

**Adelsbach/VS IPL
Core Double Precision Profile
Programming Reference Guide
DD-00017-015**

Jan Adelsbach

October 28, 2026

Contents

0.1	About this Guide	9
0.1.1	Legal Information	9
0.1.2	Feedback and Contact	9
0.2	Overview	9
0.2.1	Introduction	9
0.2.2	Link Libraries	9
0.3	General Functions	10
0.3.1	vsip_cstorage_p - Complex storage type	11
1	Support Functions	13
1.1	Initialization Functions	14
1.1.1	vsip_init - Initialize	15
1.1.2	vsip_finalize - Finalize	16
1.2	Block Support Functions	17
1.2.1	vsip_dblockcreate_p - Create a block	18
1.2.2	vsip_blockbind_p - Create a block using existing data	20
1.2.3	vsip_cblockbind_p - Create a block using existing data (complex)	22
1.2.4	vsip_blockrebind_p - Rebind existing block	24
1.2.5	vsip_cblockrebind_p - Rebind existing block (complex)	26
1.2.6	vsip_dblockadmit_p - Admit block data	27
1.2.7	vsip_blockfind_p - Get user data	28
1.2.8	vsip_cblockfind_p - Get user data (complex)	29
1.2.9	vsip_blockrelease_p - Release a block	30
1.2.10	vsip_cblockrelease_p - Release a block (complex)	31
1.2.11	vsip_dblockdestroy_p - Destroy a block	32
1.3	Vector View Support Functions	33
1.3.1	vsip_dvcreate_p - Create a Vector View	34
1.3.2	vsip_dvbind_p - Bind a Vector View to a Data Block	35
1.3.3	vsip_dvcloneview_p - Clone a Vector View	36
1.3.4	vsip_dvget_p - Get an Element from a Vector View	37
1.3.5	vsip_dvput_p - Set an Element in a Vector View	38
1.3.6	vsip_dvsubview_p - Create a Subview of a Vector View	39
1.3.7	vsip_vrealview_p - Get the Real Part View of a Complex Vector View	40
1.3.8	vsip_vimagview_p - Get the Imaginary Part View of a Complex Vector View	41
1.3.9	vsip_dvgetattrib_p - Get the Attributes of a Vector View	42
1.3.10	vsip_dvputattrib_p - Set the Attributes of a Vector View	43
1.3.11	vsip_dvgetblock_p - Get the Data Block of a Vector View	44
1.3.12	vsip_dvgetlength_p - Get the Length of a Vector View	45
1.3.13	vsip_dvputlength_p - Set the Length of a Vector View	46
1.3.14	vsip_dvgetstride_p - Get the Stride of a Vector View	47
1.3.15	vsip_dvputstride_p - Set the Stride of a Vector View	48
1.3.16	vsip_dvgetoffset_p - Get the Offset of a Vector View	49
1.3.17	vsip_dvputoffset_p - Set the Offset of a Vector View	50
1.3.18	vsip_dvdestroy_p - Destroy a Vector View	51
1.3.19	vsip_dvalldestroy_p - Destroy a Vector View and Its Data Block	52
1.4	Matrix View Support Functions	53
1.4.1	vsip_dmcreate_p - Create a Matrix View	54

1.4.2	<code>vsip_dmbind_p</code> - Bind a Matrix View to a Block	56
1.4.3	<code>vsip_dmcloneview_p</code> - Clone a Matrix View	58
1.4.4	<code>vsip_dmget_p</code> - Get Matrix Element	59
1.4.5	<code>vsip_dmput_p</code> - Set Matrix Element	60
1.4.6	<code>vsip_dmsubview_p</code> - Create a Submatrix View	61
1.4.7	<code>vsip_dmtransview_p</code> - Create a Transposed Matrix View	63
1.4.8	<code>vsip_dmrowview_p</code> - Create a Row Vector View of a Matrix	64
1.4.9	<code>vsip_dmcovview_p</code> - Create a Column Vector View of a Matrix	66
1.4.10	<code>vsip_dmdiagview_p</code> - Create a Diagonal Vector View of a Matrix	68
1.4.11	<code>vsip_mrealview_p</code> - Create a Real Part Matrix View	70
1.4.12	<code>vsip_mimagview_p</code> - Create an Imaginary Part Matrix View	71
1.4.13	<code>vsip_dmgetattrib_p</code> - Get Matrix Attributes	72
1.4.14	<code>vsip_dmputattrib_p</code> - Set Matrix Attributes	74
1.4.15	<code>vsip_dmgetblock_p</code> - Get the Data Block from a Matrix View	76
1.4.16	<code>vsip_dmgetcollength_p</code> - Get Number of Columns in a Matrix View	77
1.4.17	<code>vsip_dmputcollength_p</code> - Set Number of Columns in a Matrix View	78
1.4.18	<code>vsip_dmgetrowlength_p</code> - Get Number of Rows in a Matrix View	79
1.4.19	<code>vsip_dmputrowlength_p</code> - Set Number of Rows in a Matrix View	80
1.4.20	<code>vsip_dmgetcolstride_p</code> - Get Column Stride of a Matrix View	81
1.4.21	<code>vsip_dmputcolstride_p</code> - Set Column Stride of a Matrix View	82
1.4.22	<code>vsip_dmgetrowstride_p</code> - Get Row Stride of a Matrix View	83
1.4.23	<code>vsip_dmputrowstride_p</code> - Set Row Stride of a Matrix View	84
1.4.24	<code>vsip_dmgetoffset_p</code> - Get Matrix View Offset	85
1.4.25	<code>vsip_dmputoffset_p</code> - Set Matrix View Offset	86
1.4.26	<code>vsip_dmdestroy_p</code> - Destroy a Matrix View	87
1.4.27	<code>vsip_dmalldestroy_p</code> - Destroy Matrix View and its Data Block	88

2 Scalar Functions

89

2.1	Complex Scalar Functions	90
2.1.1	<code>vsip_arg_p</code> - Compute Phase Angle of Complex Scalar	91
2.1.2	<code>vsip_cmag_p</code> - Compute Magnitude of Complex Scalar	92
2.1.3	<code>vsip_cmagsq_p</code> - Compute Magnitude Squared of Complex Scalar	93
2.1.4	<code>vsip_conj_p</code> - Compute Complex Conjugate	94
2.1.5	<code>vsip_polar_p</code> - Convert Cartesian to Polar Coordinates	95
2.1.6	<code>vsip_rect_p</code> - Convert Polar to Cartesian Coordinates	96
2.1.7	<code>vsip_real_p</code> - Complex Real part	97
2.1.8	<code>vsip_imag_p</code> - Complex Imaginary part	98
2.1.9	<code>vsip_cmplx_p</code> - Create complex number	99
2.1.10	<code>vsip_cadd_p</code> - Complex Scalar Addition	100
2.1.11	<code>vsip_csub_p</code> - Complex Scalar Subtraction	101
2.1.12	<code>vsip_cmul_p</code> - Complex Scalar Multiplication	102
2.1.13	<code>vsip_cjmul_p</code> - Complex Scalar Conjugate Multiplication	103
2.1.14	<code>vsip_cdiv_p</code> - Complex Scalar Division	104
2.1.15	<code>vsip_rcadd_p</code> - Real-Complex Scalar Addition	105
2.1.16	<code>vsip_rcsub_p</code> - Real-Complex Scalar Subtraction	106
2.1.17	<code>vsip_rcmul_p</code> - Real-Complex Scalar Multiplication	107
2.1.18	<code>vsip_crsb_p</code> - Complex-Real Scalar Subtraction	108
2.1.19	<code>vsip_crdiv_p</code> - Complex-Real Scalar Division	109
2.1.20	<code>vsip_cneg_p</code> - Complex Scalar Negation	110
2.1.21	<code>vsip_crecip_p</code> - Complex Scalar Reciprocal	111
2.1.22	<code>vsip_csqrt_p</code> - Complex Scalar Square Root	112
2.1.23	<code>vsip_cexp_p</code> - Complex Scalar Exponential	113
2.1.24	<code>vsip_CONJ_p</code> - Complex Scalar Conjugate and Store to Pointer	114
2.1.25	<code>vsip_CMPLX_p</code> - Create a Complex Scalar and Store in a Pointer	115
2.1.26	<code>vsip_CADD_p</code> - Complex Scalar Addition and Store to Pointer	116
2.1.27	<code>vsip_CSUB_p</code> - Complex Scalar Subtraction and Store to Pointer	117
2.1.28	<code>vsip_CMUL_p</code> - Complex Scalar Multiplication and Store to Pointer	118
2.1.29	<code>vsip_CJMUL_p</code> - Complex Scalar Conjugate Multiplication and Store to Pointer	119

2.1.30	<code>vsip_CDIV_p</code> - Complex Scalar Division and Store to Pointer	120
2.1.31	<code>vsip_RCADD_p</code> - Real-Complex Scalar Addition and Store to Pointer	121
2.1.32	<code>vsip_RCSUB_p</code> - Real-Complex Scalar Subtraction and Store to Pointer	122
2.1.33	<code>vsip_RCMUL_p</code> - Real-Complex Scalar Multiplication and Store to Pointer	123
2.1.34	<code>vsip_CRSUB_p</code> - Complex-Real Scalar Subtraction and Store to Pointer	124
2.1.35	<code>vsip_CRDIV_p</code> - Complex-Real Scalar Division and Store to Pointer	125
2.1.36	<code>vsip_CNEG_p</code> - Complex Scalar Negate and Store to Pointer	126
2.1.37	<code>vsip_CRECIP_p</code> - Complex Scalar Reciprocal and Store to Pointer	127
2.1.38	<code>vsip_CSQRT_p</code> - Complex Scalar Square Root and Store to Pointer	128
2.1.39	<code>vsip_CEXP_p</code> - Complex Scalar Exponential and Store to Pointer	129
2.2	Index Scalar Functions	130
2.2.1	<code>vsip_matindex</code> - Convert Row/Column Indices to Matrix Index	131
2.2.2	<code>vsip_mrowindex</code> - Extract Row Index from Matrix Index	132
2.2.3	<code>vsip_mcolindex</code> - Extract Column Index from Matrix Index	133
2.2.4	<code>vsip_MATINDEX</code> - Convert Row/Column Indices to Matrix Index and Store in a Pointer	134
3	Random Number Generation	135
3.1	Random Number Functions	136
3.1.1	<code>vsip_randcreate</code> - Create a Random Number Generator State	137
3.1.2	<code>vsip_randdestroy</code> - Destroy a Random Number Generator State	138
3.1.3	<code>vsip_dvrandu_p</code> - Generate Uniformly Distributed Random Numbers in a Vector View	139
3.1.4	<code>vsip_dvrandn_p</code> - Fill Vector with Normally Distributed Random Numbers	140
4	Vector and Elementwise Operations	141
4.1	Elementary Math Operations	142
4.1.1	<code>vsip_vsin_p</code> - Element-wise Sine of a Vector View	143
4.1.2	<code>vsip_vasin_p</code> - Element-wise Arcsine of a Vector View	144
4.1.3	<code>vsip_vcos_p</code> - Element-wise Cosine of a Vector View	145
4.1.4	<code>vsip_vacos_p</code> - Element-wise Arccosine of a Vector View	146
4.1.5	<code>vsip_vtan_p</code> - Element-wise Tangent of a Vector View	147
4.1.6	<code>vsip_vatan_p</code> - Element-wise Arctangent of a Vector View	148
4.1.7	<code>vsip_vatan2_p</code> - Element-wise Arctangent of Two Vector Views	149
4.1.8	<code>vsip_dvexp_p</code> - Element-wise Exponential of a Vector View	150
4.1.9	<code>vsip_vexp10_p</code> - Element-wise Base-10 Exponential of a Vector View	151
4.1.10	<code>vsip_vlog_p</code> - Element-wise Natural Logarithm of a Vector View	152
4.1.11	<code>vsip_vlog10_p</code> - Element-wise Base-10 Logarithm of a Vector View	153
4.1.12	<code>vsip_dvsqrt_p</code> - Element-wise Square Root of a Vector View	154
4.2	Unary Operations	155
4.2.1	<code>vsip_dvneg_p</code> - Negate Elements of a Vector View	156
4.2.2	<code>vsip_vsumval_p</code> - Compute the Sum of Elements in a Vector View	157
4.2.3	<code>vsip_vsumsqval_p</code> - Compute the Sum of Squares of Elements in a Vector View	158
4.2.4	<code>vsip_dvmeanval_p</code> - Compute Mean Value of Vector	159
4.2.5	<code>vsip_dvmeansqval_p</code> - Compute Mean of Squared Values	160
4.2.6	<code>vsip_vsq_p</code> - Square Elements of a Vector View	161
4.2.7	<code>vsip_vrecip_p</code> - Compute Reciprocal of Elements of a Vector View	162
4.2.8	<code>vsip_dvmag_p</code> - Compute Magnitude of Elements of a Vector View	163
4.2.9	<code>vsip_vcmagsq_p</code> - Element-wise Magnitude Squared of a Complex Vector View	164
4.2.10	<code>vsip_cvconj_p</code> - Element-wise Complex Conjugate of a Complex Vector View	165
4.2.11	<code>vsip_veuler_p</code> - Convert Real Vector to Complex Euler Representation	166
4.2.12	<code>vsip_dvmodulate_p</code> - Vector Modulation with Complex Carrier	167
4.2.13	<code>vsip_dvrsqrt_p</code> - Element-wise Reciprocal Square Root of a Vector View	168
4.3	Binary Operations	169
4.3.1	<code>vsip_dvadd_p</code> - Element-wise Addition of Two Vector Views	170
4.3.2	<code>vsip_dvsub_p</code> - Element-wise Subtraction of Two Vector Views	171
4.3.3	<code>vsip_dvmul_p</code> - Element-wise Multiplication of Two Vector Views	172
4.3.4	<code>vsip_dvdiv_p</code> - Element-wise Division of Two Vector Views	173
4.3.5	<code>vsip_cvjmul_p</code> - Element-wise Complex Conjugate Multiplication of Two Complex Vector Views	174
4.3.6	<code>vsip_rcvadd_p</code> - Element-wise Real-Complex Addition	175
4.3.7	<code>vsip_rcvsub_p</code> - Element-wise Real-Complex Subtraction	176

4.3.8	<code>vsip_rcvmul_p</code> - Element-wise Real-Complex Multiplication	177
4.3.9	<code>vsip_crvsub_p</code> - Element-wise Complex-Real Subtraction	178
4.3.10	<code>vsip_dsvadd_p</code> - Add a Scalar to a Vector View	179
4.3.11	<code>vsip_dsvsub_p</code> - Subtract a Scalar to a Vector View	180
4.3.12	<code>vsip_dsvmul_p</code> - Multiply a Scalar by a Vector View	181
4.3.13	<code>vsip_dsvdiv_p</code> - Divide a Scalar by a Vector View	182
4.3.14	<code>vsip_rscvadd_p</code> - Element-wise Real-Scalar-Complex Addition	183
4.3.15	<code>vsip_rscvsub_p</code> - Element-wise Real-Scalar-Complex Subtraction	184
4.3.16	<code>vsip_rscvmul_p</code> - Element-wise Real-Scalar-Complex Multiplication	185
4.3.17	<code>vsip_dvmmul_p</code> - Vector-Matrix Multiplication	186
4.3.18	<code>vsip_rvcmmul_p</code> - Real Vector-Complex Matrix Multiplication	187
4.3.19	<code>vsip_dvexpoavg_p</code> - Vector Exponential Average	188
4.3.20	<code>vsip_vhypot_p</code> - Vector Hypotenuse (Euclidean Norm)	189
4.4	Ternary Operations	190
4.4.1	<code>vsip_dvam_p</code> - Vector Add-Multiply	191
4.4.2	<code>vsip_dvma_p</code> - Vector Multiply-Add	192
4.4.3	<code>vsip_dvmsb_p</code> - Vector Multiply-Subtract	193
4.4.4	<code>vsip_dvmsa_p</code> - Vector Multiply-Scalar-Add	194
4.4.5	<code>vsip_dvsam_p</code> - Vector Vector-Add-Scalar-Multiply	195
4.4.6	<code>vsip_dvsbm_p</code> - Vector Subtract-Multiply	196
4.4.7	<code>vsip_dvsma_p</code> - Vector Vector-Scalar-Multiply-Add	197
4.4.8	<code>vsip_dvsmsa_p</code> - Vector Vector-Scalar-Multiply-Scalar-Add	198
4.5	Logical Operations	199
4.5.1	<code>vsip_valltrue_p</code> - Check if All Elements in Boolean Vector are True	200
4.5.2	<code>vsip_vanytrue_p</code> - Check if Any Element in Boolean Vector is True	201
4.5.3	<code>vsip_vleq_p</code> - Element-wise Equal Comparison	202
4.5.4	<code>vsip_vlne_p</code> - Element-wise Not-Equal Comparison	203
4.5.5	<code>vsip_vllt_p</code> - Element-wise Less-Than Comparison	204
4.5.6	<code>vsip_vlle_p</code> - Element-wise Lesser-Or-Equal-Than Comparison	205
4.5.7	<code>vsip_vlgt_p</code> - Element-wise Greater-Than Comparison	206
4.5.8	<code>vsip_vlge_p</code> - Element-wise Greater-Or-Equal-Than Comparison	207
4.6	Selection Operations	208
4.6.1	<code>vsip_vclip_p</code> - Clip Vector Elements Between Thresholds	209
4.6.2	<code>vsip_vinvclip_p</code> - Inverse Clip Vector Elements	210
4.6.3	<code>vsip_vindexbool</code> - Find Indices of True Elements in Boolean Vector	212
4.6.4	<code>vsip_vmaxval_p</code> - Find the Maximum Value in a Vector View	213
4.6.5	<code>vsip_vminval_p</code> - Find the Minimum Value in a Vector View	214
4.6.6	<code>vsip_vmax_p</code> - Element-wise Maximum of Two Vector Views	215
4.6.7	<code>vsip_vmin_p</code> - Element-wise Minimum of Two Vector Views	216
4.6.8	<code>vsip_vmaxmg_p</code> - Element-wise Maximum of Magnitudes	217
4.6.9	<code>vsip_vminmg_p</code> - Element-wise Minimum of Magnitudes	218
4.6.10	<code>vsip_vmaxmgval_p</code> - Find Maximum Magnitude Value in Vector	219
4.6.11	<code>vsip_vminmgval_p</code> - Find Minimum Magnitude Value in Vector	220
4.6.12	<code>vsip_vcmaxmg_p</code> - Element-wise Maximum of Complex Vector Magnitudes	221
4.6.13	<code>vsip_vcminmg_p</code> - Element-wise Minimum of Complex Vector Magnitudes	222
4.6.14	<code>vsip_vcmaxmgsqval_p</code> - Find Maximum Magnitude Squared Value in Complex Vector	223
4.6.15	<code>vsip_vcminmgsqval_p</code> - Find Minimum Magnitude Squared Value in Complex Vector	224
4.7	Bitwise and Boolean Logical Operations	225
4.7.1	<code>vsip_vnot_p</code> - Boolean Vector Logical NOT	226
4.7.2	<code>vsip_vand_p</code> - Boolean Vector Logical AND	227
4.7.3	<code>vsip_vor_p</code> - Boolean Vector Logical OR	228
4.7.4	<code>vsip_vxor_p</code> - Boolean Vector Logical XOR	229
4.8	Element Generation Functions	230
4.8.1	<code>vsip_dvfill_p</code> - Fill a Vector View with a Scalar Value	231
4.8.2	<code>vsip_vramp_p</code> - Fill a Vector View with a Ramp	232
4.9	Copying Functions	233
4.9.1	<code>vsip_dvcopy_p_p</code> - Copy Vector Views	234
4.9.2	<code>vsip_dmcopy_p</code> - Copy Matrix Views	236

4.10 Manipulation Operations	237
4.10.1 vsip_vreal_p - Extract Real Part of a Complex Vector View	238
4.10.2 vsip_vimag_p - Extract Imaginary Part of a Complex Vector View	239
4.10.3 vsip_vcplx_p - Create a Complex Vector View from Real and Imaginary Parts	240
4.10.4 vsip_dvgather_p - Gather Elements from a Vector	241
4.10.5 vsip_dvscatter_p - Scatter Elements to a Vector	243
4.10.6 vsip_dvswap_p - Swap Elements Between two Vectors	245
4.10.7 vsip_vrect_p - Convert Cartesian Coordinates to Complex Numbers	247
4.10.8 vsip_vpolar_p - Convert Polar Coordinates to Cartesian	249
5 Signal Processing Functions	251
5.1 FFT Functions	252
5.1.1 vsip_ddfftop_create_p - Create FFT Objects (Out-of-Place)	253
5.1.2 vsip_ccfftip_create_p - Create FFT Object (In-Place)	255
5.1.3 vsip_fft_destroy_p - Destroy an FFT Object	257
5.1.4 vsip_fft_getattr_p - Get FFT Object Attributes	258
5.1.5 vsip_ddfftop_p - Perform FFT Operations (Out-of-Place)	259
5.1.6 vsip_ccfftip_p - Perform FFT Operations (In-Place)	260
5.1.7 vsip_ddffmop_create_p - Create Multiple-FFT Objects (Out-of-Place)	261
5.1.8 vsip_ccffmip_create_p - Create Multiple-FFT Object (In-Place)	263
5.1.9 vsip_fftm_destroy_p - Destroy a Multiple-FFT Object	264
5.1.10 vsip_fftm_getattr_p - Get Multiple-FFT Object Attributes	265
5.1.11 vsip_ddffmop_p - Perform Multiple-FFT Operations (Out-of-Place)	266
5.1.12 vsip_ccffmip_p - Perform Multiple-FFT Operations (In-Place)	267
5.2 Convolution and Correlation Functions	268
5.2.1 vsip_dconv1d_create_p - Create 1D Convolution Object	269
5.2.2 vsip_dconv1d_destroy_p - Destroy 1D Convolution Object	271
5.2.3 vsip_dconv1d_getattr_p - Get 1D Convolution Object Attributes	272
5.2.4 vsip_dconvolve1d_p - Perform 1D Convolution	273
5.2.5 vsip_dcorr1d_create_p - Create 1D Correlation Object	274
5.2.6 vsip_dcorr1d_destroy_p - Destroy 1D Correlation Object	276
5.2.7 vsip_dcorr1d_getattr_p - Get 1D Correlation Object Attributes	277
5.2.8 vsip_dcorrelate1d_p - Compute 1D Correlation	278
5.3 Window Functions	280
5.3.1 vsip_vcreate_blackman_p - Create a Blackman Window Vector	281
5.3.2 vsip_vcreate_kaiser_p - Create a Kaiser Window Vector	283
5.3.3 vsip_vcreate_cheby_p - Create a Chebyshev Window Vector	285
5.3.4 vsip_vcreate_hanning_p - Create a Hanning Window Vector	287
5.4 FIR	289
5.4.1 vsip_dfir_create_p - Create a FIR Filter	290
5.4.2 vsip_dfir_reset_p - Reset a FIR Filter	292
5.4.3 vsip_dfir_getattr_p - Get Attributes of a FIR Filter	293
5.4.4 vsip_dfirflt_p - Apply a FIR Filter to a Vector View	294
5.4.5 vsip_dfir_destroy_p - Destroy a FIR Filter	295
5.5 Miscellaneous Signal Processing Functions	296
5.5.1 vsip_vhisto_p - Compute Histogram of a Vector View	297
6 Linear Algebra Functions	299
6.1 Matrix and Vector Operations	300
6.1.1 vsip_dvdot_p - Compute the Dot Product of Two Vector Views	301
6.1.2 vsip_cvjdot_p - Compute the Conjugate Dot Product of Two Complex Vector Views	302
6.1.3 vsip_dvouter_p - Outer Product of Two Vectors	303
6.1.4 vsip_dmtrans_p - Matrix Transposition	305
6.1.5 vsip_cmherm_p - Matrix Hermitian	307
6.1.6 vsip_dgemp_p - General Matrix Product	308
6.1.7 vsip_dgems_p - General Matrix Scaling and Addition	310
6.1.8 vsip_dvmprod_p - Vector-Matrix Product	312
6.1.9 vsip_dmvprod_p - Matrix-Vector Product	314
6.1.10 vsip_dmprod_p - Matrix-Matrix Product	316

