, 8.1, 8.2, 8.3, 8.4, 8.5, 8.6, 8.7, 8.8, 8.9,
9.0, 9.1, 9.2, 9.3, 9.4, 9.5, 9.6, 9.7, 9.8, 9.9,

output (by row in-place)
4.0, 4.1, 4.2, 4.3, 4.4, 4.5, 4.6, 4.7, 4.8, 4.9,
3.0, 3.1, 3.2, 3.3, 3.4, 3.5, 3.6, 3.7, 3.8, 3.9,
2.0, 2.1, 2.2, 2.3, 2.4, 2.5, 2.6, 2.7, 2.8, 2.9,
1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 1.8, 1.9,
6.0, 6.1, 6.2, 6.3, 6.4, 6.5, 6.6, 6.7, 6.8, 6.9,
5.0, 5.1, 5.2, 5.3, 5.4, 5.5, 5.6, 5.7, 5.8, 5.9,
0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9,
7.0, 7.1, 7.2, 7.3, 7.4, 7.5, 7.6, 7.7, 7.8, 7.9,
8.0, 8.1, 8.2, 8.3, 8.4, 8.5, 8.6, 8.7, 8.8, 8.9,
9.0, 9.1, 9.2, 9.3, 9.4, 9.5, 9.6, 9.7, 9.8, 9.9,

output (by column out-of-place)
0.4, 0.3, 0.2, 0.1, 0.6, 0.5, 0.0, 0.7, 0.8, 0.9,
1.4, 1.3, 1.2, 1.1, 1.6, 1.5, 1.0, 1.7, 1.8, 1.9,
2.4, 2.3, 2.2, 2.1, 2.6, 2.5, 2.0, 2.7, 2.8, 2.9,
3.4, 3.3, 3.2, 3.1, 3.6, 3.5, 3.0, 3.7, 3.8, 3.9,
4.4, 4.3, 4.2, 4.1, 4.6, 4.5, 4.0, 4.7, 4.8, 4.9,
5.4, 5.3, 5.2, 5.1, 5.6, 5.5, 5.0, 5.7, 5.8, 5.9,
6.4, 6.3, 6.2, 6.1, 6.6, 6.5, 6.0, 6.7, 6.8, 6.9,
7.4, 7.3, 7.2, 7.1, 7.6, 7.5, 7.0, 7.7, 7.8, 7.9,
8.4, 8.3, 8.2, 8.1, 8.6, 8.5, 8.0, 8.7, 8.8, 8.9,
9.4, 9.3, 9.2, 9.1, 9.6, 9.5, 9.0, 9.7, 9.8, 9.9,

output (by column in-place)
0.4, 0.3, 0.2, 0.1, 0.6, 0.5, 0.0, 0.7, 0.8, 0.9,
1.4, 1.3, 1.2, 1.1, 1.6, 1.5, 1.0, 1.7, 1.8, 1.9,
2.4, 2.3, 2.2, 2.1, 2.6, 2.5, 2.0, 2.7, 2.8, 2.9,
3.4, 3.3, 3.2, 3.1, 3.6, 3.5, 3.0, 3.7, 3.8, 3.9,
4.4, 4.3, 4.2, 4.1, 4.6, 4.5, 4.0, 4.7, 4.8, 4.9,
5.4, 5.3, 5.2, 5.1, 5.6, 5.5, 5.0, 5.7, 5.8, 5.9,
6.4, 6.3, 6.2, 6.1, 6.6, 6.5, 6.0, 6.7, 6.8, 6.9,
7.4, 7.3, 7.2, 7.1, 7.6, 7.5, 7.0, 7.7, 7.8, 7.9,
8.4, 8.3, 8.2, 8.1, 8.6, 8.5, 8.0, 8.7, 8.8, 8.9,
9.4, 9.3, 9.2, 9.1, 9.6, 9.5, 9.0, 9.7, 9.8, 9.9,
```

See Also

vsip_*d*mpermute_once_*p* Matrix Permute Once

Permute a matrix.

Functionality

Using a permute vector permute an input matrix placing the output in an output matrix. If the input matrix and output matrix are the same then an in-place operation occurs. If the output matrix is not the same as the input matrix then elements that are not permuted are copied into the corresponding element of the output matrix. The permute vector may be modified by the permute operation.

Prototypes

```
void vsip_dmpermute_once_p(
     const vsip_dmview_p *A,
     vsip_major major,
     const vsip_vview_vi *p,
     const vsip_dmview_p *C);
```

Arguments

`A`	Matrix view of input matrix.
`major`	Permute direction. VSIP_ROW implies that rows are permuted and VSIP_COL implies that columns are permuted.
`p`	Index vector of rows or columns to permute.
`C`	Matrix view of output matrix.

Return Value

None.

Restrictions

The permute vector must be a proper permute vector. Proper means each row or column is indexed exactly one time.

If `A` and `C` reference the same data space (in-place operation) they must reference the exact same data space.

Errors

1. The matrix A and C must be conformant with each other.
2. The permute vector must be conformant with the size of the input/output matrices and the direction (major) of permutation.
3. The views must be valid.

Notes/References

Examples

```
#include<vsip.h>

#define PRINTM(_A)  {for(i=0; i<10; i++){ for(j=0; j<10; j++){ \
         printf("%3.1f, ",vsip_mget_f(_A,i,j)); } printf("\n");}}

#define INIT_DATA {for(i=0; i<10; i++){for(j=0; j<10; j++){ \
         vsip_mput_f(indta,i,j,(float)j/10.0 + (float)i ); } }}

#define INIT_P {for(i=0; i<10; i++){vsip_vput_vi(p,i,vi[i]); } }

int main(){
   int retval = vsip_init((void*)0);
   int i,j;
   vsip_vview_vi *p = vsip_vcreate_vi(10,VSIP_MEM_NONE);
   /* permute data for p */
   vsip_scalar_vi vi[10]={4, 3, 2, 1, 6, 5, 0, 7, 8, 9};
   vsip_mview_f *indta =\
                  vsip_mcreate_f(10,10,VSIP_ROW,VSIP_MEM_NONE);
   vsip_mview_f *outdta =\
                  vsip_mcreate_f(10,10,VSIP_COL,VSIP_MEM_NONE);
   /* Example of by row */
   INIT_DATA
   INIT_P
   printf("permute vector p\n"); /* print vector p */
   for(i=0; i<10; i++){
      printf("%2d,",(int)vi[i]);
   }

   printf("\ninput\n"); PRINTM(indta);

   vsip_mpermute_once_f(indta,VSIP_ROW,p,outdta); /* out-of-place */
   printf("\noutput (by row out-of-place)\n"); PRINTM(outdta);

   INIT_P
   vsip_mpermute_once_f(indta, VSIP_ROW, p, indta); /* in-place */
   printf("\noutput (by row in-place)\n"); PRINTM(indta);

   /* re-init input matrix */
```

```
    INIT_DATA
    INIT_P
    vsip_mpermute_once_f(indta,VSIP_COL,p,outdta); /* out-of-place */
    printf("\noutput (by column out-of-place)\n"); PRINTM(outdta);

    INIT_P
    vsip_mpermute_once_f(indta, VSIP_COL, p, indta); /* in-place */
    printf("\noutput (by column in-place)\n"); PRINTM(indta);