6.1.11	<code>vsip_dmprodt_p</code> - Matrix-Matrix Product with Transposition	318
6.1.12	<code>vsip_cmprodh_p</code> - Complex Matrix Product with Hermitian Transpose	320
6.1.13	<code>vsip_cmprodj_p</code> - Complex Matrix Product with Conjugate	322
6.2	Special Linear Solvers	324
6.2.1	<code>vsip_dtoepsol_p</code> - Solve a Toeplitz System of Equations	325
6.2.2	<code>vsip_dcovsol_p</code> - Solve a Covariance System of Equations	326
6.2.3	<code>vsip_dllsqsol_p</code> - Solve Linear Least Squares Problem	327
6.3	General Linear Square System Solver	328
6.3.1	<code>vsip_dlud_create_p</code> - Create LU Decomposition Object	329
6.3.2	<code>vsip_dlud_destroy_p</code> - Destroy LU Decomposition Object	330
6.3.3	<code>vsip_dlud_getattr_p</code> - Get LU Decomposition Attributes	331
6.3.4	<code>vsip_dlud_p</code> - Perform LU Decomposition	332
6.3.5	<code>vsip_dlusol_p</code> - Solve Linear System Using LU Decomposition	333
6.4	Symmetric Positive Definite Linear System Solver	334
6.4.1	<code>vsip_dchold_create_p</code> - Create Cholesky Decomposition Object	335
6.4.2	<code>vsip_dchold_destroy_p</code> - Destroy Cholesky Decomposition Object	336
6.4.3	<code>vsip_dchold_getattr_p</code> - Get Cholesky Decomposition Attributes	337
6.4.4	<code>vsip_dchold_p</code> - Perform Cholesky Decomposition	338
6.4.5	<code>vsip_dcholsol_p</code> - Solve Linear Systems Using Cholesky Decomposition	339
6.5	Over-determined Linear System Solver	340
6.5.1	<code>vsip_dqrd_create_p</code> - Create QR Decomposition Object	341
6.5.2	<code>vsip_dqrd_destroy_p</code> - Destroy QR Decomposition Object	343
6.5.3	<code>vsip_dqrd_getattr_p</code> - Get QR Decomposition Attributes	344
6.5.4	<code>vsip_dqrd_p</code> - Perform QR Decomposition	345
6.5.5	<code>vsip_dqrsol_p</code> - Solve Linear Systems Using QR Decomposition	346
6.5.6	<code>vsip_dqrdsolr_p</code> - Solve Linear Systems with Modified R Matrix	347
6.5.7	<code>vsip_dqrdprodq_p</code> - Multiply by Q Matrix from QR Decomposition	348

0.1 About this Guide

0.1.1 Legal Information

Copyright ©2025-2026 Adelsbach UG (haftungsbeschränkt). All Rights Reserved.

Copyright ©2025-2026 Jan Adelsbach. All Rights Reserved.

From herein referred to as *Adelsbach*.

This document may not be reproduced without written permission by Adelsbach.

0.1.2 Feedback and Contact

For feedback on this document, please use the following email address:

techpubs@adelsbach-research.eu

Please include the page number or a link to the page.

For general contact details, please visit <https://adelsbach-research.eu/contact>.

0.2 Overview

0.2.1 Introduction

The *Adelsbach / VSIPL* is an implementation of the digital signal processing API standard of the Object Management Group version VSIPL 1.5.

This reference manual provides a brief reference of all functionality provided in the *Adelsbach / VSIPL* library for the *Core* profile. For a more thorough and complete reference, please refer to the Object Management Group VSIPL 1.5 standard.

0.2.2 Link Libraries

The following libraries are provided with the distribution. For development it is recommended to link against the more extensive error checking library, whereas for deployment use one of the performance tuned variants.

- `libavsipl_c.a` Performance tuned library without additional error checking. May make use of processor SIMD features, please see platform details.
- `libavsipl_c_mp.a` Performance tuned library with shared memory multithreading (OpenMP)
- `libavsipl_c_dbg.a` Non-performance tuned library with extensive error checking.

0.3 General Functions

0.3.1 vsip_cstorage_p - Complex storage type

```
typedef enum _vsip_cmplx_mem {
    VSIP_CMPLX_INTERLEAVED,
    VSIP_CMPLX_SPLIT,
    VSIP_CMPLX_NONE
} vsip_cmplx_mem;

vsip_cmplx_mem vsip_cstorage_f(void);
vsip_cmplx_mem vsip_cstorage_d(void);

/* deprecated */
vsip_cmplx_mem vsip_cstorage(void);
```

Description

These functions query the manner in which complex values are stored. This can be in interleaved (real followed by imaginary part in one vector) or split format (two separate vectors).

Return Value

- Returns one of the enumerator values.

Example

```
vsip_cmplx_mem complex_storage;

// Allocate complex storage using the preferred method
complex_storage = vsip_cstorage_f();
```


Chapter 1

Support Functions

1.1 Initialization Functions

1.1.1 vsip_init - Initialize

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

Description

This function initializes the VSIPL library and must be called before any other VSIPL functions are used. It can be called multiple times without side effects.

Parameters

- void*: The argument is unused and should be set to 0 or NULL.

Return Value

- Returns 0 on success.
- Returns a non-zero value on error.

Example

```
int result;

// Initialize the VSIPL library
result = vsip_init(NULL);

if (result != 0) {
    // Handle error
}
```

1.1.2 vsip_finalize - Finalize

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

Description

This function finalizes the VSIPL library, releasing all internal resources and memory. After calling this function, no other VSIPL functions may be called. The function can be called in a nested manner, but only the outermost call will actually free up the internal initialization memory.

Parameters

- void*: The argument is unused and should be set to 0 or NULL.

Return Value

- Returns 0 on success.
- Returns a non-zero value on error.

Example

```
int result;

// Finalize the VSIPL library
result = vsip_finalize(NULL);

if (result != 0) {
    // Handle error
}
```


1.2 Block Support Functions

1.2.1 vsip_dblockcreate_p - Create a block

```
typedef enum _vsip_memory_hint {
    VSIP_MEM_NONE          = 0,
    VSIP_MEM_RDONLY        = 1,
    VSIP_MEM_CONST         = 2,
    VSIP_MEM_SHARED        = 3,
    VSIP_MEM_SHARED_RDONLY = 4,
    VSIP_MEM_SHARED_CONST  = 5
} vsip_memory_hint;

vsip_block_f* vsip_blockcreate_f(vsip_length n, vsip_memory_hint h);
vsip_block_d* vsip_blockcreate_d(vsip_length n, vsip_memory_hint h);
vsip_block_i* vsip_blockcreate_i(vsip_length n, vsip_memory_hint h);
vsip_block_bl* vsip_blockcreate_bl(vsip_length n, vsip_memory_hint h);
vsip_block_vi* vsip_blockcreate_vi(vsip_length n, vsip_memory_hint h);
vsip_block_mi* vsip_blockcreate_mi(vsip_length n, vsip_memory_hint h);
vsip_cblock_f* vsip_cblockcreate_f(vsip_length n, vsip_memory_hint h);
vsip_cblock_d* vsip_cblockcreate_d(vsip_length n, vsip_memory_hint h);
```

Description

These functions create a block of data of the specified type with $n > 0$ elements. The memory hint h describes how this data is intended to be used, such as read-only, constant, or shared memory.

Parameters

- `vsip_length n`: The number of elements in the block. Must be greater than 0.
- `vsip_memory_hint h`: Memory hint for the block, indicating properties such as read-only, constant, or shared memory.
 - `VSIP_MEM_NONE` - No memory hint
 - `VSIP_MEM_RDONLY` - The memory is to be used read-only
 - `VSIP_MEM_CONST` - The memory will hold constants
 - `VSIP_MEM_SHARED` - The memory will be shared
 - `VSIP_MEM_SHARED_RDONLY` - The memory will be shared and is read-only
 - `VSIP_MEM_SHARED_CONST` - The memory will be shared and will hold constants

Return Value

- On success, a pointer to the newly created block object is returned.
- On error, `NULL` is returned.

Error Handling

If an error occurs, the function returns `NULL`.

Example

```
vsip_length length = 10;
vsip_memory_hint hint = VSIP_MEM_NONE;
vsip_block_f *float_block;

// Create a float block
float_block = vsip_blockcreate_f(length, hint);

if (float_block == NULL) {
    // Handle error
```

```
}

vsip_block_i *int_block;

// Create an integer block
int_block = vsip_blockcreate_i(length, hint);

if (int_block == NULL) {
    // Handle error
}

vsip_cblock_f *complex_block;

// Create a complex float block
complex_block = vsip_cblockcreate_f(length, hint);

if (complex_block == NULL) {
    // Handle error
}
```

1.2.2 vsip_blockbind_p - Create a block using existing data

```
typedef enum _vsip_memory_hint {
    VSIP_MEM_NONE          = 0,
    VSIP_MEM_RDONLY        = 1,
    VSIP_MEM_CONST         = 2,
    VSIP_MEM_SHARED        = 3,
    VSIP_MEM_SHARED_RDONLY = 4,
    VSIP_MEM_SHARED_CONST  = 5
} vsip_memory_hint;

vsip_block_f* vsip_blockbind_f(vsip_scalar_f *p, vsip_length n, vsip_memory_hint h);
vsip_block_d* vsip_blockbind_d(vsip_scalar_d *p, vsip_length n, vsip_memory_hint h);
vsip_block_i* vsip_blockbind_i(vsip_scalar_i *p, vsip_length n, vsip_memory_hint h);
vsip_block_bl* vsip_blockbind_bl(vsip_scalar_bl *p, vsip_length n, vsip_memory_hint h);
vsip_block_vi* vsip_blockbind_vi(vsip_scalar_vi *p, vsip_length n, vsip_memory_hint h);
vsip_block_mi* vsip_blockbind_mi(vsip_scalar_mi *p, vsip_length n, vsip_memory_hint h);
```

Description

These functions create a new data block using an existing data array *p* with $n > 0$ elements and a given memory hint *h*. The block must be admitted before it can be used.

Parameters

- *vsip_scalar_p* *p: Pointer to the existing data array.
- *vsip_length* n: The number of elements in the data array. Must be greater than 0.
- *vsip_memory_hint* h: Memory hint for the block, indicating properties such as read-only, constant, or shared memory.
 - VSIP_MEM_NONE - No memory hint
 - VSIP_MEM_RDONLY - The memory is to be used read-only
 - VSIP_MEM_CONST - The memory will hold constants
 - VSIP_MEM_SHARED - The memory will be shared
 - VSIP_MEM_SHARED_RDONLY - The memory will be shared and is read-only
 - VSIP_MEM_SHARED_CONST - The memory will be shared and will hold constants

Return Value

- On success, a pointer to the newly created block object is returned.
- On error, NULL is returned.

Error Handling

If an error occurs, the function returns NULL.

Example

```
vsip_scalar_f float_data[10]; // Example float data array
vsip_length length = 10;
vsip_memory_hint hint = VSIP_MEM_NONE;
vsip_block_f *float_block;

// Create a float block
float_block = vsip_blockbind_f(float_data, length, hint);

if (float_block == NULL) {
    // Handle error
```

```
}  
  
// Admit the block before using it  
int result = vsip_blockadmit_f(float_block, VSIP_TRUE);  
if (result != 0) {  
    // Handle error  
}
```

1.2.3 vsip_cblockbind_p - Create a block using existing data (complex)

```
typedef enum _vsip_memory_hint {
    VSIP_MEM_NONE          = 0,
    VSIP_MEM_RDONLY        = 1,
    VSIP_MEM_CONST         = 2,
    VSIP_MEM_SHARED        = 3,
    VSIP_MEM_SHARED_RDONLY = 4,
    VSIP_MEM_SHARED_CONST  = 5
} vsip_memory_hint;
```

```
vsip_cblock_f* vsip_cblockbind_f(vsip_scalar_f *r, vsip_scalar_f *i, vsip_length n, vsip_memory_hint h);
vsip_cblock_d* vsip_cblockbind_d(vsip_scalar_d *r, vsip_scalar_d *i, vsip_length n, vsip_memory_hint h);
```

Description

This function creates a new complex data block using existing data, which can be either interleaved complex numbers or split real and imaginary data arrays. If the imaginary data array *i* is NULL, it is assumed that *r* is interleaved and contains $2n > 0$ elements. If the imaginary data array is provided, it is assumed that each of the *r* and *i* arrays contains $n > 0$ elements.

The block must be admitted before it can be used.

Parameters

- *vsip_scalar_p* *r: Pointer to the real part array or the interleaved array.
- *vsip_scalar_p* *i: Pointer to the imaginary part array. If NULL, *r* is assumed to be interleaved.
- *vsip_length* n: The number of complex elements. Must be greater than 0.
- *vsip_memory_hint* h: Memory hint for the block, indicating properties such as read-only, constant, or shared memory.
 - VSIP_MEM_NONE - No memory hint
 - VSIP_MEM_RDONLY - The memory is to be used read-only
 - VSIP_MEM_CONST - The memory will hold constants
 - VSIP_MEM_SHARED - The memory will be shared
 - VSIP_MEM_SHARED_RDONLY - The memory will be shared and is read-only
 - VSIP_MEM_SHARED_CONST - The memory will be shared and will hold constants

Return Value

- On success, a pointer to the newly created complex block object is returned.
- On error, NULL is returned.

Error Handling

If an error occurs, the function returns NULL.

Example

```
vsip_scalar_f real_data[10]; // Example data array
vsip_scalar_f imag_data[10]; // Example imaginary data array
vsip_length length = 10;
vsip_memory_hint hint = VSIP_MEM_NONE;
vsip_cblock_f *block;

// Create a block with split real and imaginary data
block = vsip_cblockbind_f(real_data, imag_data, length, hint);
```

```
if (block == NULL) {
    // Handle error
}

// Admit the block before using it
int result = vsip_cblockadmit_f(block, VSIP_TRUE);
if (result != 0) {
    // Handle error
}

// Create a block with interleaved data
vsip_scalar_f interleaved_data[20]; // Example interleaved data array
block = vsip_cblockbind_f(interleaved_data, NULL, length, hint);

if (block == NULL) {
    // Handle error
}

// Admit the block before using it
result = vsip_cblockadmit_f(block, VSIP_TRUE);
if (result != 0) {
    // Handle error
}
```

1.2.4 vsip_blockrebind_p - Rebind existing block

```
vsip_scalar_f* vsip_blockrebind_f(vsip_block_f *p, vsip_scalar_f *d);
vsip_scalar_d* vsip_blockrebind_d(vsip_block_d *p, vsip_scalar_d *d);
vsip_scalar_i* vsip_blockrebind_i(vsip_block_i *p, vsip_scalar_i *d);
vsip_scalar_bl* vsip_blockrebind_bl(vsip_block_bl *p, vsip_scalar_bl *d);
vsip_scalar_vi* vsip_blockrebind_vi(vsip_block_vi *p, vsip_scalar_vi *d);
vsip_scalar_mi* vsip_blockrebind_mi(vsip_block_mi *p, vsip_scalar_mi *d);
```

Description

These functions rebind an existing block *p* to a new user data array *d*. The new data array must have the same size as the originally bound data array.

The block must be in the released state to be rebound. After rebinding, the block must be admitted before it can be used.

A pointer to the previously bound user data array is returned. If an error occurs, NULL is returned.

Parameters

- *vsip_block_p *p*: Pointer to the block to be rebound.
- *vsip_scalar_p *d*: Pointer to the new user data array.

Return Value

- On success, a pointer to the previously bound user data array is returned.
- On error, NULL is returned.

Error Handling

If an error occurs, the function returns NULL.

Example

```
vsip_block_f *float_block;
vsip_scalar_f *new_data;
vsip_scalar_f *old_data;

// Assuming float_block has been properly initialized and is in the released state
old_data = vsip_blockrebind_f(float_block, new_data);

if (old_data == NULL) {
    // Handle error
}

// Admit the block before using it
int result = vsip_blockadmit_f(float_block, VSIP_TRUE);
if (result != 0) {
    // Handle error
}

vsip_block_i *int_block;
vsip_scalar_i *new_int_data;
vsip_scalar_i *old_int_data;

// Assuming int_block has been properly initialized and is in the released state
old_int_data = vsip_blockrebind_i(int_block, new_int_data);

if (old_int_data == NULL) {
    // Handle error
}
```



```
// Admit the block before using it
result = vsip_blockadmit_i(int_block, VSIP_TRUE);
if (result != 0) {
    // Handle error
}
```

1.2.5 vsip_cblockrebind_p - Rebind existing block (complex)

```
void vsip_cblockrebind_f(vsip_cblock_f *p, vsip_scalar_f *r, vsip_scalar_f *i, vsip_scalar_f **rr, vsip_sc  
void vsip_cblockrebind_d(vsip_cblock_d *p, vsip_scalar_d *r, vsip_scalar_d *i, vsip_scalar_d **rr, vsip_sc
```

Description

This function rebinds an existing complex block `p` to new real and imaginary part arrays `r` and `i`. If `i` is `NULL`, it is implied that `r` points to an interleaved array. The new array(s) must have the same size as the originally bound data array(s).

The block must be in the released state to be rebound. After rebinding, the block must be admitted before it can be used.

The previously bound data array(s) are stored in `rr` and `ir`. If the originally bound data is interleaved, `ir` will be set to `NULL`. On error, both `rr` and `ir` will be set to `NULL`.

Parameters

- `vsip_cblock_p *p`: Pointer to the complex block to be rebound.
- `vsip_scalar_p *r`: Pointer to the new real part array.
- `vsip_scalar_p *i`: Pointer to the new imaginary part array. If NULL, `r` is assumed to point to an interleaved array.
- `vsip_scalar_p **rr`: Pointer to store the previously bound real part array.
- `vsip_scalar_p **ir`: Pointer to store the previously bound imaginary part array. Will be NULL if the original data is interleaved.

Error Handling

On error, both `rr` and `ir` are set to `NULL`.

Example

```

vsip_cblock_f *block;
vsip_scalar_f *new_real_part;
vsip_scalar_f *new_imag_part = NULL; // For interleaved data
vsip_scalar_f *old_real_part;
vsip_scalar_f *old_imag_part;

// Assuming block has been properly initialized and is in the released state
vsip_cblockrebind_f(block, new_real_part, new_imag_part, &old_real_part, &old_imag_part);

if (old_real_part == NULL) {
    // Handle error
}

```

1.2.6 vsip_dblockadmit_p - Admit block data

```

int vsip_blockadmit_f(vsip_block_f *b, vsip_scalar_bl s);
int vsip_blockadmit_d(vsip_block_d *b, vsip_scalar_bl s);
int vsip_blockadmit_i(vsip_block_i *b, vsip_scalar_bl s);
int vsip_blockadmit_bl(vsip_block_bl *b, vsip_scalar_bl s);
int vsip_blockadmit_vi(vsip_block_vi *b, vsip_scalar_bl s);
int vsip_blockadmit_mi(vsip_block_mi *b, vsip_scalar_bl s);
int vsip_cblockadmit_f(vsip_cblock_f*, vsip_scalar_bl s);
int vsip_cblockadmit_d(vsip_cblock_d*, vsip_scalar_bl s);

```

Description

These functions admit user data to the given block `b`. After calling this function, the user array may no longer be manually manipulated outside of VSIPL routines. The boolean flag `s` indicates whether the user data should be consistent with the block data. In most cases, `s` should be set to `VSIP_TRUE`.

Parameters

- `vsip_dblock_p *b`: Pointer to the block to which user data is to be admitted.
- `vsip_scalar_bl s`: Boolean flag indicating whether the user data should be consistent with the block data.

Return Value

- Returns 0 on success.
- Returns a non-zero value on error.

Error Handling

If an error occurs, the function returns a non-zero value.

Example

```

vsip_block_f *float_block;
vsip_scalar_bl consistent = VSIP_TRUE;
int result;

// Assuming float_block has been properly initialized
result = vsip_blockadmit_f(float_block, consistent);

if (result != 0) {
    // Handle error
}

```

1.2.7 vsip_blockfind_p - Get user data

```
vsip_scalar_f* vsip_blockfind_f(const vsip_block_f *p);  
vsip_scalar_d* vsip_blockfind_d(const vsip_block_d *p);  
vsip_scalar_i* vsip_blockfind_i(const vsip_block_i *p);  
vsip_scalar_bl* vsip_blockfind_bl(const vsip_block_bl *p);  
vsip_scalar_vi* vsip_blockfind_vi(const vsip_block_vi *p);  
vsip_scalar_mi* vsip_blockfind_mi(const vsip_block_mi *p);
```

Description

These functions return a pointer to the user data array bound to the given block p. The block must have been bound previously and must be in the released state before calling these functions.

Parameters

- const vsip_block_p *p: Pointer to the block whose user data array is to be queried.

Return Value

- On success, a pointer to the user data array is returned.
- On error, NULL is returned.

Error Handling

If an error occurs, the function returns NULL.