    vsip_valldestroy_vi(p);
    vsip_malldestroy_f(indta);
    vsip_malldestroy_f(outdta);
    vsip_finalize((void*)0);
    return retval;
 }
```

Output of example is the same as that for `vsip_`*d*`mpermute_`*p*.

See Also

VSIPL Sort API

Version 1.0

Addendum to Version 1.3 or later of VSIPL specification

Approved by VSIPL Forum

January 31, 2008

© 2008 Georgia Tech Research Corporation, all rights reserved.

A non-exclusive, non-royalty bearing license is hereby granted to all persons to copy, modify, distribute and produce derivative works for any purpose, provided that this copyright notice and following disclaimer appear on all copies:

THIS LICENSE INCLUDES NO WARRANTIES, EXPRESSED OR IMPLIED, WHETHER ORAL OR WRITTEN, WITH RESPECT TO THE SOFTWARE OR OTHER MATERIAL INCLUDING, BUT NOT LIMITED TO, ANY IMPLIED WARRANTIES OF MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE, OR ARISING FROM A COURSE OF PERFORMANCE OR DEALING, OR FROM USAGE OR TRADE, OR OF NON-INFRINGEMENT OF ANY PATENTS OF THIRD PARTIES. THE INFORMATION IN THIS DOCUMENT SHOULD NOT BE CONSTRUED AS A COMMITMENT OF DEVELOPMENT BY ANY PARTY.

The US Government has a license under these copyrights, and this material may be reproduced by or for the US Government.

Introduction

The goal of this document is to define functionality for sorting VSIPL vectors. No matrix sort is currently defined although it is straightforward to write user functions which will accomplish simple sorts by row or column using the vector sort. This document is an addendum to the VSIPL specification and does not stand-alone.

VSIPL Sort Fundamentals

All sort functionality in VSIPL is defined only on data of type real. Vectors of type complex may be sorted using gather functionality and an index vector (see *select sort* section below).

Sorts are done in place. If the original vector is desired it must be copied to a new vector before calling the function.

If an index vector which records the sort permutation is desired then an index vector should be created and passed in as an argument. If the index vector argument is `NULL` then the sort permutation is not recorded. A bool argument of `VSIP_TRUE` for `fill` indicates that the (non-null) index vector should be initialized as a ramp starting at zero with a unit increment. A bool argument of `VSIP_FALSE` for `fill` means the input index vector values should not be modified. The input index vector is permuted element-wise to match the permutation of the data vector.

Note that, for instance, a returned index vector that was initialized with the `fill` argument set to true will allow the permutation of an associated vector by using VSIPL gather functionality.

Select Sort

A function called *select sort* was originally envisioned for this section; however it quickly became complicated. The API was complicated to design, and the function was hard to write and difficult to use. The ideas behind select sort are strait-forward and easy to implement using other library functionality, and performance gains from a dedicated function don't seem to be worthwhile. So instead of select sort this section is included as a *clue* to better optimize your sorts. The section also supplies clues on why we have included the index vector as part of the sort functionality.

There are many different requirements for sorting which are application specific. In particular one may have a very large amount of data to sort through even though only a small portion is of interest. For instance data greater than some value; or there may be some other condition or a set of conditions which define the data of interest.

The point here is that it is computationally expensive to sort data and sorting data of no interest is not efficient. Before doing the sort first the application should prune off unwanted values using *logical* and/or *selection* operations. Boolean vector result of true/false values from the logical operators can be turned into index vectors using the

`indexbool` function (see selection operations from the main specification). Gather functionality allows the collection of the proper data to be sorted. If an index relationship back to the original data is required then the index vector from the boolean operation which is used to gather the boolean data is passed in as the index argument and the fill flag is set to false. This index vector is permuted to match the data permutation and the index relationship of the sorted data back to the original vector is maintained.

This type sorting is also used for sorting complex vectors. Since only vectors of type real may be sorted the implication is that the complex vector must have some operation on it that converts it from complex to real. By maintaining an index vector through all the operations and passing this into sort then the sort results may be translated back to the complex data.

Type Definitions for Sort

The following are type definitions for VSIPL sort functionality.

```
typedef enum {
     VSIP_SORT_ASCENDING = 0,
     VSIP_SORT_DESCENDING = 1
} vsip_sort_dir;
typedef enum {
     VSIP_SORT_BYVALUE   = 0,
     VSIP_SORT_BYMAGNITUDE = 1,
} vsip_sort_mode;
```

The enumerated value `VSIP_SORT_ASCENDING` will cause a sort so that the smaller value after the mode condition is evaluated has the smaller index (smallest to largest).

The enumerated value `VSIP_SORT_DESCENDING` will cause a sort so that the largest value after the mode condition is evaluated has the smaller index (largest to smallest).

The enumerated value `VSIP_SORT_BYVALUE` will cause the value to be sorted as is.

The enumerated value `VSIP_SORT_BYMAGNITUDE` will cause the sort to be done on the absolute value.

Sort

The following man pages represent sort operations defined in VSIPL.

`vsip_`**`vsortip_`**`p` Vector sort in-place.

vsip_vsortip_*p* — Sort a Vector In-Place

Sort data in a vector.

Functionality

Sort a data vector in ascending or descending order either by value or by magnitude. If a valid index vector is included as the `index` argument then that vector is permuted to match the permutation of the data vector. If the boolean `fill` argument is true and the index vector is valid then the index vector is initialized with a ramp index starting at zero with increment one. If the index vector is null it and the fill flag are ignored.

Prototypes