Example

```
vsip_block_f *float_block;  
vsip_scalar_f *float_data;  
  
// Assuming float_block has been properly initialized, bound, and released  
float_data = vsip_blockfind_f(float_block);  
  
if (float_data == NULL) {  
    // Handle error  
}
```

1.2.8 vsip_cblockfind_p - Get user data (complex)

```
void vsip_cblockfind_f(const vsip_cblock_f *p, vsip_scalar_f **rr, vsip_scalar_f **ri);
void vsip_cblockfind_d(const vsip_cblock_d *p, vsip_scalar_d **rr, vsip_scalar_d **ri);
```

Description

This function queries the user data array(s) of the given complex block p. Depending on the data format, the function sets the pointers rr and ri accordingly:

- If the data is in interleaved format, only rr will be set, and ri will be set to NULL.
- If the data is in split format, both rr and ri will be set to point to the real and imaginary parts, respectively.

The block must have been bound previously and must be in the released state before calling this function.

Parameters

- `const vsip_cblock_p *p`: Pointer to the complex block whose user data arrays are to be queried.
- `vsip_scalar_p **rr`: Pointer to the real part of the user array.
- `vsip_scalar_p **ri`: Pointer to the imaginary part of the user array.

Error Handling

On error, both rr and ri are set to NULL.

Example

```
vsip_cblock_f *block;
vsip_scalar_f *real_part;
vsip_scalar_f *imag_part;

// Assuming block has been properly initialized, bound, and released
vsip_cblockfind_f(block, &real_part, &imag_part);

if (real_part == NULL) {
    // Handle error
}
```

1.2.9 vsip_blockrelease_p - Release a block

```
vsip_scalar_f* vsip_blockrelease_f(vsip_block_f *b, vsip_scalar_bl u);
vsip_scalar_d* vsip_blockrelease_d(vsip_block_d *b, vsip_scalar_bl u);
vsip_scalar_i* vsip_blockrelease_i(vsip_block_i *b, vsip_scalar_bl u);
vsip_scalar_bl* vsip_blockrelease_bl(vsip_block_bl *b, vsip_scalar_bl u);
vsip_scalar_vi* vsip_blockrelease_vi(vsip_block_vi *b, vsip_scalar_bl u);
vsip_scalar_mi* vsip_blockrelease_mi(vsip_block_mi *b, vsip_scalar_bl u);
```

Description

These functions release the user arrays in a block `b` and return a pointer to the user data array. The flag `u` determines whether the data must be maintained during the state change. The block must have been bound previously.

Parameters

- `vsip_block_p *`: Pointer to the block to be released.
- `vsip_scalar_bl u`: Flag indicating whether the data should be maintained.

Return Value

- On success, a pointer to the user data array is returned.
- On error, `NULL` is returned.

Error Handling

If an error occurs, the function returns `NULL`.

Example

```
vsip_block_f *float_block;
vsip_scalar_bl maintain_data = VSIP_TRUE;
vsip_scalar_f *float_data;

// Assuming float_block has been properly initialized and bound
float_data = vsip_blockrelease_f(float_block, maintain_data);

if (float_data == NULL) {
    // Handle error
}
```

1.2.10 vsip_cblockrelease_p - Release a block (complex)

```
void vsip_cblockrelease_f(vsip_cblock_f *b, vsip_scalar_bl u, vsip_scalar_f **rr, vsip_scalar_f **ri);
void vsip_cblockrelease_d(vsip_cblock_d *b, vsip_scalar_bl u, vsip_scalar_d **rr, vsip_scalar_d **ri);
```

Description

This function releases the complex block `b` and queries the user array(s). Depending on the data format, the function sets the pointers `rr` and `ri` accordingly:

- If the data is in interleaved format, only `rr` will be set, and `ri` will be set to `NULL`.
- If the data is in split format, both `rr` and `ri` will be set to point to the real and imaginary parts, respectively.

The flag `u` determines whether the data must be maintained during the state change. The block must have been bound previously and must be in the released state before calling this function.

Parameters

- `vsip_cblock_p *b`: Pointer to the complex block to be released.
- `vsip_scalar_bl u`: Flag indicating whether the data should be maintained.
- `vsip_scalar_p **rr`: Pointer to the real part of the user array.
- `vsip_scalar_p **ri`: Pointer to the imaginary part of the user array.

Error Handling

On error, both `rr` and `ri` are set to `NULL`.

Example

```
vsip_cblock_f *block;
vsip_scalar_bl maintain_data = VSIP_TRUE;
vsip_scalar_f *real_part;
vsip_scalar_f *imag_part;

// Assuming block has been properly initialized and bound
vsip_cblockrelease_f(block, maintain_data, &real_part, &imag_part);

if (real_part == NULL) {
    // Handle error
}
```

1.2.11 vsip_dblockdestroy_p - Destroy a block

```
void vsip_blockdestroy_f(vsip_block_f *b);
void vsip_blockdestroy_d(vsip_block_d *b);
void vsip_blockdestroy_i(vsip_block_i *b);
void vsip_blockdestroy_bl(vsip_block_bl *b);
void vsip_blockdestroy_vi(vsip_block_vi *b);
void vsip_blockdestroy_mi(vsip_block_mi *b);
void vsip_cblockdestroy_f(vsip_cblock_f *b);
void vsip_cblockdestroy_d(vsip_cblock_d *b);
```

Description

These functions destroy the block specified by the pointer `b`. Destroying a block involves deallocating the memory associated with it and performing any necessary cleanup operations. After calling one of these functions, the block pointer `b` becomes invalid and should not be used further.

Parameters

- `vsip_dblock_p *b` Pointer to a floating-point or integer block to be destroyed.

Return Value

These functions do not return a value.

Example

```
vsip_block_f *block = vsip_blockcreate_f(10, VSIP_MEM_NONE);
// Use the block...
vsip_blockdestroy_f(block); // Destroy the block when done
```

Notes

Ensure that the block pointer is valid and has been properly initialized before calling these functions. Attempting to destroy an already destroyed block or an invalid pointer may result in undefined behavior.

1.3 Vector View Support Functions

1.3.1 vsip_dvcreate_p - Create a Vector View

```
typedef enum _vsip_memory_hint {
    VSIP_MEM_NONE          = 0,
    VSIP_MEM_RDONLY        = 1,
    VSIP_MEM_CONST          = 2,
    VSIP_MEM_SHARED         = 3,
    VSIP_MEM_SHARED_RDONLY  = 4,
    VSIP_MEM_SHARED_CONST   = 5
} vsip_memory_hint;

vsip_vview_f* vsip_vcreate_f(vsip_length n, vsip_memory_hint h);
vsip_vview_d* vsip_vcreate_d(vsip_length n, vsip_memory_hint h);
vsip_vview_bl* vsip_vcreate_bl(vsip_length n, vsip_memory_hint h);
vsip_vview_vi* vsip_vcreate_vi(vsip_length n, vsip_memory_hint h);
vsip_vview_mi* vsip_vcreate_mi(vsip_length n, vsip_memory_hint h);
vsip_cvview_f* vsip_cvcreate_f(vsip_length n, vsip_memory_hint h);
vsip_cvview_d* vsip_cvcreate_d(vsip_length n, vsip_memory_hint h);
```

Description

This function creates a vector view of the specified length `n` with a given memory hint `h`. The memory hint describes how the data is intended to be used, such as read-only, constant, or shared memory.

Parameters

- `vsip_length n`: The number of elements in the vector view. Must be greater than 0.
- `vsip_memory_hint h`: Memory hint for the vector view, indicating properties such as read-only, constant, or shared memory.
 - `VSIP_MEM_NONE` - No memory hint
 - `VSIP_MEM_RDONLY` - The memory is to be used read-only
 - `VSIP_MEM_CONST` - The memory will hold constants
 - `VSIP_MEM_SHARED` - The memory will be shared
 - `VSIP_MEM_SHARED_RDONLY` - The memory will be shared and is read-only
 - `VSIP_MEM_SHARED_CONST` - The memory will be shared and will hold constants

Return Value

- On success, a pointer to the newly created vector view object is returned.
- On error, `NULL` is returned.

Error Handling

If an error occurs, the function returns `NULL`.

Example

```
vsip_length length = 10;
vsip_memory_hint hint = VSIP_MEM_NONE;
vsip_vview_f *vector_view;

// Create a vector view
vector_view = vsip_vcreate_f(length, hint);

if (vector_view == NULL) {
    // Handle error
}
```

1.3.2 vsip_dvbind_p - Bind a Vector View to a Data Block

```
vsip_vview_f* vsip_vbind_f(const vsip_block_f* b, vsip_offset o, vsip_stride s, vsip_length n);
vsip_vview_d* vsip_vbind_d(const vsip_block_d* b, vsip_offset o, vsip_stride s, vsip_length n);
vsip_vview_i* vsip_vbind_i(const vsip_block_i* b, vsip_offset o, vsip_stride s, vsip_length n);
vsip_vview_bl* vsip_vbind_bl(const vsip_block_bl* b, vsip_offset o, vsip_stride s, vsip_length n);
vsip_vview_vi* vsip_vbind_vi(const vsip_block_vi* b, vsip_offset o, vsip_stride s, vsip_length n);
vsip_vview_mi* vsip_vbind_mi(const vsip_block_mi* b, vsip_offset o, vsip_stride s, vsip_length n);
vsip_cvview_f* vsip_cvbind_f(const vsip_cblock_f* b, vsip_offset o, vsip_stride s, vsip_length n);
vsip_cvview_d* vsip_cvbind_d(const vsip_cblock_d* b, vsip_offset o, vsip_stride s, vsip_length n);
```

Description

This function binds a vector view to an existing data block `b` with a specified offset `o`, stride `s`, and length `n`. The vector view provides a view into the data block starting from the offset and stepping by the stride for the specified length.

Parameters

- `const vsip_dblock_p* b`: Pointer to the data block to which the vector view will be bound.
- `vsip_offset o`: Offset within the data block where the vector view starts.
- `vsip_stride s`: Stride between elements in the vector view.
- `vsip_length n`: The number of elements in the vector view.

Return Value

- On success, a pointer to the newly created vector view object is returned.
- On error, `NULL` is returned.

Error Handling

If an error occurs, the function returns `NULL`.

Example

```
vsip_block_f *data_block;
vsip_offset offset = 0;
vsip_stride stride = 1;
vsip_length length = 10;
vsip_vview_f *vector_view;

// Assuming data_block has been properly initialized
vector_view = vsip_vbind_f(data_block, offset, stride, length);

if (vector_view == NULL) {
    // Handle error
}

// The vector view is now bound to the data block
```

1.3.3 vsip_dvcloneview_p - Clone a Vector View

```
vsip_vview_f* vsip_vcloneview_f(const vsip_vview_f* v);
vsip_vview_d* vsip_vcloneview_d(const vsip_vview_d* 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_cvview_f* vsip_cvcloneview_f(const vsip_cvview_f* v);
vsip_cvview_d* vsip_cvcloneview_d(const vsip_cvview_d* v);
```

Description

This function creates a new vector view that shares the same underlying data block as the input vector view but has its own independent view parameters. The cloned view references the same data as the original view but maintains its own metadata (length, stride, offset, and block).

Parameters

- `const vsip_dvview_p* v`: Pointer to the source complex vector view to be cloned.

Return Value

- On success, returns a pointer to the newly created complex vector view that shares data with the input view.
- On error, returns `NULL`.

Example

```
vsip_cvview_f *original_vector;
vsip_cvview_f *cloned_vector;
vsip_length i;

// Create a complex vector
original_vector = vsip_cvcreate_f(10, VSIP_MEM_NONE);

// Clone the vector view
cloned_vector = vsip_cvcloneview_f(original_vector);

if (cloned_vector == NULL) {
    // Handle error
}
```

Notes

- The cloned view shares the same underlying data block as the source complex vector.
- Changes to the data through one view will be visible through all other views that share the same data block.
- The cloned view has the same length, stride, and offset as the original view.

1.3.4 vsip_dvget_p - Get an Element from a Vector View

```
vsip_scalar_f vsip_vget_f(const vsip_vview_f* v, vsip_index j);
vsip_scalar_d vsip_vget_d(const vsip_vview_d* v, vsip_index j);
vsip_scalar_bl vsip_vget_bl(const vsip_vview_bl* v, vsip_index j);
vsip_scalar_vi vsip_vget_vi(const vsip_vview_vi* v, vsip_index j);
vsip_scalar_mi vsip_vget_mi(const vsip_vview_mi* v, vsip_index j);
vsip_cscalar_f vsip_cvget_f(const vsip_cvview_f* v, vsip_index j);
vsip_cscalar_d vsip_cvget_d(const vsip_cvview_d* v, vsip_index j);
```

Description

This function retrieves the element at the specified index *j* from the vector view *v*.

Parameters

- `const vsip_dvview_p* v`: Pointer to the vector view.
- `vsip_index j`: Index of the element to retrieve.

Return Value

- The value of the element at the specified index.

Example

```
vsip_vview_f *vector_view;
vsip_index index = 3;
vsip_scalar_f value;

// Assuming vector_view has been properly initialized
value = vsip_vget_f(vector_view, index);
```

1.3.5 vsip_dvput_p - Set an Element in a Vector View

```
void vsip_vput_f(const vsip_vview_f* v, vsip_index j, vsip_scalar_f x);
void vsip_vput_d(const vsip_vview_d* v, vsip_index j, vsip_scalar_d x);
void vsip_vput_bl(const vsip_vview_bl* v, vsip_index j, vsip_scalar_bl x);
void vsip_vput_vi(const vsip_vview_vi* v, vsip_index j, vsip_scalar_vi x);
void vsip_vput_mi(const vsip_vview_mi* v, vsip_index j, vsip_scalar_mi x);
void vsip_cvput_f(const vsip_cvview_f* v, vsip_index j, vsip_cscalar_f x);
void vsip_cvput_d(const vsip_cvview_d* v, vsip_index j, vsip_cscalar_d x);
```

Description

This function sets the element at the specified index *j* in the vector view *v* to the value *x*.

Parameters

- `const vsip_dvview_p* v`: Pointer to the vector view.
- `vsip_index j`: Index of the element to set.
- `vsip_dscalar_p x`: The new value for the element.

Example

```
vsip_vview_f *vector_view;
vsip_index index = 3;
vsip_scalar_f new_value = 10.0;

// Assuming vector_view has been properly initialized
vsip_vput_f(vector_view, index, new_value);
```

1.3.6 vsip_dvsubview_p - Create a Subview of a Vector View

```
vsip_vview_f* vsip_vsubview_f(const vsip_vview_f* v, vsip_index j, vsip_length n);
vsip_vview_d* vsip_vsubview_d(const vsip_vview_d* v, vsip_index j, vsip_length n);
vsip_vview_bl* vsip_vsubview_bl(const vsip_vview_bl* v, vsip_index j, vsip_length n);
vsip_vview_vi* vsip_vsubview_vi(const vsip_vview_vi* v, vsip_index j, vsip_length n);
vsip_vview_mi* vsip_vsubview_mi(const vsip_vview_mi* v, vsip_index j, vsip_length n);
vsip_cvview_f* vsip_cvsubview_f(const vsip_cvview_f* v, vsip_index j, vsip_length n);
vsip_cvview_d* vsip_cvsubview_d(const vsip_cvview_d* v, vsip_index j, vsip_length n);
```

Description

This function creates a subview of an existing vector view `v`, starting from the index `j` and extending for `n` elements. The subview provides a view into a subset of the original vector view.

Parameters

- `const vsip_dvview_p* v`: Pointer to the original vector view from which the subview will be created.
- `vsip_index j`: Starting index within the original vector view for the subview.
- `vsip_length n`: The number of elements in the subview.

Return Value

- On success, a pointer to the newly created subview object is returned.
- On error, NULL is returned.

Error Handling

If an error occurs, the function returns NULL.

Example

```
vsip_vview_f *original_view;
vsip_index start_index = 5;
vsip_length subview_length = 10;
vsip_vview_f *subview;

// Assuming original_view has been properly initialized
subview = vsip_vsubview_f(original_view, start_index, subview_length);

if (subview == NULL) {
    // Handle error
}

// The subview is now a view into a subset of the original vector view
```

1.3.7 vsip_vrealview_p - Get the Real Part View of a Complex Vector View

```
vsip_vview_f* vsip_vrealview_f(const vsip_cvview_f* v);  
vsip_vview_d* vsip_vrealview_d(const vsip_cvview_d* v);
```

Description

This function returns a view of the real part of the complex vector view *v*.

Parameters

- `const vsip_cvview_p* v`: Pointer to the complex vector view.

Return Value

- On success, a pointer to the real part view of the complex vector view is returned.
- On error, NULL is returned.

Example

```
vsip_cvview_f *complex_vector_view;  
vsip_vview_f *real_part_view;  
  
// Assuming complex_vector_view has been properly initialized  
real_part_view = vsip_vrealview_f(complex_vector_view);  
  
if (real_part_view == NULL) {  
    // Handle error  
}
```


1.3.8 vsip_vimagview_p - Get the Imaginary Part View of a Complex Vector View

```
vsip_vview_f* vsip_vimagview_f(const vsip_cvview_f* v);  
vsip_vview_d* vsip_vimagview_d(const vsip_cvview_d* v);
```

Description

This function returns a view of the imaginary part of the complex vector view v.

Parameters

- `const vsip_cvview_p* v`: Pointer to the complex vector view.

Return Value

- On success, a pointer to the imaginary part view of the complex vector view is returned.
- On error, NULL is returned.

Example

```
vsip_cvview_f *complex_vector_view;  
vsip_vview_f *imaginary_part_view;  
  
// Assuming complex_vector_view has been properly initialized  
imaginary_part_view = vsip_vimagview_f(complex_vector_view);  
  
if (imaginary_part_view == NULL) {  
    // Handle error  
}
```

1.3.9 vsip_dvgetattrib_p - Get the Attributes of a Vector View

```

typedef struct _vsip_vattr_f {
    vsip_offset    offset;
    vsip_stride    stride;
    vsip_length    length;
    vsip_block_f   *block;
} vsip_vattr_f;
/* same for other datatypes with the respective vsip_dblock_p */

void vsip_vgetattrib_f(const vsip_vview_f* v, vsip_vattr_f *a);
void vsip_vgetattrib_d(const vsip_vview_d* v, vsip_vattr_d *a);
void vsip_vgetattrib_i(const vsip_vview_i* v, vsip_vattr_i *a);
void vsip_vgetattrib_bl(const vsip_vview_bl* v, vsip_vattr_bl *a);
void vsip_vgetattrib_vi(const vsip_vview_vi* v, vsip_vattr_vi *a);
void vsip_vgetattrib_mi(const vsip_vview_mi* v, vsip_vattr_mi *a);
void vsip_cvgetattrib_f(const vsip_cvview_f* v, vsip_cvattr_f *a);
void vsip_cvgetattrib_d(const vsip_cvview_d* v, vsip_cvattr_d *a);

```

Description

This function retrieves the attributes of the vector view `v` and stores them in the structure pointed to by `a`.

Parameters

- `const vsip_dvview_p* v`: Pointer to the vector view.
- `vsip_dvattr_p *a`: Pointer to a structure where the attributes will be stored.

Example

```

vsip_vview_f *vector_view;
vsip_vattr_f attributes;

// Assuming vector_view has been properly initialized
vsip_vgetattrib_f(vector_view, &attributes);

// The attributes of the vector view are now stored in 'attributes'

```

1.3.10 vsip_dvputattrib_p - Set the Attributes of a Vector View

```
typedef struct _vsip_vattr_f {
    vsip_offset    offset;
    vsip_stride    stride;
    vsip_length    length;
    vsip_block_f  *block;
} vsip_vattr_f;
/* same for other datatypes with the respective vsip_dblock_p */

vsip_vview_f* vsip_vputattrib_f(vsip_vview_f* v, const vsip_vattr_f *a);
vsip_vview_d* vsip_vputattrib_d(vsip_vview_d* v, const vsip_vattr_d *a);
vsip_vview_i* vsip_vputattrib_i(vsip_vview_i* v, const vsip_vattr_i *a);
vsip_vview_bl* vsip_vputattrib_bl(vsip_vview_bl* v, const vsip_vattr_bl *a);
vsip_vview_vi* vsip_vputattrib_vi(vsip_vview_vi* v, const vsip_vattr_vi *a);
vsip_vview_mi* vsip_vputattrib_mi(vsip_vview_mi* v, const vsip_vattr_mi *a);
vsip_cvview_f* vsip_cvputattrib_f(vsip_cvview_f* v, const vsip_cvattr_f *a);
vsip_cvview_d* vsip_cvputattrib_d(vsip_cvview_d* v, const vsip_cvattr_d *a);
```

Description

This function sets the attributes of the vector view *v* to the values specified in the structure pointed to by *a*.

Parameters

- *vsip_dvview_p* * *v*: Pointer to the vector view.
- *const vsip_dvattr_p* **a*: Pointer to a structure containing the new attributes.

Return Value

- On success, a pointer to the modified vector view is returned.
- On error, NULL is returned.

Example

```
vsip_vview_f *vector_view;
vsip_vattr_f new_attributes;

// Assuming vector_view has been properly initialized and new_attributes is set
vector_view = vsip_vputattrib_f(vector_view, &new_attributes);

if (vector_view == NULL) {
    // Handle error
}
```

1.3.11 vsip_dvgetblock_p - Get the Data Block of a Vector View

```

vsip_block_f* vsip_vgetblock_f(const vsip_vview_f* v);
vsip_block_d* vsip_vgetblock_d(const vsip_vview_d* 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_cblock_f* vsip_cvgetblock_f(const vsip_cvview_f* v);
vsip_cblock_d* vsip_cvgetblock_d(const vsip_cvview_d* v);

```

Description

This function returns the data block associated with the vector view `v`.

Parameters

- `const vsip_dvview_p* v`: Pointer to the vector view.

Return Value

- On success, a pointer to the data block is returned.
- On error, `NULL` is returned.

Example

```

vsip_vview_f *vector_view;
vsip_block_f *data_block;

// Assuming vector_view has been properly initialized
data_block = vsip_vgetblock_f(vector_view);

if (data_block == NULL) {
    // Handle error
}

x

```

1.3.12 vsip_dvgetlength_p - Get the Length of a Vector View

```
vsip_length vsip_vgetlength_f(const vsip_vview_f* v);  
vsip_length vsip_vgetlength_d(const vsip_vview_d* 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_cvgetlength_f(const vsip_cvview_f* v);  
vsip_length vsip_cvgetlength_d(const vsip_cvview_d* v);
```

Description

This function returns the length of the vector view v.

Parameters

- const vsip_dvview_p* v: Pointer to the vector view.

Return Value

- The length of the vector view.

Example

```
vsip_vview_f *vector_view;  
vsip_length length;  
  
// Assuming vector_view has been properly initialized  
length = vsip_vgetlength_f(vector_view);
```

1.3.13 vsip_dvputlength_p - Set the Length of a Vector View

```

vsip_vview_f* vsip_vputlength_f(vsip_vview_f* v, vsip_length n);
vsip_vview_d* vsip_vputlength_d(vsip_vview_d* v, vsip_length n);
vsip_vview_bl* vsip_vputlength_bl(vsip_vview_bl* v, vsip_length n);
vsip_vview_vi* vsip_vputlength_vi(vsip_vview_vi* v, vsip_length n);
vsip_vview_mi* vsip_vputlength_mi(vsip_vview_mi* v, vsip_length n);
vsip_cvview_f* vsip_cvputlength_f(vsip_cvview_f* v, vsip_length n);
vsip_cvview_d* vsip_cvputlength_d(vsip_cvview_d* v, vsip_length n);

```

Description

This function sets the length of the vector view `v` to the specified value `n`.

Parameters

- `vsip_dvview_p* v`: Pointer to the vector view.
- `vsip_length n`: The new length of the vector view.

Return Value

- On success, a pointer to the modified vector view is returned.
- On error, `NULL` is returned.

Example

```

vsip_vview_f *vector_view;
vsip_length new_length = 15;

// Assuming vector_view has been properly initialized
vector_view = vsip_vputlength_f(vector_view, new_length);

if (vector_view == NULL) {
    // Handle error
}

```

1.3.14 vsip_dvgetstride_p - Get the Stride of a Vector View

```
vsip_stride vsip_vgetstride_f(const vsip_vview_f* v);  
vsip_stride vsip_vgetstride_d(const vsip_vview_d* 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_cvgetstride_f(const vsip_cvview_f* v);  
vsip_stride vsip_cvgetstride_d(const vsip_cvview_d* v);
```

Description

This function returns the stride between elements in the vector view v.

Parameters

- const vsip_dvview_p* v: Pointer to the vector view.

Return Value

- The stride between elements in the vector view.

Example

```
vsip_vview_f *vector_view;  
vsip_stride stride;  
  
// Assuming vector_view has been properly initialized  
stride = vsip_vgetstride_f(vector_view);
```

1.3.15 vsip_dvputstride_p - Set the Stride of a Vector View

```

vsip_vview_f* vsip_vputstride_f(vsip_vview_f* v, vsip_stride s);
vsip_vview_d* vsip_vputstride_d(vsip_vview_d* v, vsip_stride s);
vsip_vview_bl* vsip_vputstride_bl(vsip_vview_bl* v, vsip_stride s);
vsip_vview_vi* vsip_vputstride_vi(vsip_vview_vi* v, vsip_stride s);
vsip_vview_mi* vsip_vputstride_mi(vsip_vview_mi* v, vsip_stride s);
vsip_cvview_f* vsip_cvputstride_f(vsip_cvview_f* v, vsip_stride s);
vsip_cvview_d* vsip_cvputstride_d(vsip_cvview_d* v, vsip_stride s);

```

Description

This function sets the stride between elements in the vector view `v` to the specified value `s`.

Parameters

- `vsip_dvview_p* v`: Pointer to the vector view.
- `vsip_stride s`: The new stride between elements.

Return Value

- On success, a pointer to the modified vector view is returned.
- On error, `NULL` is returned.

Example

```

vsip_vview_f *vector_view;
vsip_stride new_stride = 2;

// Assuming vector_view has been properly initialized
vector_view = vsip_vputstride_f(vector_view, new_stride);

if (vector_view == NULL) {
    // Handle error
}

```


1.3.16 vsip_dvgetoffset_p - Get the Offset of a Vector View

```
vsip_offset vsip_vgetoffset_f(const vsip_vview_f* v);  
vsip_offset vsip_vgetoffset_d(const vsip_vview_d* 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_cvgetoffset_f(const vsip_cvview_f* v);  
vsip_offset vsip_cvgetoffset_d(const vsip_cvview_d* v);
```

Description

This function returns the offset within the data block where the vector view *v* starts.

Parameters

- `const vsip_dvview_p* v`: Pointer to the vector view.

Return Value

- The offset within the data block.

Example

```
vsip_vview_f *vector_view;  
vsip_offset offset;  
  
// Assuming vector_view has been properly initialized  
offset = vsip_vgetoffset_f(vector_view);
```

1.3.17 vsip_dvputoffset_p - Set the Offset of a Vector View

```

vsip_vview_f* vsip_vputoffset_f(vsip_vview_f* v, vsip_offset o);
vsip_vview_d* vsip_vputoffset_d(vsip_vview_d* v, vsip_offset o);
vsip_vview_bl* vsip_vputoffset_bl(vsip_vview_bl* v, vsip_offset o);
vsip_vview_vi* vsip_vputoffset_vi(vsip_vview_vi* v, vsip_offset o);
vsip_vview_mi* vsip_vputoffset_mi(vsip_vview_mi* v, vsip_offset o);
vsip_cvview_f* vsip_cvputoffset_f(vsip_cvview_f* v, vsip_offset o);
vsip_cvview_d* vsip_cvputoffset_d(vsip_cvview_d* v, vsip_offset o);

```

Description

This function sets the offset within the data block for the vector view *v* to the specified value *o*.

Parameters

- *vsip_dvview_p* * *v*: Pointer to the vector view.
- *vsip_offset* *o*: The new offset within the data block.

Return Value

- On success, a pointer to the modified vector view is returned.
- On error, NULL is returned.

Example

```

vsip_vview_f *vector_view;
vsip_offset new_offset = 5;

// Assuming vector_view has been properly initialized
vector_view = vsip_vputoffset_f(vector_view, new_offset);

if (vector_view == NULL) {
    // Handle error
}

```

1.3.18 vsip_dvdestroy_p - Destroy a Vector View

```

vsip_block_f* vsip_vdestroy_f(vsip_vview_f* v);
vsip_block_d* vsip_vdestroy_d(vsip_vview_d* v);
vsip_block_i* vsip_vdestroy_i(vsip_vview_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_cblock_f* vsip_cvdestroy_f(vsip_cvview_f* v);
vsip_cblock_d* vsip_cvdestroy_d(vsip_cvview_d* v);

```

Description

This function destroys a vector view `v` and returns a pointer to the underlying data block. After calling this function, the vector view is no longer valid, but the data block can still be used.

Parameters

- `vsip_dvview_p* v`: Pointer to the vector view to be destroyed.

Return Value

- On success, a pointer to the underlying data block is returned.
- On error, `NULL` is returned.

Error Handling

If an error occurs, the function returns `NULL`.

Example

```

vsip_vview_f *vector_view;
vsip_block_f *data_block;

// Assuming vector_view has been properly initialized
data_block = vsip_vdestroy_f(vector_view);

if (data_block == NULL) {
    // Handle error
}

// The data block can still be used after the vector view is destroyed

```

1.3.19 vsip_dvalldestroy_p - Destroy a Vector View and Its Data Block

```
void vsip_valldestroy_f(vsip_vview_f *v);
void vsip_valldestroy_d(vsip_vview_d *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_cvalldestroy_f(vsip_cvview_f* v);
void vsip_cvalldestroy_d(vsip_cvview_d* v);
```

Description

This function destroys a vector view *v* and its underlying data block. After calling this function, both the vector view and the data block are no longer valid.

Parameters

- *vsip_dvview_p *v*: Pointer to the vector view to be destroyed along with its data block.

Example

```
vsip_vview_f *vector_view;

// Assuming vector_view has been properly initialized
vsip_valldestroy_f(vector_view);

// Both the vector view and its data block are now invalid
```

1.4 Matrix View Support Functions

1.4.1 vsip_dmcreate_p - Create a Matrix View

```

typedef enum _vsip_memory_hint {
    VSIP_MEM_NONE          = 0,
    VSIP_MEM_RDONLY        = 1,
    VSIP_MEM_CONST         = 2,
    VSIP_MEM_SHARED        = 3,
    VSIP_MEM_SHARED_RDONLY = 4,
    VSIP_MEM_SHARED_CONST  = 5
} vsip_memory_hint;

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

vsip_mview_f* vsip_mcreate_f(vsip_length row_length, vsip_length col_length,
                             vsip_major major, vsip_mem_hint hint);
vsip_mview_d* vsip_mcreate_d(vsip_length row_length, vsip_length col_length,
                             vsip_major major, vsip_mem_hint hint);
vsip_cmview_f* vsip_cmcreate_f(vsip_length row_length, vsip_length col_length,
                                vsip_major major, vsip_mem_hint hint);
vsip_cmview_d* vsip_cmcreate_d(vsip_length row_length, vsip_length col_length,
                                vsip_major major, vsip_mem_hint hint);

```

Description

This function creates a new matrix view with the specified dimensions. The function allocates both a data block and a matrix view, and binds them together.

Whether the matrix is stored in row- or column major order can be selected using the major argument.

Parameters

- vsip_length row_length: Number of rows in the matrix.
- vsip_length col_length: Number of columns in the matrix.
- vsip_major major: Whether the matrix is supposed to be row- or column major.
- vsip_mem_hint hint: Memory allocation hint that can be used to optimize memory access.
 - VSIP_MEM_NONE - No memory hint
 - VSIP_MEM_RDONLY - The memory is to be used read-only
 - VSIP_MEM_CONST - The memory will hold constants
 - VSIP_MEM_SHARED - The memory will be shared
 - VSIP_MEM_SHARED_RDONLY - The memory will be shared and is read-only
 - VSIP_MEM_SHARED_CONST - The memory will be shared and will hold constants

Return Value

- On success, returns a pointer to the newly created matrix view.
- On error, returns NULL.

Example

```

vsip_mview_f *matrix;
vsip_length rows = 100;
vsip_length cols = 100;

```

```
// Create a 100x100 matrix initialized to 0.0
```

```
matrix = vsip_mcreate_f(rows, cols, VSIP_ROW, VSIP_MEM_NONE);

if (matrix == NULL) {
    // Handle error
}
```

Notes

- The created matrix has contiguous memory layout with unit strides in both dimensions.
- This function is equivalent to calling `vsip_blockcreate_p`, then `vsip_mbind_p`, and finally filling the matrix with the specified value.

1.4.2 vsip_dmbind_p - Bind a Matrix View to a Block

```
void vsip_mbind_f(const vsip_block_f* block, vsip_offset offset,
                 vsip_stride col_stride, vsip_stride col_length,
                 vsip_length row_length, vsip_length row_length);
void vsip_mbind_d(const vsip_block_d* block, vsip_offset offset,
                 vsip_stride col_stride, vsip_stride col_length,
                 vsip_length row_length, vsip_length row_length);
void vsip_cmbind_f(const vsip_cblock_f* block, vsip_offset offset,
                 vsip_stride col_stride, vsip_stride col_length,
                 vsip_length row_length, vsip_length row_length);
void vsip_cmbind_d(const vsip_cblock_d* block, vsip_offset offset,
                 vsip_stride col_stride, vsip_stride col_length,
                 vsip_length row_length, vsip_length row_length);
```

Description

This function binds a matrix view to a section of a data block, allowing access to the block's data through the matrix view interface. The binding specifies the location of the matrix within the block, the strides between elements, and the dimensions of the matrix.

The matrix view becomes a "window" into the block, with the specified dimensions and strides. This allows for efficient access to submatrices or non-contiguous sections of a larger data block without copying data.

Parameters

- `const vsip_dblock_p* block`: Pointer to the block of data to bind to.
- `vsip_offset offset`: The offset (in elements) from the start of the block to the first element of the matrix (0,0 position).
- `vsip_stride col_stride`: The stride (in elements) between consecutive columns of the matrix.
- `vsip_length col_length`: The number of columns in the matrix view.
- `vsip_stride row_stride`: The stride (in elements) between consecutive rows of the matrix.
- `vsip_length row_length`: The number of rows in the matrix view.

Example

```
vsip_block_f *block;
vsip_mview_f matrix_view;
vsip_scalar_f *data;
vsip_length block_size = 1000;

// Allocate a block of data
block = vsip_blockcreate_f(block_size, VSIP_MEM_NONE);

// Populate block with data here

// Bind a 10x10 matrix view to the block starting at offset 0
// with contiguous memory layout (col_stride = 1, row_stride = 10)
matrix_view = vsip_mbind_f(block, 0, 1, 10, 10, 10);
```

Notes

- The block must be large enough to contain the matrix view with the specified strides.
- The strides determine how elements are accessed in memory:
 - `col_stride` is the step size between columns (typically 1 for contiguous columns)
 - `row_stride` is the step size between rows (typically equal to the number of columns for contiguous rows)

- Non-unit strides allow for accessing non-contiguous sections of the block.
- The matrix view does not own the data; the block must remain valid as long as the view is in use.
- This function is useful for creating views of submatrices or for implementing specialized matrix layouts.

1.4.3 vsip_dmcloneview_p - Clone a Matrix View

```
vsip_mview_f* vsip_mcloneview_f(const vsip_mview_f* matrix);
vsip_mview_d* vsip_mcloneview_d(const vsip_mview_d* matrix);
vsip_cmview_f* vsip_cmcloneview_f(const vsip_cmview_f* matrix);
vsip_cmview_d* vsip_cmcloneview_d(const vsip_cmview_d* matrix);
```

Description

This function creates a new matrix view that shares the same data block as the input matrix view but has its own independent view parameters. The cloned view references the same underlying data but maintains its own metadata (dimensions, strides, offset).

This is useful when you need multiple independent views of the same data, or when you want to create a view with different parameters (like different submatrix boundaries) while sharing the same data storage.

Parameters

- `const vsip_dmview_p* matrix`: Pointer to the source matrix view to be cloned.

Return Value

- On success, returns a pointer to the newly created matrix view that shares data with the input view.
- On error, returns NULL.

Example

```
vsip_mview_f *original_matrix;
vsip_mview_f *cloned_matrix;

// Clone the matrix view
cloned_matrix = vsip_mcloneview_f(original_matrix);

if (cloned_matrix == NULL) {
    // Handle error
}
```

Notes

- The cloned view shares the same underlying data block as the original view.
- Changes to the data through one view will be visible through all other views that share the same data block.
- The cloned view has the same dimensions, strides, and offset as the original view.
- This function is useful for creating multiple independent views of the same data without copying the actual data.
- To create a completely independent copy (including the data), use `vsip_dmcopy_p_p` to copy to a new matrix.

1.4.4 vsip_dmget_p - Get Matrix Element

```
vsip_scalar_f vsip_mget_f(const vsip_mview_f *v, vsip_index i, vsip_index j);
vsip_scalar_d vsip_mget_d(const vsip_mview_d *v, vsip_index i, vsip_index j);
vsip_cscalar_f vsip_cmget_f(const vsip_cmview_f *v, vsip_index i, vsip_index j);
vsip_cscalar_d vsip_cmget_d(const vsip_cmview_d *v, vsip_index i, vsip_index j);
```

Description

This function retrieves the value of a specific element from a matrix view. The element is identified by its row and column indices (0-based).

Parameters

- `const vsip_dmview_p* v`: Pointer to the matrix view.
- `vsip_index i`: Row index of the element to retrieve (0-based).
- `vsip_index j`: Column index of the element to retrieve (0-based).

Return Value

- Returns the value of the matrix element at position (i,j) as a `vsip_dscalar_p`.

Example

```
vsip_mview_f *matrix;
vsip_scalar_f value;
vsip_index i, j;

// Create and initialize a matrix
matrix = vsip_mcreate_f(5, 5, VSIP_ROW, VSIP_MEM_NONE);

// Fill the matrix with some values
for (i = 0; i < 5; i++) {
    for (j = 0; j < 5; j++) {
        vsip_mput_f(matrix, i, j, i * 5 + j + 1);
    }
}

// Retrieve specific elements
value = vsip_mget_f(matrix, 0, 0); // Top-left corner
printf("Element at (0,0): %f\n", value);

value = vsip_mget_f(matrix, 2, 3); // Middle element
printf("Element at (2,3): %f\n", value);

value = vsip_mget_f(matrix, 4, 4); // Bottom-right corner
printf("Element at (4,4): %f\n", value);
```

Notes

- The function does not perform bounds checking and may return an error or undefined value if the indices are out of range.
- For submatrix views, the indices are relative to the submatrix, not the parent matrix.

1.4.5 vsip_dmput_p - Set Matrix Element

```
void vsip_mput_f(const vsip_mview_f *v, vsip_index i, vsip_index j, vsip_scalar_f vv);
void vsip_mput_d(const vsip_mview_d *v, vsip_index i, vsip_index j, vsip_scalar_d vv);
void vsip_cput_f(const vsip_cmview_f *v, vsip_index i, vsip_index j, vsip_cscalar_f vv);
void vsip_cput_d(const vsip_cmview_d *v, vsip_index i, vsip_index j, vsip_cscalar_d vv);
```

Description

This function sets the value of a specific element in a matrix view. The element is identified by its row and column indices (0-based).

Parameters

- `const vsip_dmview_p* v`: Pointer to the matrix view.
- `vsip_index i`: Row index of the element to set (0-based).
- `vsip_index j`: Column index of the element to set (0-based).
- `vsip_dscalar_p vv`: The value to assign to the matrix element.

Example

```
vsip_mview_f *matrix;
vsip_index i, j;

// Create a matrix
matrix = vsip_mcreate_f(5, 5, VSIP_ROW, VSIP_MEM_NONE);

// Set specific elements
vsip_mput_f(matrix, 0, 0, 1.0f); // Top-left corner
vsip_mput_f(matrix, 2, 2, 5.0f); // Center element
vsip_mput_f(matrix, 4, 4, 9.0f); // Bottom-right corner
```

Notes

- The indices are 0-based (first row/column is index 0).
- The function does not perform bounds checking and may cause a memory access error.
- For submatrix views, the indices are relative to the submatrix, not the parent matrix.

1.4.6 vsip_dmsubview_p - Create a Submatrix View

```
vsip_mview_f* vsip_msubview_f(const vsip_mview_f* matrix,
                             vsip_index row_offset, vsip_index col_offset,
                             vsip_length row_length, vsip_length col_length);
vsip_mview_d* vsip_msubview_d(const vsip_mview_d* matrix,
                             vsip_index row_offset, vsip_index col_offset,
                             vsip_length row_length, vsip_length col_length);
vsip_cmview_f* vsip_cmsubview_f(const vsip_cmview_f* matrix,
                                vsip_index row_offset, vsip_index col_offset,
                                vsip_length row_length, vsip_length col_length);
vsip_cmview_d* vsip_cmsubview_d(const vsip_cmview_d* matrix,
                                vsip_index row_offset, vsip_index col_offset,
                                vsip_length row_length, vsip_length col_length);
```

Description

This function creates a new matrix view that represents a submatrix of an existing matrix view. The submatrix is defined by its offset from the parent matrix and its dimensions. The new view shares the same underlying data block as the parent matrix but provides access to only the specified subregion.

This operation is efficient as it doesn't copy any data, but rather creates a new view that references a portion of the original matrix's data.

Parameters

- `const vsip_mview_f* matrix`: Pointer to the source matrix view.
- `vsip_index row_offset`: The row offset of the submatrix from the parent matrix (0-based).
- `vsip_index col_offset`: The column offset of the submatrix from the parent matrix (0-based).
- `vsip_length row_length`: The number of rows in the submatrix.
- `vsip_length col_length`: The number of columns in the submatrix.

Return Value

- On success, returns a pointer to the newly created submatrix view.
- On error (e.g., if the submatrix would extend beyond the parent matrix boundaries), returns `NULL`.

Example

```
vsip_mview_f *parent_matrix;
vsip_mview_f *submatrix;

// Create a parent matrix
parent_matrix = vsip_mcreate_f(100, 100, VSIP_ROW, VSIP_MEM_NONE);

// Create a 50x50 submatrix starting at row 25, column 25
submatrix = vsip_msubview_f(parent_matrix, 25, 25, 50, 50);

if (submatrix == NULL) {
    // Handle error (e.g., invalid submatrix dimensions)
}
```

Notes

- The submatrix view shares the same underlying data block as the parent matrix.
- Modifications to the submatrix will affect the parent matrix and vice versa.
- The submatrix must be entirely contained within the parent matrix.

- The strides of the submatrix are inherited from the parent matrix.
- This function is useful for working with portions of a matrix without copying data.
- For non-contiguous submatrices or more complex views, consider using `vsip_dmbind_p` directly.

1.4.7 vsip_dmtransview_p - Create a Transposed Matrix View

```
vsip_mview_f* vsip_mtransview_f(const vsip_mview_f* matrix);
vsip_mview_d* vsip_mtransview_d(const vsip_mview_d* matrix);
vsip_cmview_f* vsip_cmtransview_f(const vsip_cmview_f* matrix);
vsip_cmview_d* vsip_cmtransview_d(const vsip_cmview_d* matrix);
```

Description

This function creates a new matrix view that represents the transpose of the input matrix. The transposed view shares the same underlying data block as the original matrix but presents it with rows and columns swapped. This operation is efficient as it doesn't copy any data, but rather creates a new view with transposed dimensions and strides.

For an $m \times n$ input matrix, the transposed view will be an $n \times m$ matrix where the element at position (i, j) in the transposed view corresponds to the element at position (j, i) in the original matrix.

Parameters

- `const vsip_dmview_p* matrix`: Pointer to the source matrix view to be transposed.

Return Value

- On success, returns a pointer to the newly created transposed matrix view.
- On error, returns NULL.

Example

```
vsip_mview_f *original_matrix;
vsip_mview_f *transposed_matrix;
vsip_length i, j;

// Create a 4x3 matrix
original_matrix = vsip_mcreate_f(4, 3, VSIP_ROW, VSIP_MEM_NONE);

// Fill the matrix with some values
for (i = 0; i < 4; i++) {
    for (j = 0; j < 3; j++) {
        vsip_mput_f(original_matrix, i, j, i * 3 + j + 1);
    }
}

// Create a transposed view (3x4)
transposed_matrix = vsip_mtransview_f(original_matrix);

if (transposed_matrix == NULL) {
    // Handle error
}

// Now transposed_matrix is a 3x4 view of the original 4x3 matrix data
// Accessing transposed_matrix[0][1] is equivalent to original_matrix[1][0]
```

Notes

- The transposed view shares the same underlying data block as the original matrix.
- Modifications to the transposed view will affect the original matrix and vice versa.
- The transposed view has swapped dimensions compared to the original matrix.
- The strides of the transposed view are adjusted to provide the transposed access pattern.
- This operation is efficient as it doesn't copy any data, only creates a new view.