```
void vsip_vsortip_p(
      const vsip_vview_p *a,
      vsip_sort_mode mode,
      vsip_sort_dir dir,
      vsip_bool fill,
      const vsip_vview_vi *index);
typedef enum {
      VSIP_SORT_ASCENDING = 0,    /* smaller values first     */
      VSIP_SORT_DESCENDING = 1    /* larger values first      */
} vsip_sort_dir;
typedef enum {
      VSIP_SORT_BYVALUE   = 0, /* sort by value */
      VSIP_SORT_BYMAGNITUDE = 1 /* sort by magnitude */
} vsip_sort_mode;
```

Arguments

`a`	Vector/Matrix view of input/output data.
`mode`	Sort type.
`dir`	Sort direction.
`fill`	Indicates if the function should initialize the index vector (`VSIP_TRUE`); or assume the index vector has already been initialized (`VSIP_FALSE`).
`index`	An index vector of the proper type to store the original index of the sorted values or null if no index vector is desired. The length of a non-null index vector index length must match the total number of values in the input views.

Return Value

Restrictions

Errors

1. The index argument must be a valid index vector or `NULL`.
2. The views must be conformant.
3. The view objects must be valid.
4. Arguments for `mode`, `dir`, and `fill` must be valid.

Notes/References

Examples

This example creates a complex vector of random values and then sorts the vector according to magnitude.

```
#include<stdio.h>
#include<vsip.h>

#define NSIZE 10
#define SEED 4
#define PROCS 1
#define ID 1

#define CPRINT(_v) { /* complex vector print macro */\
   int i; \
   printf("\n[");\
   for(i=0; i<vsip_cvgetlength_d(_v); i++){ \
      vsip_cscalar_d a = vsip_cvget_d(_v,i); \
      printf("(%f, %f) \n", vsip_real_d(a),vsip_imag_d(a)); \
} printf("]\n");}

#define PRINT(_v) { /* real vector print macro */\
   int i; \
   printf("\n[");\
   for(i=0; i<vsip_vgetlength_d(_v); i++){ \
      vsip_scalar_d a = vsip_vget_d(_v,i); \
      printf("%f\n", a);\
} printf("]\n");}

int main(){
   int retval = vsip_init((void*)0);
   vsip_length N = NSIZE;

   /* create objects */
   vsip_cvview_d *cdata_in = vsip_cvcreate_d(N,VSIP_MEM_NONE);
   vsip_cvview_d *cdata_sorted =\
             vsip_cvcreate_d(N,VSIP_MEM_NONE);
   vsip_vview_d *re = vsip_vrealview_d(cdata_in);
```

```
    vsip_vview_d *im = vsip_vimagview_d(cdata_in);
    vsip_vview_d *rdata = vsip_vcreate_d(N,VSIP_MEM_NONE);
    vsip_vview_vi *ind = vsip_vcreate_vi(N,VSIP_MEM_NONE);
    vsip_randstate *state = \
              vsip_randcreate(SEED,PROCS,ID, VSIP_PRNG);

    /* make some complex data for cdata_in*/
    vsip_vrandn_d(state,re);
    vsip_vrandn_d(state,im);
    printf("\ninput vector");CPRINT(cdata_in);

    /* sort the complex data by magnitude */
    vsip_cvmag_d(cdata_in,rdata);
    vsip_vsortip_d(rdata,VSIP_SORT_BYVALUE, \
                   VSIP_SORT_ASCENDING,VSIP_TRUE,ind);
    vsip_cvgather_d(cdata_in,ind,cdata_sorted);
    printf("\noutput vector"); CPRINT(cdata_sorted);
    vsip_cvmag_d(cdata_sorted,rdata);
    printf("\nmagnitude of output data");PRINT(rdata);

    /* destroy objects */
    vsip_randdestroy(state);
    vsip_vdestroy_d(re);
    vsip_vdestroy_d(im);
    vsip_cvalldestroy_d(cdata_in);
    vsip_cvalldestroy_d(cdata_sorted);
    vsip_valldestroy_d(rdata);
    vsip_valldestroy_vi(ind);
    vsip_finalize((void*)0);
    return retval;
}
```

Output of example below.

```
input vector
[(-1.213093, -0.207605)
(1.507828, -0.331832)
(1.683972, -0.671796)
(0.243529, 0.384777)
(0.451852, -0.150182)
(0.973947, -0.927580)
(0.124482, -0.136170)
(0.305283, -1.190044)
(-0.369669, 0.771362)
(-1.261732, -0.429227)
]

output vector
[(0.124482, -0.136170)
(0.243529, 0.384777)
(0.451852, -0.150182)
(-0.369669, 0.771362)
```

```
(0.305283, -1.190044)
(-1.213093, -0.207605)
(-1.261732, -0.429227)
(0.973947, -0.927580)
(1.507828, -0.331832)
(1.683972, -0.671796)
]

magnitude of output data
[0.184494
0.455367
0.476156
0.855369
1.228577
1.230729
1.332743
1.344982
1.543910
1.813028
]
```

See Also