1.4.8 vsip_dmrowview_p - Create a Row Vector View of a Matrix

```
vsip_vview_f* vsip_mrowview_f(const vsip_mview_f* matrix, vsip_index row_index);
vsip_vview_d* vsip_mrowview_d(const vsip_mview_d* matrix, vsip_index row_index);
vsip_cvview_f* vsip_cmrowview_f(const vsip_cmview_f* matrix, vsip_index row_index);
vsip_cvview_d* vsip_cmrowview_d(const vsip_cmview_d* matrix, vsip_index row_index);
```

Description

This function creates a vector view that represents a single row of a matrix. The resulting vector view shares the same underlying data block as the matrix but provides access to only the specified row. This operation is efficient as it doesn't copy any data, but rather creates a new view that references the row data.

The created vector view has a length equal to the number of columns in the source matrix. The vector view maintains the same data type as the matrix elements.

Parameters

- `const vsip_dmview_p* matrix`: Pointer to the source matrix view.
- `vsip_index row_index`: The index of the row to extract (0-based).

Return Value

- On success, returns a pointer to the newly created vector view representing the specified row.
- On error (e.g., if the row index is out of bounds), returns `NULL`.

Example

```
vsip_mview_f *matrix;
vsip_vview_f *row_vector;
vsip_length i, j;

// Create a 5x10 matrix
matrix = vsip_mcreate_f(5, 10, VSIP_ROW, VSIP_MEM_NONE);

// Fill the matrix with some values
for (i = 0; i < 5; i++) {
    for (j = 0; j < 10; j++) {
        vsip_mput_f(matrix, i, j, i * 10 + j + 1);
    }
}

// Get a vector view of the 3rd row (index 2)
row_vector = vsip_mrowview_f(matrix, 2);

if (row_vector == NULL) {
    // Handle error
}

// Now row_vector represents the 3rd row of the matrix
// and has length equal to the number of columns (10)
```

Notes

- The row vector view shares the same underlying data block as the source matrix.
- Modifications to the row vector will affect the source matrix and vice versa.
- The row index must be within the valid range of the matrix ($0 \leq \text{row_index} < \text{number of rows}$).
- The created vector view has a length equal to the number of columns in the source matrix.

- The vector view maintains the same stride as the row stride of the source matrix.
- This operation is efficient as it doesn't copy any data, only creates a new view.

1.4.9 vsip_dmcolview_p - Create a Column Vector View of a Matrix

```
vsip_vview_f* vsip_mcolview_f(const vsip_mview_f* matrix, vsip_index col_index);
vsip_vview_d* vsip_mcolview_d(const vsip_mview_d* matrix, vsip_index col_index);
vsip_cvview_f* vsip_cmcolview_f(const vsip_cmview_f* matrix, vsip_index col_index);
vsip_cvview_d* vsip_cmcolview_d(const vsip_cmview_d* matrix, vsip_index col_index);
```

Description

This function creates a vector view that represents a single column of a matrix. The resulting vector view shares the same underlying data block as the matrix but provides access to only the specified column. This operation is efficient as it doesn't copy any data, but rather creates a new view that references the row data.

The created vector view has a length equal to the number of rows in the source matrix. The vector view maintains the same data type as the matrix elements.

Parameters

- `const vsip_dmview_p* matrix`: Pointer to the source matrix view.
- `vsip_index col_index`: The index of the column to extract (0-based).

Return Value

- On success, returns a pointer to the newly created vector view representing the specified column.
- On error (e.g., if the column index is out of bounds), returns NULL.

Example

```
vsip_mview_f *matrix;
vsip_vview_f *col_vector;
vsip_length i, j;

// Create a 5x10 matrix
matrix = vsip_mcreate_f(5, 10, VSIP_ROW, VSIP_MEM_NONE);

// Fill the matrix with some values
for (i = 0; i < 5; i++) {
    for (j = 0; j < 10; j++) {
        vsip_mput_f(matrix, i, j, i * 10 + j + 1);
    }
}

// Get a vector view of the 3rd row (index 2)
col_vector = vsip_mcolview_f(matrix, 2);

if (col_vector == NULL) {
    // Handle error
}

// Now col_vector represents the 3rd row of the matrix
// and has length equal to the number of rows (5)
```

Notes

- The column vector view shares the same underlying data block as the source matrix.
- Modifications to the column vector will affect the source matrix and vice versa.
- The column index must be within the valid range of the matrix ($0 \leq \text{col_index} < \text{number of columns}$).
- The created vector view has a length equal to the number of rows in the source matrix.

- The vector view maintains the same stride as the column stride of the source matrix.
- This operation is efficient as it doesn't copy any data, only creates a new view.

1.4.10 vsip_dmdiagview_p - Create a Diagonal Vector View of a Matrix

```
vsip_vview_f* vsip_mdiagview_f(const vsip_mview_f* matrix, vsip_index diagonal);
vsip_vview_d* vsip_mdiagview_d(const vsip_mview_d* matrix, vsip_index diagonal);
vsip_cvview_f* vsip_cmdiagview_f(const vsip_cmview_f* matrix, vsip_index diagonal);
vsip_cvview_d* vsip_cmdiagview_d(const vsip_cmview_d* matrix, vsip_index diagonal);
```

Description

This function creates a vector view that represents a diagonal of a matrix. The resulting vector view shares the same underlying data block as the matrix but provides access to only the elements along the specified diagonal.

The diagonal is specified by an index where:

- Index 0 represents the main diagonal
- Positive indices represent super-diagonals (above the main diagonal)
- Negative indices represent sub-diagonals (below the main diagonal)

The length of the resulting vector depends on the diagonal index and the matrix dimensions. For a diagonal with index d in an $m \times n$ matrix, the length of the vector is:

$$\min(m - \max(0, -d), n - \max(0, d))$$

Parameters

- `const vsip_dmview_p* matrix`: Pointer to the source matrix view.
- `vsip_index diagonal`: The index of the diagonal to extract:
 - 0: Main diagonal
 - >0: Super-diagonal (above main diagonal)
 - <0: Sub-diagonal (below main diagonal)

Return Value

- On success, returns a pointer to the newly created vector view representing the specified diagonal.
- On error (e.g., if the diagonal index is invalid for the matrix dimensions), returns NULL.

Example

```
vsip_mview_f *matrix;
vsip_vview_f *diag_vector;
vsip_length i, j;

// Create a 5x5 matrix
matrix = vsip_mcreate_f(5, 5, VSIP_ROW, VSIP_MEM_NONE);

// Fill the matrix with some values
for (i = 0; i < 5; i++) {
    for (j = 0; j < 5; j++) {
        vsip_mput_f(matrix, i, j, i * 5 + j + 1);
    }
}

// Get a vector view of the main diagonal (index 0)
diag_vector = vsip_mdiagview_f(matrix, 0);

if (diag_vector == NULL) {
    // Handle error
}
```

Notes

- The diagonal vector view shares the same underlying data block as the source matrix.
- Modifications to the diagonal vector will affect the source matrix and vice versa.
- The diagonal index must be valid for the matrix dimensions (i.e., the diagonal must exist in the matrix).
- The length of the resulting vector depends on the diagonal index and matrix dimensions.
- For the main diagonal (index 0) of an $n \times n$ matrix, the vector length is n .
- For super-diagonals (index > 0), the vector length is $n - \text{index}$.
- For sub-diagonals (index < 0), the vector length is $n + \text{index}$.
- This operation is efficient as it doesn't copy any data, only creates a new view.

1.4.11 vsip_mrealview_p - Create a Real Part Matrix View

```
vsip_mview_f* vsip_mrealview_f(const vsip_cmview_f* cmatrix);
vsip_mview_d* vsip_mrealview_d(const vsip_cmview_d* cmatrix);
```

Description

This function creates a real matrix view that represents the real parts of a complex matrix. The resulting matrix view shares the same underlying data block as the complex matrix but provides access to only the real components of each complex element.

For a complex matrix A with elements $a_{ij} = x_{ij} + iy_{ij}$, the real view matrix B will have elements $b_{ij} = x_{ij}$.

Parameters

- `const vsip_cmview_p* cmatrix`: Pointer to the source complex matrix view.

Return Value

- On success, returns a pointer to the newly created real matrix view representing the real parts.
- On error, returns NULL.

Example

```
vsip_cmview_f *complex_matrix;
vsip_mview_f *real_matrix;
vsip_length i, j;

// Create a 3x3 complex matrix
complex_matrix = vsip_cmcreate_f(3, 3, VSIP_ROW, VSIP_MEM_NONE);

// Fill with complex values
for (i = 0; i < 3; i++) {
    for (j = 0; j < 3; j++) {
        vsip_cput_f(complex_matrix, i, j,
                    VSIP_CMPLX_F(i*3+j+1, (i*3+j+1)*0.1f));
    }
}

// Create a real view of the complex matrix
real_matrix = vsip_mrealview_f(complex_matrix);

if (real_matrix == NULL) {
    // Handle error
}
```

Notes

- The real matrix view shares the same underlying data block as the source complex matrix.
- Modifications to the real matrix view will affect the real parts of the complex matrix and vice versa.
- The real matrix view has the same dimensions as the source complex matrix.
- This operation is efficient as it doesn't copy any data, only creates a new view.

1.4.12 vsip_mimagview_p - Create an Imaginary Part Matrix View

```
vsip_mview_f* vsip_mimagview_f(const vsip_cmview_f* cmatrix);
vsip_mview_d* vsip_mimagview_d(const vsip_cmview_d* cmatrix);
```

Description

This function creates a real matrix view that represents the imaginary parts of a complex matrix. The resulting matrix view shares the same underlying data block as the complex matrix but provides access to only the imaginary components of each complex element.

For a complex matrix A with elements $a_{ij} = x_{ij} + iy_{ij}$, the imaginary view matrix B will have elements $b_{ij} = y_{ij}$.

Parameters

- `const vsip_cmview_p* cmatrix`: Pointer to the source complex matrix view.

Return Value

- On success, returns a pointer to the newly created real matrix view representing the imaginary parts.
- On error, returns NULL.

Example

```
vsip_cmview_f *complex_matrix;
vsip_mview_f *imag_matrix;
vsip_length i, j;

// Create a 3x3 complex matrix
complex_matrix = vsip_cmcreate_f(3, 3, VSIP_ROW, VSIP_MEM_NONE);

// Fill with complex values
for (i = 0; i < 3; i++) {
    for (j = 0; j < 3; j++) {
        vsip_cput_f(complex_matrix, i, j,
                    VSIP_CMPLX_F(i*3+j+1, (i*3+j+1)*0.1f));
    }
}

// Create an imaginary view of the complex matrix
imag_matrix = vsip_mimagview_f(complex_matrix);

if (imag_matrix == NULL) {
    // Handle error
}
```

Notes

- The imaginary matrix view shares the same underlying data block as the source complex matrix.
- Modifications to the imaginary matrix view will affect the imaginary parts of the complex matrix and vice versa.
- The imaginary matrix view has the same dimensions as the source complex matrix.
- This operation is efficient as it doesn't copy any data, only creates a new view.

1.4.13 vsip_dmgetattrib_p - Get Matrix Attributes

```
typedef struct {
    vsip_length row_length;    /* Number of rows */
    vsip_length col_length;    /* Number of columns */
    vsip_offset offset;        /* Offset into the block */
    vsip_stride row_stride;    /* Stride between rows */
    vsip_stride col_stride;    /* Stride between columns */
    vsip_block_f* block;       /* Pointer to the data block */
} vsip_mattr_f;
/* same for the other datatypes with the respective vsip_dmview_p */

void vsip_mgetattrib_f(const vsip_mview_f* v, vsip_mattr_f* attr);
void vsip_mgetattrib_d(const vsip_mview_d* v, vsip_mattr_d* attr);
void vsip_cmgetattrib_f(const vsip_cmview_f* v, vsip_cmattr_f* attr);
void vsip_cmgetattrib_d(const vsip_cmview_d* v, vsip_cmattr_d* attr);
```

Description

This function retrieves all attributes of a matrix view and stores them in a `vsip_dmattr_p` structure. The attribute structure contains complete information about the matrix view's dimensions, memory layout, and binding to its data block.

Parameters

- `const vsip_dmview_p* v`: Pointer to the matrix view.
- `vsip_dmattr_p* attr`: Pointer to the attribute structure where the matrix attributes will be stored.

Example

```
vsip_mview_f *matrix;
vsip_mattr_f attr;

// Create a matrix
matrix = vsip_mcreate_f(100, 100, VSIP_ROW, VSIP_MEM_NONE);

// Get all matrix attributes
vsip_mgetattrib_f(matrix, &attr);

printf("Matrix attributes:\n");
printf("  Dimensions: %lu x %lu\n", attr.row_length, attr.col_length);
printf("  Memory offset: %ld\n", attr.offset);
printf("  Row stride: %ld\n", attr.row_stride);
printf("  Column stride: %ld\n", attr.col_stride);
printf("  Block pointer: %p\n", (void*)attr.block);

// Create a submatrix view and examine its attributes
vsip_mview_f *submatrix = vsip_msubview_f(matrix, 10, 10, 50, 50);
vsip_mgetattrib_f(submatrix, &attr);

printf("\nSubmatrix attributes:\n");
printf("  Dimensions: %lu x %lu\n", attr.row_length, attr.col_length);
printf("  Memory offset: %ld\n", attr.offset);

// Create a transposed view and examine its attributes
vsip_mview_f *transposed = vsip_mtransview_f(matrix);
vsip_mgetattrib_f(transposed, &attr);

printf("\nTransposed matrix attributes:\n");
printf("  Dimensions: %lu x %lu\n", attr.row_length, attr.col_length);
```



```
printf("  Row stride: %ld\n", attr.row_stride);  
printf("  Column stride: %ld\n", attr.col_stride);
```

Notes

- The `vsip_mattr_p` structure contains all information needed to completely describe a matrix view.
- The `block` field points to the underlying data block that stores the matrix elements.
- For row-major matrices, `col_stride` is typically 1 and `row_stride` equals the number of columns.
- For column-major matrices, `row_stride` is typically 1 and `col_stride` equals the number of rows.
- For transposed views, the row and column strides are swapped compared to the original matrix.
- The `offset` indicates how many elements from the start of the block the matrix begins at.

1.4.14 vsip_dmputattrib_p - Set Matrix Attributes

```
typedef struct {
    vsip_length row_length;    /* Number of rows */
    vsip_length col_length;    /* Number of columns */
    vsip_offset offset;        /* Offset into the block */
    vsip_stride row_stride;    /* Stride between rows */
    vsip_stride col_stride;    /* Stride between columns */
    vsip_block_f* block;       /* Pointer to the data block */
} vsip_mattr_f;
/* same for the other datatypes with the respective vsip_dmview_p */

vsip_mview_f* vsip_mputattrib_f(vsip_mview_f* v, const vsip_mattr_f* attr);
vsip_mview_d* vsip_mputattrib_d(vsip_mview_d* v, const vsip_mattr_d* attr);
vsip_cmview_f* vsip_cputattrib_f(vsip_cmview_f* v, const vsip_cmattr_f* attr);
vsip_cmview_d* vsip_cputattrib_d(vsip_cmview_d* v, const vsip_cmattr_d* attr);
```

Description

This function modifies the attributes of an existing matrix view according to the parameters specified in a `vsip_dmattr_p` structure. It allows you to change the view's dimensions, memory layout, and binding to its data block in a single operation.

Parameters

- `vsip_dmview_p* v`: Pointer to the matrix view to be modified.
- `const vsip_dmattr_p* attr`: Pointer to the attribute structure containing the new attributes.

Return Value

- On success, returns a pointer to the modified matrix view.
- On error, returns `NULL`.

Example

```
vsip_mview_f *matrix;
vsip_mattr_f attr;
vsip_block_f *new_block;

// Create a matrix
matrix = vsip_mcreate_f(100, 100, VSIP_ROW, VSIP_MEM_NONE);

// Get current attributes
vsip_mgetattrib_f(matrix, &attr);
printf("Original dimensions: %lu x %lu\n", attr.row_length, attr.col_length);

// Modify the view to show only a submatrix
attr.row_length = 50;    // Show only first 50 rows
attr.col_length = 50;    // Show only first 50 columns
attr.offset = 0;         // Start from beginning of block
// Keep the same strides and block

if (vsip_mputattrib_f(matrix, &attr) == NULL) {
    // Handle error
}
```

Notes

- This function completely reconfigures the matrix view according to the provided attributes.

- The new configuration must be valid (e.g., the block must be large enough to contain the view with the specified offset and strides).
- The view's dimensions can be changed, but must be compatible with the block size and strides.
- The offset must be valid for the specified block.
- Strides must be positive and compatible with the block size and view dimensions.

1.4.15 vsip_dmgetblock_p - Get the Data Block from a Matrix View

```
vsip_block_f* vsip_mgetblock_f(const vsip_mview_f *v);  
vsip_block_d* vsip_mgetblock_d(const vsip_mview_d *v);  
vsip_cblock_f* vsip_cmgetblock_f(const vsip_cmview_f *v);  
vsip_cblock_d* vsip_cmgetblock_d(const vsip_cmview_d *v);
```

Description

This function returns a pointer to the data block associated with a matrix view. The data block contains the actual storage for the matrix elements.

Parameters

- `const vsip_dmview_p* v`: Pointer to the matrix view.

Return Value

- Returns a pointer to the `vsip_dblock_p` associated with the matrix view.

Notes

- Multiple matrix views can share the same data block.
- The block should not be destroyed directly if it's still being used by any matrix views.
- This function is useful for:
 - Creating additional views of the same data with different parameters
 - Checking if two matrices share the same underlying storage
 - Performing operations that require access to the raw data

1.4.16 vsip_dmgetcollength_p - Get Number of Columns in a Matrix View

```
vsip_length vsip_mgetcollength_f(const vsip_mview_f *v);  
vsip_length vsip_mgetcollength_d(const vsip_mview_d *v);  
vsip_length vsip_cmgetcollength_f(const vsip_cmview_f *v);  
vsip_length vsip_cmgetcollength_d(const vsip_cmview_d *v);
```

Description

This function returns the number of columns in a matrix view. The number of columns represents the size of the matrix in its second dimension (width) and determines how many elements are in each row of the matrix.

Parameters

- `const vsip_dmview_p* v`: Pointer to the matrix view.

Return Value

- Returns the number of columns in the matrix view as a `vsip_length` value.

1.4.17 vsip_dmputcollength_p - Set Number of Columns in a Matrix View

```
vsip_mview_f* vsip_mputcollength_f(const vsip_mview_f *v, vsip_length len);  
vsip_mview_d* vsip_mputcollength_d(const vsip_mview_d *v, vsip_length len);  
vsip_cmview_f* vsip_cputcollength_f(const vsip_cmview_f *v, vsip_length len);  
vsip_cmview_d* vsip_cputcollength_d(const vsip_cmview_d *v, vsip_length len);
```

Description

This function modifies the number of columns in an existing matrix view. It allows you to change the width of the matrix view while keeping all other attributes (row count, block, offset, and strides) the same.

Parameters

- `const vsip_dmview_p* v`: Pointer to the matrix view to be modified.
- `vsip_length len`: The new number of columns for the matrix view.

Return Value

- Returns a pointer to the modified matrix view.

Notes

- The new column length must be compatible with the matrix's block and strides:
 - The product of (new column length - 1) and column stride must not exceed the block size minus the offset
- Changing the column length affects how many elements are accessible in each row of the matrix view.
- For row-major matrices, this operation is generally safe as long as the new length doesn't exceed the block boundaries.
- For column-major matrices or matrices with non-unit column strides, be cautious as changing the column length might make the view invalid if it extends beyond the block boundaries.

1.4.18 vsip_dmgetrowlength_p - Get Number of Rows in a Matrix View

```
vsip_length vsip_mgetrowlength_f(const vsip_mview_f *v);  
vsip_length vsip_mgetrowlength_d(const vsip_mview_d *v);  
vsip_length vsip_cmgetrowlength_f(const vsip_cmview_f *v);  
vsip_length vsip_cmgetrowlength_d(const vsip_cmview_d *v);
```

Description

This function returns the number of rows in a matrix view. The number of rows represents the size of the matrix in its first dimension (height) and determines how many elements are in each column of the matrix.

Parameters

- `const vsip_dmview_p* v`: Pointer to the matrix view.

Return Value

- Returns the number of rows in the matrix view as a `vsip_length` value.

1.4.19 vsip_dmputrowlength_p - Set Number of Rows in a Matrix View

```
vsip_mview_f* vsip_mputrowlength_f(const vsip_mview_f *v, vsip_length len);  
vsip_mview_d* vsip_mputrowlength_d(const vsip_mview_d *v, vsip_length len);  
vsip_cmview_f* vsip_cputrowlength_f(const vsip_cmview_f *v, vsip_length len);  
vsip_cmview_d* vsip_cputrowlength_d(const vsip_cmview_d *v, vsip_length len);
```

Description

This function modifies the number of rows in an existing matrix view. It allows you to change the height of the matrix view while keeping all other attributes (column count, block, offset, and strides) the same.

Parameters

- `const vsip_dmview_p* v`: Pointer to the matrix view to be modified.
- `vsip_length len`: The new number of rows for the matrix view.

Return Value

- Returns a pointer to the modified matrix view.

Notes

- The new row length must be compatible with the matrix's block and strides:
 - The product of (new row length - 1) and row stride must not exceed the block size minus the offset
- Changing the row length affects how many elements are accessible in each column of the matrix view.
- For column-major matrices, this operation is generally safe as long as the new length doesn't exceed the block boundaries.
- For row-major matrices or matrices with non-unit row strides, be cautious as changing the row length might make the view invalid if it extends beyond the block boundaries.

1.4.20 vsip_dmgetcolstride_p - Get Column Stride of a Matrix View

```
vsip_stride vsip_mgetcolstride_f(const vsip_mview_f *v);  
vsip_stride vsip_mgetcolstride_d(const vsip_mview_d *v);  
vsip_stride vsip_cmgetcolstride_f(const vsip_cmview_f *v);  
vsip_stride vsip_cmgetcolstride_d(const vsip_cmview_d *v);
```

Description

This function returns the column stride of a matrix view, which represents the number of elements to skip in memory when moving from one column to the next within a row.

Parameters

- `const vsip_dmview_p* v`: Pointer to the matrix view.

Return Value

- Returns the column stride as a `vsip_stride` value.

Example

```
vsip_mview_f *matrix;  
vsip_stride col_stride;  
  
// Create a standard row-major matrix  
matrix = vsip_mcreate_f(100, 100, VSIP_ROW, VSIP_MEM_NONE);  
col_stride = vsip_mgetcolstride_f(matrix);  
printf("Standard matrix column stride: %ld\n", col_stride); // Typically 1
```

Notes

- For row-major matrices, the column stride is typically 1 (contiguous elements along rows).
- For column-major matrices, the column stride equals the number of rows.
- For transposed views, the column stride of the transposed view equals the row stride of the original matrix.

1.4.21 vsip_dmputcolstride_p - Set Column Stride of a Matrix View

```
vsip_mview_f* vsip_mputcolstride_f(const vsip_mview_f *v, vsip_stride stride);
vsip_mview_d* vsip_mputcolstride_d(const vsip_mview_d *v, vsip_stride stride);
vsip_cmview_f* vsip_cputcolstride_f(const vsip_cmview_f *v, vsip_stride stride);
vsip_cmview_d* vsip_cputcolstride_d(const vsip_cmview_d *v, vsip_stride stride);
```

Description

This function modifies the column stride of an existing matrix view. The column stride determines how elements are laid out in memory along the columns of the matrix (how many elements to skip when moving from one column to the next within a row).

Parameters

- `const vsip_dmview_p* v`: Pointer to the matrix view to be modified.
- `vsip_stride stride`: The new column stride value.

Return Value

- Returns a pointer to the modified matrix view.

Example

```
vsip_mview_f *matrix;
vsip_stride original_stride, new_stride;

// Create a matrix
matrix = vsip_mcreate_f(100, 100, VSIP_ROW, VSIP_MEM_NONE);
original_stride = vsip_mgetcolstride_f(matrix);
printf("Original column stride: %ld\n", original_stride); // Typically 1

// Change to stride of 2 (every other element in rows)
if (vsip_mputcolstride_f(matrix, 2) == NULL) {
    // Handle error
}
printf("New column stride: %ld\n", vsip_mgetcolstride_f(matrix)); // Output: 2
```

Notes

- The new stride must be compatible with the matrix dimensions and block size:
 - $(\text{column length} - 1) * \text{new stride} + \text{offset}$ must be $\leq$ block size
- Changing the column stride affects how elements are accessed when moving along rows.

1.4.22 vsip_dmgetrowstride_p - Get Row Stride of a Matrix View

```
vsip_stride vsip_mgetrowstride_f(const vsip_mview_f *v);  
vsip_stride vsip_mgetrowstride_d(const vsip_mview_d *v);  
vsip_stride vsip_cmgetrowstride_f(const vsip_cmview_f *v);  
vsip_stride vsip_cmgetrowstride_d(const vsip_cmview_d *v);
```

Description

This function returns the row stride of a matrix view, which represents the number of elements to skip in memory when moving from one row to the next within a column.

Parameters

- `const vsip_dmview_p* v`: Pointer to the matrix view.

Return Value

- Returns the row stride as a `vsip_stride` value.

Example

```
vsip_mview_f *matrix;  
vsip_stride row_stride;  
  
// Create a standard row-major matrix  
matrix = vsip_mcreate_f(100, 100, VSIP_ROW, VSIP_MEM_NONE);  
row_stride = vsip_mgetrowstride_f(matrix);  
printf("Standard matrix row stride: %ld\n", row_stride); // Typically 1
```

Notes

- For column-major matrices, the row stride is typically 1 (contiguous elements along columns).
- For row-major matrices, the row stride equals the number of columns.
- For transposed views, the row stride of the transposed view equals the column stride of the original matrix.

1.4.23 vsip_dmputrowstride_p - Set Row Stride of a Matrix View

```
vsip_mview_f* vsip_mputrowstride_f(const vsip_mview_f *v, vsip_stride stride);
vsip_mview_d* vsip_mputrowstride_d(const vsip_mview_d *v, vsip_stride stride);
vsip_cmview_f* vsip_cputrowstride_f(const vsip_cmview_f *v, vsip_stride stride);
vsip_cmview_d* vsip_cputrowstride_d(const vsip_cmview_d *v, vsip_stride stride);
```

Description

This function modifies the row stride of an existing matrix view. The row stride determines how elements are laid out in memory along the rows of the matrix (how many elements to skip when moving from one row to the next within a column).

Parameters

- `const vsip_dmview_p* v`: Pointer to the matrix view to be modified.
- `vsip_stride stride`: The new row stride value.

Return Value

- Returns a pointer to the modified matrix view.

Example

```
vsip_mview_f *matrix;
vsip_stride original_stride, new_stride;

// Create a matrix
matrix = vsip_mcreate_f(100, 100, VSIP_ROW, VSIP_MEM_NONE);
original_stride = vsip_mgetrowstride_f(matrix);
printf("Original row stride: %ld\n", original_stride); // Typically 1

// Change to stride of 2 (every other element in rows)
if (vsip_mputrowstride_f(matrix, 2) == NULL) {
    // Handle error
}
printf("New row stride: %ld\n", vsip_mgetcolstride_f(matrix)); // Output: 2
```

Notes

- The new stride must be compatible with the matrix dimensions and block size:
 - $(\text{row length} - 1) * \text{new stride} + \text{offset}$ must be $\leq$ block size
- Changing the row stride affects how elements are accessed when moving along columns.

1.4.24 vsip_dmgetoffset_p - Get Matrix View Offset

```
vsip_offset vsip_mgetoffset_f(const vsip_mview_f *v);  
vsip_offset vsip_mgetoffset_d(const vsip_mview_d *v);  
vsip_offset vsip_cmgetoffset_f(const vsip_cmview_f *v);  
vsip_offset vsip_cmgetoffset_d(const vsip_cmview_d *v);
```

Description

This function returns the offset of a matrix view within its associated data block. The offset represents the number of elements from the start of the block to the first element (0,0) of the matrix view.

Parameters

- `const vsip_dmview_p* v`: Pointer to the matrix view.

Return Value

- Returns the offset in elements from the start of the block to the first element of the matrix view.

Notes

- For matrices created with `vsip_mcreate_p`, the offset is typically 0.
- The offset, combined with the strides, completely defines where the matrix view is located within its block.

1.4.25 vsip_dmputoffset_p - Set Matrix View Offset

```
vsip_mview_f* vsip_mputoffset_f(const vsip_mview_f *v, vsip_offset off);
vsip_mview_d* vsip_mputoffset_d(const vsip_mview_d *v, vsip_offset off);
vsip_cmview_f* vsip_cmputoffset_f(const vsip_cmview_f *v, vsip_offset off);
vsip_cmview_d* vsip_cmputoffset_d(const vsip_cmview_d *v, vsip_offset off);
```

Description

This function modifies the offset of an existing matrix view. The offset determines where the matrix view begins within its associated data block, measured in elements from the start of the block.

Parameters

- `const vsip_dmview_p* v`: Pointer to the matrix view to be modified.
- `vsip_offset off`: The new offset in elements from the start of the block.

Return Value

- Returns a pointer to the modified matrix view.

Notes

- The new offset must be such that the entire view fits within the block boundaries.
- The condition for a valid offset is:

$$\text{offset} + \text{row_length} \times \text{row_stride} + \text{col_length} \times \text{col_stride} < \text{block_size}$$

1.4.26 vsip_dmdestroy_p - Destroy a Matrix View

```
vsip_block_f* vsip_mdestroy_f(vsip_mview_f *matrix);  
vsip_block_d* vsip_mdestroy_d(vsip_mview_d *matrix);  
vsip_cblock_f* vsip_cmdestroy_f(vsip_cmview_f *matrix);  
vsip_cblock_d* vsip_cmdestroy_d(vsip_cmview_d *matrix);
```

Description

This function destroys a matrix view and returns its associated data block.

Parameters

- vsip_dmview_p* matrix: Pointer to the matrix view to be destroyed.

Example

```
vsip_mview_f *matrix;  
  
// Create a matrix  
matrix = vsip_mcreate_f(100, 100, VSIP_ROW, VSIP_MEM_NONE);  
  
// Use the matrix...  
// ...  
  
// Destroy the matrix when no longer needed  
vsip_mdestroy_f(matrix);
```

1.4.27 vsip_dmalldestroy_p - Destroy Matrix View and its Data Block

```
void vsip_malldestroy_f(vsip_mview_f *matrix);  
void vsip_malldestroy_d(vsip_mview_d *matrix);  
void vsip_cmalldestroy_f(vsip_cmview_f *matrix);  
void vsip_cmalldestroy_d(vsip_cmview_d *matrix);
```

Description

This function destroys a matrix view and its associated data block. If the view is from a derived block such as a complex block, the complex block must be destroyed in a separate manner to free up the memory.

Parameters

- `vsip_dmview_p * matrix`: Pointer to a matrix view object to be destroyed.

Chapter 2

Scalar Functions

2.1 Complex Scalar Functions

2.1.1 vsip_arg_p - Compute Phase Angle of Complex Scalar

```
vsip_scalar_f vsip_arg_f(vsip_cscalar_f x);  
vsip_scalar_d vsip_arg_d(vsip_cscalar_d x);
```

Description

This function computes the phase angle (argument) of a complex scalar value. The phase angle θ is computed as:

$$\theta = \text{atan2}(\text{imag}(x), \text{real}(x))$$

where `atan2` is the two-argument arctangent function that correctly handles all quadrants of the complex plane. The returned phase angle is in the range $[-\pi, \pi]$ radians.

Parameters

- `vsip_cscalar_p x`: Input complex scalar value.

Return Value

- Returns the phase angle in radians, in the range $[-\pi, \pi]$.

2.1.2 vsip_cmag_p - Compute Magnitude of Complex Scalar

```
vsip_scalar_f vsip_cmag_f(vsip_cscalar_f x);  
vsip_scalar_d vsip_cmag_d(vsip_cscalar_d x);
```

Description

This function computes the magnitude (absolute value) of a complex scalar. The magnitude is computed as:

$$\text{magnitude} = \sqrt{\text{real}(x)^2 + \text{imag}(x)^2}$$

This function provides a numerically stable implementation that avoids potential overflow and underflow issues.

Parameters

- `vsip_cscalar_p x`: Input complex scalar value.

Return Value

- Returns the magnitude (absolute value) of the complex scalar.

2.1.3 vsip_cmagsq_p - Compute Magnitude Squared of Complex Scalar

```
vsip_scalar_f vsip_cmagsq_f(vsip_cscalar_f x);  
vsip_scalar_d vsip_cmagsq_d(vsip_cscalar_d x);
```

Description

This function computes the squared magnitude of a complex scalar. The squared magnitude is computed as:

$$\text{magnitude_squared} = \text{real}(x)^2 + \text{imag}(x)^2$$

This function is often preferred over `vsip_cmag_p` if only magnitudes are to be compared.

Parameters

- `vsip_cscalar_p x`: Input complex scalar value.

Return Value

- Returns the squared magnitude of the complex scalar.

2.1.4 vsip_conj_p - Compute Complex Conjugate

```
vsip_cscalar_f vsip_conj_f(vsip_cscalar_f x);  
vsip_cscalar_d vsip_conj_d(vsip_cscalar_d x);
```

Description

This function computes the complex conjugate of a complex scalar value. The complex conjugate is computed as:

$$\text{conj}(x) = \text{real}(x) - j \cdot \text{imag}(x)$$

where x is the input complex scalar.

Parameters

- `vsip_cscalar_p x`: Input complex scalar value.

Return Value

- Returns the complex conjugate of the input scalar.

2.1.5 vsip_polar_p - Convert Cartesian to Polar Coordinates

```
void vsip_polar_f(vsip_cscalar_f a, vsip_scalar_f *radius, vsip_scalar_f *theta);  
void vsip_polar_d(vsip_cscalar_d a, vsip_scalar_d *radius, vsip_scalar_d *theta);
```

Description

This function converts a complex scalar from Cartesian coordinates (real and imaginary parts) to polar coordinates (magnitude and phase angle).

The function computes both the magnitude (radius) and phase angle (theta) of the complex number:

$$\text{radius} = |a| = \sqrt{\text{real}(a)^2 + \text{imag}(a)^2}$$

$$\text{theta} = \text{atan2}(\text{imag}(a), \text{real}(a))$$

The phase angle is returned in radians in the range $[-\pi, \pi]$.

Parameters

- vsip_cscalar_p a: Input complex scalar in Cartesian coordinates.
- vsip_scalar_p *radius: Pointer to store the magnitude (radius).
- vsip_scalar_p *theta: Pointer to store the phase angle in radians.

2.1.6 vsip_rect_p - Convert Polar to Cartesian Coordinates

```
vsip_cscalar_f vsip_rect_f(vsip_scalar_f radius, vsip_scalar_f theta);  
vsip_cscalar_d vsip_rect_d(vsip_scalar_d radius, vsip_scalar_d theta);
```

Description

This function converts polar coordinates (magnitude and phase angle) to a complex scalar in Cartesian coordinates (real and imaginary parts). The function computes the Cartesian coordinates from polar coordinates using Euler's formula:

$$\text{real} = \text{radius} \cdot \cos(\text{theta})$$

$$\text{imag} = \text{radius} \cdot \sin(\text{theta})$$

The phase angle should be provided in radians.

Parameters

- `vsip_scalar_p radius`: Magnitude (radius) of the complex number.
- `vsip_scalar_p theta`: Phase angle in radians.

Return Value

- Returns the complex scalar in Cartesian coordinates.

2.1.7 vsip_real_p - Complex Real part

```
vsip_scalar_f vsip_real_f(vsip_cscalar_f x);  
vsip_scalar_d vsip_real_d(vsip_cscalar_d x);
```

Description

This function extracts the real part of the complex scalar x.

Parameters

- vsip_cscalar_f x: The complex scalar from which to extract the real part.

Return Value

- The real part of the complex scalar.

Example

```
vsip_cscalar_f complex_value = {1.0, 2.0};  
vsip_scalar_f real_part;  
  
real_part = vsip_real_f(complex_value);
```

2.1.8 vsip_imag_p - Complex Imaginary part

```
vsip_scalar_f vsip_imag_f(vsip_cscalar_f x);  
vsip_scalar_d vsip_imag_d(vsip_cscalar_d x);
```

Description

This function extracts the imaginary part of the complex scalar x.

Parameters

- vsip_cscalar_f x: The complex scalar from which to extract the imaginary part.

Return Value

- The imaginary part of the complex scalar.

Example

```
vsip_cscalar_f complex_value = {1.0, 2.0};  
vsip_scalar_f imag_part;  
  
imag_part = vsip_imag_f(complex_value);
```

2.1.9 vsip_cmplx_p - Create complex number

```
vsip_cscalar_f vsip_cmplx_f(vsip_scalar_f r, vsip_scalar_f i);  
vsip_cscalar_d vsip_cmplx_d(vsip_scalar_d r, vsip_scalar_d i);
```

Description

This function creates a complex scalar from the real part `r` and the imaginary part `i`.

Parameters

- `vsip_scalar_f r`: The real part of the complex scalar.
- `vsip_scalar_f i`: The imaginary part of the complex scalar.

Return Value

- The created complex scalar.

Example

```
vsip_scalar_f real_part = 1.0;  
vsip_scalar_f imag_part = 2.0;  
vsip_cscalar_f complex_value;  
  
complex_value = vsip_cmplx_f(real_part, imag_part);
```

2.1.10 vsip_cadd_p - Complex Scalar Addition

```
vsip_cscalar_f vsip_cadd_f(vsip_cscalar_f x, vsip_cscalar_f y);  
vsip_cscalar_d vsip_cadd_d(vsip_cscalar_d x, vsip_cscalar_d y);
```

Description

This function performs complex scalar addition.

The operation performs:

$$\text{result} = x + y$$

Parameters

- vsip_cscalar_f a: The first complex scalar value.
- vsip_cscalar_f b: The second complex scalar value.

Return Value

- The complex scalar value result.

2.1.11 vsip_csub_p - Complex Scalar Subtraction

```
vsip_cscalar_f vsip_csub_f(vsip_cscalar_f x, vsip_cscalar_f y);  
vsip_cscalar_d vsip_csub_d(vsip_cscalar_d x, vsip_cscalar_d y);
```

Description

This function performs complex scalar subtraction.

The operation performs:

$$\text{result} = x - y$$

Parameters

- vsip_cscalar_f a: The first complex scalar value.
- vsip_cscalar_f b: The second complex scalar value.

Return Value

- The complex scalar value result.

2.1.12 vsip_cmul_p - Complex Scalar Multiplication

```
vsip_cscalar_f vsip_cmul_f(vsip_cscalar_f x, vsip_cscalar_f y);  
vsip_cscalar_d vsip_cmul_d(vsip_cscalar_d x, vsip_cscalar_d y);
```

Description

This function performs complex scalar multiplication.

The operation performs:

$$\text{result} = xy$$

Parameters

- vsip_cscalar_f a: The first complex scalar value.
- vsip_cscalar_f b: The second complex scalar value.

Return Value

- The complex scalar value result.

2.1.13 vsip_cjmul_p - Complex Scalar Conjugate Multiplication

```
vsip_cscalar_f vsip_cjmul_f(vsip_cscalar_f x, vsip_cscalar_f y);  
vsip_cscalar_d vsip_cjmul_d(vsip_cscalar_d x, vsip_cscalar_d y);
```

Description

This function performs complex scalar conjugate multiplication.

The operation performs:

$$\text{result} = x\bar{y}$$

Parameters

- vsip_cscalar_f a: The first complex scalar value.
- vsip_cscalar_f b: The second complex scalar value.

Return Value

- The complex scalar value result.

2.1.14 vsip_cdiv_p - Complex Scalar Division

```
vsip_cscalar_f vsip_cdiv_f(vsip_cscalar_f x, vsip_cscalar_f y);  
vsip_cscalar_d vsip_cdiv_d(vsip_cscalar_d x, vsip_cscalar_d y);
```

Description

This function performs complex scalar division.

The operation performs:

$$\text{result} = \frac{x}{y}$$

Parameters

- vsip_cscalar_f a: The first complex scalar value.
- vsip_cscalar_f b: The second complex scalar value.

Return Value

- The complex scalar value result.

2.1.15 vsip_rcadd_p - Real-Complex Scalar Addition

```
vsip_cscalar_f vsip_rcadd_f(vsip_scalar_f a, vsip_cscalar_f b);  
vsip_cscalar_d vsip_rcadd_d(vsip_scalar_d a, vsip_cscalar_d b);
```

Description

This function performs addition between a real scalar and a complex scalar.

The operation performs:

$$\text{result} = a + b$$

Parameters

- vsip_scalar_f a: The real scalar value.
- vsip_cscalar_f b: The complex scalar value.

Return Value

- The complex scalar value result.

2.1.16 vsip_rcsub_p - Real-Complex Scalar Subtraction

```
vsip_cscalar_f vsip_rcsub_f(vsip_scalar_f a, vsip_cscalar_f b);  
vsip_cscalar_d vsip_rcsub_d(vsip_scalar_d a, vsip_cscalar_d b);
```

Description

This function performs subtraction between a real scalar and a complex scalar.

The operation performs:

$$\text{result} = a - b$$

Parameters

- vsip_scalar_f a: The real scalar value.
- vsip_cscalar_f b: The complex scalar value.

Return Value

- The complex scalar value result.

2.1.17 vsip_rcmul_p - Real-Complex Scalar Multiplication

```
vsip_cscalar_f vsip_rcmul_f(vsip_scalar_f a, vsip_cscalar_f b);  
vsip_cscalar_d vsip_rcmul_d(vsip_scalar_d a, vsip_cscalar_d b);
```

Description

This function performs multiplication between a real scalar and a complex scalar.

The operation performs:

$$\text{result} = ab$$

Parameters

- vsip_scalar_f a: The real scalar value.
- vsip_cscalar_f b: The complex scalar value.

Return Value

- The complex scalar value result.

2.1.18 vsip_crsub_p - Complex-Real Scalar Subtraction

```
vsip_cscalar_f vsip_crsub_f(vsip_cscalar_f a, vsip_scalar_f b);  
vsip_cscalar_d vsip_crsub_d(vsip_cscalar_d a, vsip_scalar_d b);
```

Description

This function performs subtraction between a complex scalar and a real scalar.

The operation performs:

$$\text{result} = a - b$$

Parameters

- vsip_cscalar_f a: The complex scalar value.
- vsip_scalar_f b: The real scalar value.

Return Value

- The complex scalar value result.

2.1.19 vsip_crdiv_p - Complex-Real Scalar Division

```
vsip_cscalar_f vsip_crdiv_f(vsip_cscalar_f a, vsip_scalar_f b);  
vsip_cscalar_d vsip_crdiv_d(vsip_cscalar_d a, vsip_scalar_d b);
```

Description

This function performs division between a complex scalar and a real scalar.

The operation performs:

$$\text{result} = \frac{a}{b}$$

Parameters

- vsip_cscalar_f a: The complex scalar value.
- vsip_scalar_f b: The real scalar value.

Return Value

- The complex scalar value result.

2.1.20 vsip_cneg_p - Complex Scalar Negation

```
vsip_cscalar_f vsip_cneg_f(vsip_cscalar_f x);  
vsip_cscalar_d vsip_cneg_d(vsip_cscalar_d x);
```

Description

This function performs negation of a complex scalar. The operation performs:

$$\text{result} = -x$$

Parameters

- vsip_cscalar_f a: The complex scalar value.

Return Value

- The complex scalar value result.

2.1.21 vsip_crecip_p - Complex Scalar Reciprocal

```
vsip_cscalar_f vsip_crecip_f(vsip_cscalar_f x);  
vsip_cscalar_d vsip_crecip_d(vsip_cscalar_d x);
```

Description

This function performs a reciprocal of a complex scalar. The operation performs:

$$\text{result} = \frac{1}{x}$$

Parameters

- vsip_cscalar_f a: The complex scalar value.

Return Value

- The complex scalar value result.

2.1.22 vsip_csqrt_p - Complex Scalar Square Root

```
vsip_cscalar_f vsip_csqrt_f(vsip_cscalar_f x);  
vsip_cscalar_d vsip_csqrt_d(vsip_cscalar_d x);
```

Description

This function performs the square root of a complex scalar. The operation performs:

$$\text{result} = \sqrt{x}$$

Parameters

- vsip_cscalar_f a: The complex scalar value.

Return Value

- The complex scalar value result.

2.1.23 vsip_cexp_p - Complex Scalar Exponential

```
vsip_cscalar_f vsip_cexp_f(vsip_cscalar_f x);  
vsip_cscalar_d vsip_cexp_d(vsip_cscalar_d x);
```

Description

This function performs the exponentiation of a complex scalar. The operation performs:

$$\text{result} = e^x$$

Parameters

- vsip_cscalar_f a: The complex scalar value.

Return Value

- The complex scalar value result.

2.1.24 vsip_CONJ_p - Complex Scalar Conjugate and Store to Pointer

```
void vsip_CONJ_f(vsip_cscalar_f x, vsip_cscalar_f *r);  
void vsip_CONJ_d(vsip_cscalar_d x, vsip_cscalar_d *r);
```

Description

This function computes the complex conjugate of a complex scalar storing the result in the output parameter.

The operation performs:

$$*r = \text{conj}(x)$$

Parameters

- `vsip_scalar_p x`: The complex scalar.
- `vsip_cscalar_p * r`: Pointer to the complex scalar where the result will be stored.

2.1.25 vsip_CMPLX_p - Create a Complex Scalar and Store in a Pointer

```
void vsip_CMPLX_f(vsip_scalar_f a, vsip_scalar_f b, vsip_cscalar_f *r);  
void vsip_CMPLX_d(vsip_scalar_d a, vsip_scalar_d b, vsip_cscalar_d *r);
```

Description

This function creates a complex scalar from the real part a and the imaginary part b and stores the result in the complex scalar pointed to by r.

Parameters

- vsip_scalar_p a: The real part of the complex scalar.
- vsip_scalar_p b: The imaginary part of the complex scalar.
- vsip_cscalar_p * r: Pointer to the complex scalar where the result will be stored.

Example

```
vsip_scalar_f real_part = 1.0;  
vsip_scalar_f imag_part = 2.0;  
vsip_cscalar_f complex_value;  
  
vsip_CMPLX_f(real_part, imag_part, &complex_value);
```

2.1.26 vsip_CADD_p - Complex Scalar Addition and Store to Pointer

```
void vsip_CADD_f(vsip_cscalar_f a, vsip_cscalar_f b, vsip_cscalar_f *r);  
void vsip_CADD_d(vsip_cscalar_d a, vsip_cscalar_d b, vsip_cscalar_d *r);
```

Description

This function performs complex scalar addition of two complex scalar values and storing the result in the output pointer.

The operation performs:

$$*r = a + b$$

Parameters

- vsip_cscalar_p a: First complex scalar number.
- vsip_cscalar_p b: Second complex scalar number.
- vsip_cscalar_p * r: Pointer to the complex scalar where the result will be stored.

2.1.27 vsip_CSUB_p - Complex Scalar Subtraction and Store to Pointer

```
void vsip_CSUB_f(vsip_cscalar_f a, vsip_cscalar_f b, vsip_cscalar_f *r);  
void vsip_CSUB_d(vsip_cscalar_d a, vsip_cscalar_d b, vsip_cscalar_d *r);
```

Description

This function performs complex scalar subtraction of two complex scalar values and storing the result in the output pointer.

The operation performs:

$$*r = a - b$$

Parameters

- vsip_cscalar_p a: First complex scalar number.
- vsip_cscalar_p b: Second complex scalar number.
- vsip_cscalar_p * r: Pointer to the complex scalar where the result will be stored.

2.1.28 vsip_CMUL_p - Complex Scalar Multiplication and Store to Pointer

```
void vsip_CMUL_f(vsip_cscalar_f a, vsip_cscalar_f b, vsip_cscalar_f *r);  
void vsip_CMUL_d(vsip_cscalar_d a, vsip_cscalar_d b, vsip_cscalar_d *r);
```

Description

This function performs complex scalar multiplication of two complex scalar values and storing the result in the output pointer.

The operation performs:

$$*r = ab$$

Parameters

- vsip_cscalar_p a: First complex scalar number.
- vsip_cscalar_p b: Second complex scalar number.
- vsip_cscalar_p * r: Pointer to the complex scalar where the result will be stored.

2.1.29 vsip_CJMUL_p - Complex Scalar Conjugate Multiplication and Store to Pointer

```
void vsip_CJMUL_f(vsip_cscalar_f a, vsip_cscalar_f b, vsip_cscalar_f *r);  
void vsip_CJMUL_d(vsip_cscalar_d a, vsip_cscalar_d b, vsip_cscalar_d *r);
```

Description

This function performs complex scalar conjugate multiplication of two complex scalar values and storing the result in the output pointer.

The operation performs:

$$*r = a\bar{b}$$

Parameters

- vsip_cscalar_p a: First complex scalar number.
- vsip_cscalar_p b: Second complex scalar number.
- vsip_cscalar_p * r: Pointer to the complex scalar where the result will be stored.

2.1.30 vsip_CDIV_p - Complex Scalar Division and Store to Pointer

```
void vsip_CDIV_f(vsip_cscalar_f a, vsip_cscalar_f b, vsip_cscalar_f *r);  
void vsip_CDIV_d(vsip_cscalar_d a, vsip_cscalar_d b, vsip_cscalar_d *r);
```

Description

This function performs complex scalar division of two complex scalar values storing the result in the output pointer.

The operation performs:

$$*r = \frac{a}{b}$$

Parameters

- vsip_cscalar_p a: First complex scalar number.
- vsip_cscalar_p b: Second complex scalar number.
- vsip_cscalar_p * r: Pointer to the complex scalar where the result will be stored.

2.1.31 vsip_RCADD_p - Real-Complex Scalar Addition and Store to Pointer

```
void vsip_RCADD_f(vsip_scalar_f a, vsip_cscalar_f b, vsip_cscalar_f *r);  
void vsip_RCADD_d(vsip_scalar_d a, vsip_cscalar_d b, vsip_cscalar_d *r);
```

Description

This function performs an addition between a real scalar and a complex scalar storing the result in the output pointer.

The operation performs:

$$*r = a + b$$

Parameters

- vsip_scalar_p a: Real scalar number.
- vsip_cscalar_p b: Complex scalar number.
- vsip_cscalar_p * r: Pointer to the complex scalar where the result will be stored.

2.1.32 vsip_RCSUB_p - Real-Complex Scalar Subtraction and Store to Pointer

```
void vsip_RCSUB_f(vsip_scalar_f a, vsip_cscalar_f b, vsip_cscalar_f *r);  
void vsip_RCSUB_d(vsip_scalar_d a, vsip_cscalar_d b, vsip_cscalar_d *r);
```

Description

This function performs an subtraction between a real scalar and a complex scalar storing the result in the output pointer.

The operation performs:

$$*r = a - b$$

Parameters

- vsip_scalar_p a: Real scalar number.
- vsip_cscalar_p b: Complex scalar number.
- vsip_cscalar_p * r: Pointer to the complex scalar where the result will be stored.

2.1.33 vsip_RCMUL_p - Real-Complex Scalar Multiplication and Store to Pointer

```
void vsip_RCMUL_f(vsip_scalar_f a, vsip_cscalar_f b, vsip_cscalar_f *r);  
void vsip_RCMUL_d(vsip_scalar_d a, vsip_cscalar_d b, vsip_cscalar_d *r);
```

Description

This function performs an subtraction between a real scalar and a complex scalar storing the result in the output pointer.

The operation performs:

$$*r = ab$$

Parameters

- vsip_scalar_p a: Real scalar number.
- vsip_cscalar_p b: Complex scalar number.
- vsip_cscalar_p * r: Pointer to the complex scalar where the result will be stored.

2.1.34 vsip_CRSUB_p - Complex-Real Scalar Subtraction and Store to Pointer

```
void vsip_CRSUB_f(vsip_cscalar_f a, vsip_scalar_f b, vsip_cscalar_f *r);  
void vsip_CRSUB_d(vsip_cscalar_d a, vsip_scalar_d b, vsip_cscalar_d *r);
```

Description

This function performs subtraction between a complex scalar and a real scalar storing the result in the output pointer.

The operation performs:

$$*r = a - b$$

Parameters

- vsip_cscalar_p a: Complex scalar number.
- vsip_scalar_p b: Real scalar number.
- vsip_cscalar_p * r: Pointer to the complex scalar where the result will be stored.

2.1.35 vsip_CRDIV_p - Complex-Real Scalar Division and Store to Pointer

```
void vsip_CRDIV_f(vsip_cscalar_f a, vsip_scalar_f b, vsip_cscalar_f *r);  
void vsip_CRDIV_d(vsip_cscalar_d a, vsip_scalar_d b, vsip_cscalar_d *r);
```

Description

This function performs division between a complex scalar and a real scalar storing the result in the output pointer.

The operation performs:

$$*r = \frac{a}{b}$$

Parameters

- vsip_cscalar_p a: Complex scalar number.
- vsip_scalar_p b: Real scalar number.
- vsip_cscalar_p * r: Pointer to the complex scalar where the result will be stored.

2.1.36 vsip_CNEG_p - Complex Scalar Negate and Store to Pointer

```
void vsip_CNEG_f(vsip_cscalar_f x, vsip_cscalar_f *r);  
void vsip_CNEG_d(vsip_cscalar_d x, vsip_cscalar_d *r);
```

Description

This function computes the complex negate of a complex scalar storing the result in the output parameter.

The operation performs:

$$*r = -x$$

Parameters

- vsip_scalar_p x: The complex scalar.
- vsip_cscalar_p * r: Pointer to the complex scalar where the result will be stored.

2.1.37 vsip_CRECIp_p - Complex Scalar Reciprocal and Store to Pointer

```
void vsip_CRECIp_f(vsip_cscalar_f x, vsip_cscalar_f *r);  
void vsip_CRECIp_d(vsip_cscalar_d x, vsip_cscalar_d *r);
```

Description

This function computes the complex reciprocal of a complex scalar storing the result in the output parameter.

The operation performs:

$$*r = \frac{1}{x}$$

Parameters

- vsip_scalar_p x: The complex scalar.
- vsip_cscalar_p * r: Pointer to the complex scalar where the result will be stored.

2.1.38 vsip_CSQRT_p - Complex Scalar Square Root and Store to Pointer

```
void vsip_CSQRT_f(vsip_cscalar_f x, vsip_cscalar_f *r);  
void vsip_CSQRT_d(vsip_cscalar_d x, vsip_cscalar_d *r);
```

Description

This function computes the complex square root of a complex scalar storing the result in the output parameter.

The operation performs:

$$*r = \sqrt{x}$$

Parameters

- vsip_scalar_p x: The complex scalar.
- vsip_cscalar_p * r: Pointer to the complex scalar where the result will be stored.

2.1.39 vsip_CEXP_p - Complex Scalar Exponential and Store to Pointer

```
void vsip_CEXP_f(vsip_cscalar_f x, vsip_cscalar_f *r);  
void vsip_CEXP_d(vsip_cscalar_d x, vsip_cscalar_d *r);
```

Description

This function computes the complex exponential of a complex scalar storing the result in the output parameter.

The operation performs:

$$*r = e^x$$

Parameters

- `vsip_scalar_p x`: The complex scalar.
- `vsip_cscalar_p * r`: Pointer to the complex scalar where the result will be stored.

2.2 Index Scalar Functions

2.2.1 vsip_matindex - Convert Row/Column Indices to Matrix Index

```
vsip_scalar_mi vsip_matindex(vsip_scalar_vi row, vsip_scalar_vi col);
```

Description

This function converts row and column indices to a matrix index.

Parameters

- `vsip_scalar_vi row`: Row index (0-based).
- `vsip_scalar_vi col`: Column index (0-based).

Return Value

- Returns the matrix index.

2.2.2 vsip_mrowindex - Extract Row Index from Matrix Index

```
vsip_scalar_vi vsip_mrowindex(vsip_scalar_mi index);
```

Description

This function extracts the row index from the matrix index. It effectively performs the inverse operation of `vsip_matindex` for the row component.

Parameters

- `vsip_scalar_mi index`: Matrix index (as created by `vsip_matindex`).

Return Value

- Returns the row index (0-based) corresponding to the input matrix index.

2.2.3 vsip_mcolindex - Extract Column Index from Matrix Index

```
vsip_scalar_vi vsip_mcolindex(vsip_scalar_mi index);
```

Description

This function extracts the column index from a matrix index. It effectively performs the inverse operation of `vsip_matindex` for the column component.

Parameters

- `vsip_scalar_mi index`: Linearized matrix index (as created by `vsip_matindex`).

Return Value

- Returns the column index (0-based) corresponding to the input matrix index.

2.2.4 vsip_MATINDEX - Convert Row/Column Indices to Matrix Index and Store in a Pointer

```
void vsip_MATINDEX(vsip_scalar_vi row, vsip_scalar_vi col, vsip_scalar_mi *r);
```

Description

This function converts row and column indices to a single linear matrix index, storing the result in the provided output parameter.

Unlike the function version, `vsip_matindex`, this macro stores the result in the provided output parameter rather than returning it directly.

Parameters

- `vsip_scalar_vi row`: Row index (0-based).
- `vsip_scalar_vi col`: Column index (0-based).
- `vsip_scalar_mi *r`: Pointer to store the resulting matrix index.

Chapter 3

Random Number Generation

3.1 Random Number Functions

3.1.1 vsip_randcreate - Create a Random Number Generator State

```
vsip_randstate *vsip_randcreate(vsip_index seed, vsip_index numprocs,  
                                vsip_index id, vsip_rng portable);
```

Description

This function creates and initializes a random number generator state. The function allows for parallel random number generation by specifying the number of processes (numprocs) and the process ID (id). The portable parameter specifies the type of random number generator to use.

Parameters

- vsip_index seed: The seed value for the random number generator.
- vsip_index numprocs: The number of parallel processes.
- vsip_index id: The ID of the current process (must be in the range [0, numprocs-1]).
- vsip_rng portable: The type of random number generator to use (e.g., VSIP_PRNG for portable random number generation).

Return Value

- On success, a pointer to the newly created random number generator state is returned.
- On error, NULL is returned.

Example

```
vsip_randstate *rand_state;  
vsip_index seed = 42;  
vsip_index numprocs = 1;  
vsip_index id = 0;  
  
// Create a random number generator state  
rand_state = vsip_randcreate(seed, numprocs, id, VSIP_PRNG);  
  
if (rand_state == NULL) {  
    // Handle error  
}
```

3.1.2 vsip_randdestroy - Destroy a Random Number Generator State

```
int vsip_randdestroy(vsip_randstate *state);
```

Description

This function destroys the random number generator state `state` and frees all associated resources. After calling this function, the random number generator state should no longer be used.

Parameters

- `vsip_randstate* state`: Pointer to the random number generator state to be destroyed.

Return Value

- Returns 0 on success.
- Returns a non-zero value on error.

Example

```
vsip_randstate *rand_state;
int result;

// Assuming rand_state has been properly initialized
result = vsip_randdestroy(rand_state);

if (result != 0) {
    // Handle error
}
```

3.1.3 vsip_dvrandu_p - Generate Uniformly Distributed Random Numbers in a Vector View

```
void vsip_vrandu_f(vsip_randstate *state, const vsip_vview_f *r);
void vsip_vrandu_d(vsip_randstate *state, const vsip_vview_d *r);
void vsip_cvrandu_f(vsip_randstate *state, const vsip_cvview_f *r);
void vsip_cvrandu_d(vsip_randstate *state, const vsip_cvview_d *r);
```

Description

This function fills the vector view `r` with uniformly distributed random numbers in the range `[0, 1)` using the random number generator state `state`.

Parameters

- `vsip_randstate* state`: Pointer to the random number generator state.
- `const vsip_dvview_p* r`: Pointer to the destination vector view where the random numbers will be stored.

Example

```
vsip_randstate *rand_state;
vsip_vview_f *random_vector;

// Initialize random number generator state
rand_state = vsip_randcreate(42, 0, 1, VSIP_PRNG);

// Assuming random_vector has been properly initialized
vsip_vrandu_f(rand_state, random_vector);

// Clean up
vsip_randdestroy(rand_state);
```

3.1.4 vsip_dvrandn_p - Fill Vector with Normally Distributed Random Numbers

```
void vsip_vrandn_f(vsip_randstate *state, const vsip_vview_f *r);
void vsip_vrandn_d(vsip_randstate *state, const vsip_vview_d *r);
void vsip_cvrandn_f(vsip_randstate *state, const vsip_cvview_f *r);
void vsip_cvrandn_d(vsip_randstate *state, const vsip_cvview_d *r);
```

Description

This function fills a vector with random numbers drawn from a standard normal distribution (mean = 0, standard deviation = 1) using the specified random number generator state. The random numbers are generated according to the normal (Gaussian) probability density function:

$$f(x) = \frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}}$$

Parameters

- vsip_randstate* state: Pointer to the random number generator state.
- const vsip_dvview_p* r: Pointer to the output vector that will be filled with normally distributed random numbers.

Example

```
vsip_randstate *rand_state;
vsip_vview_f *random_vector;

// Initialize random number generator state
rand_state = vsip_randcreate(42, 0, 1, VSIP_PRNG);

// Assuming random_vector has been properly initialized
vsip_vrandn_f(rand_state, random_vector);

// Clean up
vsip_randdestroy(rand_state);
```

Chapter 4

Vector and Elementwise Operations

4.1 Elementary Math Operations

4.1.1 vsip_vsin_p - Element-wise Sine of a Vector View

```
void vsip_vsin_f(const vsip_vview_f* a, const vsip_vview_f* r);  
void vsip_vsin_d(const vsip_vview_d* a, const vsip_vview_d* r);
```

Description

This function computes the element-wise sine of the vector view `a` and stores the result in the vector view `r`.

$$r_i = \sin(a_i)$$

Parameters

- `const vsip_vview_p* a`: Pointer to the source vector view.
- `const vsip_vview_p* r`: Pointer to the destination vector view.

Example

```
vsip_vview_f *src_vector_view;  
vsip_vview_f *dst_vector_view;  
  
// Assuming src_vector_view and dst_vector_view have been properly initialized  
vsip_vsin_f(src_vector_view, dst_vector_view);
```

4.1.2 vsip_vasin_p - Element-wise Arcsine of a Vector View

```
void vsip_vasin_f(const vsip_vview_f* a, const vsip_vview_f* r);  
void vsip_vasin_d(const vsip_vview_d* a, const vsip_vview_d* r);
```

Description

This function computes the element-wise arcsine of the vector view `a` and stores the result in the vector view `r`.

$$r_i = \sin^{-1}(a_i)$$

Parameters

- `const vsip_vview_p* a`: Pointer to the source vector view.
- `const vsip_vview_p* r`: Pointer to the destination vector view.

Example

```
vsip_vview_f *src_vector_view;  
vsip_vview_f *dst_vector_view;  
  
// Assuming src_vector_view and dst_vector_view have been properly initialized  
vsip_vasin_f(src_vector_view, dst_vector_view);
```


4.1.3 vsip_vcos_p - Element-wise Cosine of a Vector View

```
void vsip_vcos_f(const vsip_vview_f* a, const vsip_vview_f* r);  
void vsip_vcos_d(const vsip_vview_d* a, const vsip_vview_d* r);
```

Description

This function computes the element-wise cosine of the vector view `a` and stores the result in the vector view `r`.

$$r_i = \cos(a_i)$$

Parameters

- `const vsip_vview_p* a`: Pointer to the source vector view.
- `const vsip_vview_p* r`: Pointer to the destination vector view.

Example

```
vsip_vview_f *src_vector_view;  
vsip_vview_f *dst_vector_view;  
  
// Assuming src_vector_view and dst_vector_view have been properly initialized  
vsip_vcos_f(src_vector_view, dst_vector_view);
```

4.1.4 vsip_vacos_p - Element-wise Arccosine of a Vector View

```
void vsip_vacos_f(const vsip_vview_f* a, const vsip_vview_f* r);  
void vsip_vacos_d(const vsip_vview_d* a, const vsip_vview_d* r);
```

Description

This function computes the element-wise cosine of the vector view a and stores the result in the vector view r.

$$r_i = \cos^{-1}(a_i)$$

Parameters

- const vsip_vview_p* a: Pointer to the source vector view.
- const vsip_vview_p* r: Pointer to the destination vector view.

Example

```
vsip_vview_f *src_vector_view;  
vsip_vview_f *dst_vector_view;  
  
// Assuming src_vector_view and dst_vector_view have been properly initialized  
vsip_vacos_f(src_vector_view, dst_vector_view);
```

4.1.5 vsip_vtan_p - Element-wise Tangent of a Vector View

```
void vsip_vtan_f(const vsip_vview_f* a, const vsip_vview_f* r);  
void vsip_vtan_d(const vsip_vview_d* a, const vsip_vview_d* r);
```

Description

This function computes the element-wise tangent of the vector view `a` and stores the result in the vector view `r`.

$$r_i = \tan(a_i)$$

Parameters

- `const vsip_vview_p* a`: Pointer to the source vector view.
- `const vsip_vview_p* r`: Pointer to the destination vector view.

Example

```
vsip_vview_f *src_vector_view;  
vsip_vview_f *dst_vector_view;
```

```
// Assuming src_vector_view and dst_vector_view have been properly initialized  
vsip_vtan_f(src_vector_view, dst_vector_view);
```

4.1.6 vsip_vatan_p - Element-wise Arctangent of a Vector View

```
void vsip_vatan_f(const vsip_vview_f* a, const vsip_vview_f* r);  
void vsip_vatan_d(const vsip_vview_d* a, const vsip_vview_d* r);
```

Description

This function computes the element-wise arctangent (inverse tangent) of the vector view a and stores the result in the vector view r.

$$r_i = \tan^{-1}(a_i)$$

Parameters

- const vsip_vview_p* a: Pointer to the source vector view.
- const vsip_vview_p* r: Pointer to the destination vector view.

Example

```
vsip_vview_f *src_vector_view;  
vsip_vview_f *dst_vector_view;  
  
// Assuming src_vector_view and dst_vector_view have been properly initialized  
vsip_vatan_f(src_vector_view, dst_vector_view);
```

4.1.7 vsip_vatan2_p - Element-wise Arctangent of Two Vector Views

```
void vsip_vatan2_f(const vsip_vview_f* a, const vsip_vview_f* b, const vsip_vview_f* r);
void vsip_vatan2_d(const vsip_vview_d* a, const vsip_vview_d* b, const vsip_vview_d* r);
```

Description

This function computes the element-wise arctangent of the quotient of the corresponding elements in the vector views a and b and stores the result in the vector view r.

$$r_i = \tan^{-1}(a/b)$$

Parameters

- const vsip_vview_p* a: Pointer to the first source vector view.
- const vsip_vview_p* b: Pointer to the second source vector view.
- const vsip_vview_p* r: Pointer to the destination vector view.

Example

```
vsip_vview_f *vector_view_a;
vsip_vview_f *vector_view_b;
vsip_vview_f *result_vector_view;
```

```
// Assuming vector_view_a, vector_view_b, and result_vector_view have been properly initialized
vsip_vatan2_f(vector_view_a, vector_view_b, result_vector_view);
```

4.1.8 vsip_dvexp_p - Element-wise Exponential of a Vector View

```
void vsip_vexp_f(const vsip_vview_f* a, const vsip_vview_f* r);  
void vsip_vexp_d(const vsip_vview_d* a, const vsip_vview_d* r);
```

Description

This function computes the element-wise exponential of the vector view a and stores the result in the vector view r.

$$r_i = e^{a_i}$$

Parameters

- const vsip_dvview_p* a: Pointer to the source vector view.
- const vsip_dvview_p* r: Pointer to the destination vector view.

Example

```
vsip_vview_f *src_vector_view;  
vsip_vview_f *dst_vector_view;
```

```
// Assuming src_vector_view and dst_vector_view have been properly initialized  
vsip_vexp_f(src_vector_view, dst_vector_view);
```

4.1.9 vsip_vexp10_p - Element-wise Base-10 Exponential of a Vector View

```
void vsip_vexp10_f(const vsip_vview_f* a, const vsip_vview_f* r);  
void vsip_vexp10_d(const vsip_vview_d* a, const vsip_vview_d* r);
```

Description

This function computes the element-wise base-10 exponential of the vector view `a` and stores the result in the vector view `r`.

$$r_i = 10^{a_i}$$

Parameters

- `const vsip_vview_p* a`: Pointer to the source vector view.
- `const vsip_vview_p* r`: Pointer to the destination vector view.

Example

```
vsip_vview_f *src_vector_view;  
vsip_vview_f *dst_vector_view;  
  
// Assuming src_vector_view and dst_vector_view have been properly initialized  
vsip_vexp10_f(src_vector_view, dst_vector_view);
```

4.1.10 vsip_vlog_p - Element-wise Natural Logarithm of a Vector View

```
void vsip_vlog_f(const vsip_vview_f* a, const vsip_vview_f* r);  
void vsip_vlog_d(const vsip_vview_d* a, const vsip_vview_d* r);
```

Description

This function computes the element-wise natural logarithm of the vector view `a` and stores the result in the vector view `r`.

$$r_i = \log(a_i)$$

Parameters

- `const vsip_vview_p* a`: Pointer to the source vector view.
- `const vsip_vview_p* r`: Pointer to the destination vector view.

Example

```
vsip_vview_f *src_vector_view;  
vsip_vview_f *dst_vector_view;  
  
// Assuming src_vector_view and dst_vector_view have been properly initialized  
vsip_vlog_f(src_vector_view, dst_vector_view);
```


4.1.11 vsip_vlog10_p - Element-wise Base-10 Logarithm of a Vector View

```
void vsip_vlog10_f(const vsip_vview_f* a, const vsip_vview_f* r);  
void vsip_vlog10_d(const vsip_vview_d* a, const vsip_vview_d* r);
```

Description

This function computes the element-wise base-10 logarithm of the vector view `a` and stores the result in the vector view `r`.

$$r_i = \log_{10}(a_i)$$

Parameters

- `const vsip_vview_p* a`: Pointer to the source vector view.
- `const vsip_vview_p* r`: Pointer to the destination vector view.

Example

```
vsip_vview_f *src_vector_view;  
vsip_vview_f *dst_vector_view;  
  
// Assuming src_vector_view and dst_vector_view have been properly initialized  
vsip_vlog10_f(src_vector_view, dst_vector_view);
```

4.1.12 vsip_dvsqrt_p - Element-wise Square Root of a Vector View

```
void vsip_vsqr_f(const vsip_vview_f* a, const vsip_vview_f* r);  
void vsip_vsqr_d(const vsip_vview_d* a, const vsip_vview_d* r);  
  
void vsip_cvsqr_f(const vsip_cvview_f* a, const vsip_cvview_f* r);  
void vsip_cvsqr_d(const vsip_cvview_d* a, const vsip_cvview_d* r);
```

Description

This function computes the element-wise square root of the vector view `a` and stores the result in the vector view `r`.

$$r_i = \sqrt{a_i}$$

Parameters

- `const vsip_dvview_p* a`: Pointer to the source vector view.
- `const vsip_dvview_p* r`: Pointer to the destination vector view.

Example

```
vsip_vview_f *src_vector_view;  
vsip_vview_f *dst_vector_view;  
  
// Assuming src_vector_view and dst_vector_view have been properly initialized  
vsip_vsqr_f(src_vector_view, dst_vector_view);
```

4.2 Unary Operations

4.2.1 vsip_dvneg_p - Negate Elements of a Vector View

```

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_vneg_d(const vsip_vview_d* a, const vsip_vview_d* r);
void vsip_cvneg_f(const vsip_cvview_f* a, const vsip_cvview_f* r);
void vsip_cvneg_d(const vsip_cvview_d* a, const vsip_cvview_d* r);

```

Description

This function negates each element of the vector view `a` and stores the result in the vector view `r`.

$$r_i = -a_i$$

Parameters

- `const vsip_dvview_p* a`: Pointer to the source vector view.
- `const vsip_dvview_p* r`: Pointer to the destination vector view.

Example

```

vsip_vview_f *src_vector_view;
vsip_vview_f *dst_vector_view;

// Assuming src_vector_view and dst_vector_view have been properly initialized
vsip_vneg_f(src_vector_view, dst_vector_view);

```

4.2.2 vsip_vsumval_p - Compute the Sum of Elements in a Vector View

```
vsip_scalar_f vsip_vsumval_f(const vsip_vview_f* a);  
vsip_scalar_d vsip_vsumval_d(const vsip_vview_d* a);
```

Description

This function computes the sum of all elements in the vector view a and returns it.

$$\sum_i^n a_i$$

Parameters

- `const vsip_vview_p* a`: Pointer to the vector view.

Return Value

- The sum of all elements in the vector view.

Example

```
vsip_vview_f *vector_view;  
vsip_scalar_f sum;  
  
// Assuming vector_view has been properly initialized  
sum = vsip_vsumval_f(vector_view);
```

Notes

- On boolean vector it counts the amount of logically true elements.

4.2.3 vsip_vsumsqval_p - Compute the Sum of Squares of Elements in a Vector View

```
vsip_scalar_f vsip_vsumsqval_f(const vsip_vview_f* a);  
vsip_scalar_d vsip_vsumsqval_d(const vsip_vview_d* a);
```

Description

This function computes the sum of the squares of all elements in the vector view a and returns it.

$$\sum_i^n a_i^2$$

Parameters

- const vsip_vview_p* a: Pointer to the vector view.

Return Value

- The sum of the squares of all elements in the vector view.

Example

```
vsip_vview_f *vector_view;  
vsip_scalar_f sum_of_squares;  
  
// Assuming vector_view has been properly initialized  
sum_of_squares = vsip_vsumsqval_f(vector_view);
```

4.2.4 vsip_dvmeanval_p - Compute Mean Value of Vector

```
vsip_scalar_f vsip_vmeanval_f(vsip_vview_f const *a);
vsip_scalar_d vsip_vmeanval_d(vsip_vview_d const *a);
vsip_cscalar_f vsip_cvmeanval_f(vsip_cvview_f const *a);
vsip_cscalar_d vsip_cvmeanval_d(vsip_cvview_d const *a);
```

Description

This function computes the arithmetic mean (average) value of all elements in a vector. The arithmetic mean is calculated as:

$$\text{mean} = \frac{1}{n} \sum_{i=0}^{n-1} a_i$$

where n is the length of the vector and a_i are the elements of the input vector.

Parameters

- `vsip_dvview_p const* a`: Input vector for which to compute the mean value.

Return Value

- Returns the arithmetic mean of all elements in the vector.

4.2.5 vsip_dvmeansqval_p - Compute Mean of Squared Values

```
vsip_scalar_f vsip_vmeansqval_f(vsip_vview_f const *a);
vsip_scalar_d vsip_vmeansqval_d(vsip_vview_d const *a);
vsip_cscalar_f vsip_cvmeansqval_f(vsip_cvview_f const *a);
vsip_cscalar_d vsip_cvmeansqval_d(vsip_cvview_d const *a);
```

Description

This function computes the mean of the squared values of all elements in a vector. The mean of squared values is calculated as:

$$\text{mean_sq} = \frac{1}{n} \sum_{i=0}^{n-1} |a_i|^2$$

where n is the length of the vector and a_i are the elements of the input vector.

Parameters

- `vsip_dvview_p const* a`: Input vector for which to compute the mean of squared values.

Return Value

- Returns the mean of the squared values of all elements in the vector.

4.2.6 vsip_vsq_p - Square Elements of a Vector View

```
void vsip_vsq_f(const vsip_vview_f* a, const vsip_vview_f* r);  
void vsip_vsq_d(const vsip_vview_d* a, const vsip_vview_d* r);
```

Description

This function squares each element of the vector view `a` and stores the result in the vector view `r`.

$$r_i = a_i^2$$

Parameters

- `const vsip_vview_p* a`: Pointer to the source vector view.
- `const vsip_vview_p* r`: Pointer to the destination vector view.

Example

```
vsip_vview_f *src_vector_view;  
vsip_vview_f *dst_vector_view;
```

```
// Assuming src_vector_view and dst_vector_view have been properly initialized  
vsip_vsq_f(src_vector_view, dst_vector_view);
```

4.2.7 vsip_vrecip_p - Compute Reciprocal of Elements of a Vector View

```
void vsip_vrecip_f(const vsip_vview_f* a, const vsip_vview_f* r);  
void vsip_vrecip_d(const vsip_vview_d* a, const vsip_vview_d* r);
```

Description

This function computes the reciprocal of each element of the vector view `a` and stores the result in the vector view `r`.

$$r_i = \frac{1}{a_i}$$

Parameters

- `const vsip_vview_p* a`: Pointer to the source vector view.
- `const vsip_vview_p* r`: Pointer to the destination vector view.

Example

```
vsip_vview_f *src_vector_view;  
vsip_vview_f *dst_vector_view;  
  
// Assuming src_vector_view and dst_vector_view have been properly initialized  
vsip_vrecip_f(src_vector_view, dst_vector_view);
```

4.2.8 vsip_dvmag_p - Compute Magnitude of Elements of a Vector View

```

void vsip_vmag_i(const vsip_vview_i* a, const vsip_vview_i* r);
void vsip_vmag_f(const vsip_vview_f* a, const vsip_vview_f* r);
void vsip_vmag_d(const vsip_vview_d* a, const vsip_vview_d* r);
void vsip_cvmag_f(const vsip_cvview_f* a, const vsip_vview_f* r);
void vsip_cvmag_d(const vsip_cvview_d* a, const vsip_vview_d* r);

```

Description

This function computes the magnitude (absolute value) of each element of the vector view `a` and stores the result in the vector view `r`.

$$r_i = |a_i|$$

Parameters

- `const vsip_dvview_p* a`: Pointer to the source vector view.
- `const vsip_dvview_p* r`: Pointer to the destination vector view.

Example

```

vsip_vview_f *src_vector_view;
vsip_vview_f *dst_vector_view;

// Assuming src_vector_view and dst_vector_view have been properly initialized
vsip_vmag_f(src_vector_view, dst_vector_view);

```

4.2.9 vsip_vcmagsq_p - Element-wise Magnitude Squared of a Complex Vector View

```
void vsip_vcmagsq_f(const vsip_cvview_f* a, const vsip_vview_f* r);  
void vsip_vcmagsq_d(const vsip_cvview_d* a, const vsip_vview_d* r);
```

Description

This function computes the element-wise magnitude squared of the complex vector view `a` and stores the result in the real vector view `r`.

Parameters

- `const vsip_cvview_p* a`: Pointer to the source complex vector view.
- `const vsip_vview_p* r`: Pointer to the destination real vector view.

Example

```
vsip_cvview_f *complex_vector;  
vsip_vview_f *real_vector;  
  
// Assuming complex_vector and real_vector have been properly initialized  
vsip_vcmagsq_f(complex_vector, real_vector);
```

4.2.10 vsip_cvconj_p - Element-wise Complex Conjugate of a Complex Vector View

```
void vsip_cvconj_f(const vsip_cvview_f* a, const vsip_cvview_f* r);  
void vsip_cvconj_d(const vsip_cvview_d* a, const vsip_cvview_d* r);
```

Description

This function computes the element-wise complex conjugate of the complex vector view a and stores the result in the complex vector view r.

Parameters

- const vsip_cvview_p* a: Pointer to the source complex vector view.
- const vsip_cvview_p* r: Pointer to the destination complex vector view.

Example

```
vsip_cvview_f *complex_vector;  
vsip_cvview_f *result_vector;  
  
// Assuming complex_vector and result_vector have been properly initialized  
vsip_cvconj_f(complex_vector, result_vector);
```

4.2.11 vsip_veuler_p - Convert Real Vector to Complex Euler Representation

```
void vsip_veuler_f(vsip_vview_f const *a, vsip_cvview_f const *r);  
void vsip_veuler_d(vsip_vview_d const *a, vsip_cvview_d const *r);
```

Description

This function converts a real vector of angles (in radians) to a complex vector using Euler's formula. The operation performs element-wise conversion using Euler's formula:

$$r_i = e^{ja_i} = \cos(a_i) + j \sin(a_i)$$

for all i from 0 to $n-1$, where n is the length of the vectors, a_i is the angle in radians from the input vector, and r_i is the corresponding complex number on the unit circle.

Parameters

- `vsip_vview_p const* a`: Input real vector containing angles in radians.
- `vsip_cvview_p const* r`: Output complex vector that will store the results.

Notes

- Both vectors must have the same length.
- The input angles must be in radians.

4.2.12 vsip_dvmodulate_p - Vector Modulation with Complex Carrier

```

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_cscalar_f cvsip_vmodulate_f(const vsip_cvview_f *a, vsip_scalar_f nu,
                                vsip_scalar_f phi, const vsip_cvview_f *r);

vsip_scalar_d vsip_vmodulate_d(const vsip_vview_d *a, vsip_scalar_d nu,
                               vsip_scalar_d phi, const vsip_cvview_d *r);
vsip_cscalar_d cvsip_vmodulate_d(const vsip_cvview_d *a, vsip_scalar_d nu,
                                vsip_scalar_d phi, const vsip_cvview_d *r);

```

Description

This function modulates a real-valued baseband signal with a complex exponential carrier. The modulation is performed as:

$$r_i = a_i \cdot e^{j(2\pi\nu i + \phi)}$$

for all i from 0 to $n - 1$, where n is the length of the vectors, a_i is the input signal, and r_i is the complex modulated output.

Parameters

- `const vsip_dvview_p* a`: Input real vector containing the baseband signal to be modulated.
- `vsip_scalar_f nu`: Normalized frequency of the carrier in cycles per sample, ν .
- `vsip_scalar_f phi`: Initial phase of the carrier in radians, ϕ .
- `const vsip_cvview_p* r`: Output complex vector that will store the modulated signal.

Return Value

- Returns the final phase of the carrier (in radians) after modulation, which can be used for continuous phase modulation across multiple calls.

Notes

- The input and output vectors must have the same length.

4.2.13 vsip_dvrsqrt_p - Element-wise Reciprocal Square Root of a Vector View

```
void vsip_vrsqrt_f(const vsip_vview_f* a, const vsip_vview_f* r);  
void vsip_vrsqrt_d(const vsip_vview_d* a, const vsip_vview_d* r);
```

Description

This function computes the element-wise reciprocal square root of the vector view `a` and stores the result in the vector view `r`.

$$r_i = \frac{1}{\sqrt{a_i}}$$

Parameters

- `const vsip_dvview_p* a`: Pointer to the source vector view.
- `const vsip_dvview_p* r`: Pointer to the destination vector view.

Example

```
vsip_vview_f *src_vector_view;  
vsip_vview_f *dst_vector_view;  
  
// Assuming src_vector_view and dst_vector_view have been properly initialized  
vsip_vrsqrt_f(src_vector_view, dst_vector_view);
```


4.3 Binary Operations

4.3.1 vsip_dvadd_p - Element-wise Addition of Two Vector Views

```
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_vadd_d(const vsip_vview_d* a, const vsip_vview_d* b, const vsip_vview_d* r);
void vsip_cvadd_f(const vsip_cvview_f* a, const vsip_cvview_f* b, const vsip_cvview_f* r);
void vsip_cvadd_d(const vsip_cvview_d* a, const vsip_cvview_d* b, const vsip_cvview_d* r);
```

Description

This function performs element-wise addition of the vector views a and b and stores the result in the vector view r.

Parameters

- const vsip_dvview_p* a: Pointer to the first source vector view.
- const vsip_dvview_p* b: Pointer to the second source vector view.
- const vsip_dvview_p* r: Pointer to the destination vector view.

Example

```
vsip_vview_f *vector_view_a;
vsip_vview_f *vector_view_b;
vsip_vview_f *result_vector_view;

// Assuming vector_view_a, vector_view_b, and result_vector_view have been properly initialized
vsip_vadd_f(vector_view_a, vector_view_b, result_vector_view);
```

4.3.2 vsip_dvsub_p - Element-wise Subtraction of Two Vector Views

```
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_vsub_d(const vsip_vview_d* a, const vsip_vview_d* b, const vsip_vview_d* r);
void vsip_cvsub_f(const vsip_cvview_f* a, const vsip_cvview_f* b, const vsip_cvview_f* r);
void vsip_cvsub_d(const vsip_cvview_d* a, const vsip_cvview_d* b, const vsip_cvview_d* r);
```

Description

This function performs element-wise subtraction of the vector view b from the vector view a and stores the result in the vector view r.

Parameters

- const vsip_dvview_p* a: Pointer to the minuend vector view.
- const vsip_dvview_p* b: Pointer to the subtrahend vector view.
- const vsip_dvview_p* r: Pointer to the destination vector view.

Example

```
vsip_vview_f *vector_view_a;
vsip_vview_f *vector_view_b;
vsip_vview_f *result_vector_view;

// Assuming vector_view_a, vector_view_b, and result_vector_view have been properly initialized
vsip_vsub_f(vector_view_a, vector_view_b, result_vector_view);
```

4.3.3 vsip_dvmul_p - Element-wise Multiplication of Two Vector Views

```
void vsip_vmul_f(const vsip_vview_f* a, const vsip_vview_f* b, const vsip_vview_f* r);
void vsip_vmul_d(const vsip_vview_d* a, const vsip_vview_d* b, const vsip_vview_d* r);
void vsip_cvmul_f(const vsip_cvview_f* a, const vsip_cvview_f* b, const vsip_cvview_f* r);
void vsip_cvmul_d(const vsip_cvview_d* a, const vsip_cvview_d* b, const vsip_cvview_d* r);
```

Description

This function performs element-wise multiplication of the vector views `a` and `b` and stores the result in the vector view `r`.

Parameters

- `const vsip_dvview_p* a`: Pointer to the first source vector view.
- `const vsip_dvview_p* b`: Pointer to the second source vector view.
- `const vsip_dvview_p* r`: Pointer to the destination vector view.

Example

```
vsip_vview_f *vector_view_a;
vsip_vview_f *vector_view_b;
vsip_vview_f *result_vector_view;
```

```
// Assuming vector_view_a, vector_view_b, and result_vector_view have been properly initialized
vsip_vmul_f(vector_view_a, vector_view_b, result_vector_view);
```

4.3.4 vsip_dvdiv_p - Element-wise Division of Two Vector Views

```
void vsip_vdiv_f(const vsip_vview_f* a, const vsip_vview_f* b, const vsip_vview_f* r);  
void vsip_vdiv_d(const vsip_vview_d* a, const vsip_vview_d* b, const vsip_vview_d* r);  
void vsip_cvdiv_f(const vsip_cvview_f* a, const vsip_cvview_f* b, const vsip_cvview_f* r);  
void vsip_cvdiv_d(const vsip_cvview_d* a, const vsip_cvview_d* b, const vsip_cvview_d* r);
```

Description

This function performs element-wise division of the vector view a by the vector view b and stores the result in the vector view r.

Parameters

- const vsip_vview_p* a: Pointer to the numerator vector view.
- const vsip_vview_p* b: Pointer to the denominator vector view.
- const vsip_vview_p* r: Pointer to the destination vector view.

Example

```
vsip_vview_f *vector_view_a;  
vsip_vview_f *vector_view_b;  
vsip_vview_f *result_vector_view;  
  
// Assuming vector_view_a, vector_view_b, and result_vector_view have been properly initialized  
vsip_vdiv_f(vector_view_a, vector_view_b, result_vector_view);
```

4.3.5 vsip_cvjmul_p - Element-wise Complex Conjugate Multiplication of Two Complex Vector Views

```
void vsip_cvjmul_f(const vsip_cvview_f* a, const vsip_cvview_f* b, const vsip_cvview_f* w);  
void vsip_cvjmul_d(const vsip_cvview_d* a, const vsip_cvview_d* b, const vsip_cvview_d* w);
```

Description

This function performs element-wise complex conjugate multiplication of the complex vector views a and b and stores the result in the complex vector view w.

Parameters

- const vsip_cvview_p* a: Pointer to the first source complex vector view.
- const vsip_cvview_p* b: Pointer to the second source complex vector view.
- const vsip_cvview_p* w: Pointer to the destination complex vector view.

Example

```
vsip_cvview_f *complex_vector_a;  
vsip_cvview_f *complex_vector_b;  
vsip_cvview_f *result_vector;
```

```
// Assuming complex_vector_a, complex_vector_b, and result_vector have been properly initialized  
vsip_cvjmul_f(complex_vector_a, complex_vector_b, result_vector);
```

4.3.6 vsip_rcvadd_p - Element-wise Real-Complex Addition

```
void vsip_rcvadd_f(const vsip_vview_f* a, const vsip_cvview_f* b, const vsip_cvview_f* r);  
void vsip_rcvadd_d(const vsip_vview_d* a, const vsip_cvview_d* b, const vsip_cvview_d* r);
```

Description

This function performs element-wise addition of the real vector view a and the complex vector view b and stores the result in the complex vector view r.

Parameters

- const vsip_vview_p* a: Pointer to the source real vector view.
- const vsip_cvview_p* b: Pointer to the source complex vector view.
- const vsip_cvview_p* r: Pointer to the destination complex vector view.

Example

```
vsip_vview_f *real_vector;  
vsip_cvview_f *complex_vector;  
vsip_cvview_f *result_vector;  
  
// Assuming real_vector, complex_vector, and result_vector have been properly initialized  
vsip_rcvadd_f(real_vector, complex_vector, result_vector);
```

4.3.7 vsip_rcvsub_p - Element-wise Real-Complex Subtraction

```
void vsip_rcvsub_f(const vsip_vview_f* a, const vsip_cvview_f* b, const vsip_cvview_f* r);  
void vsip_rcvsub_d(const vsip_vview_d* a, const vsip_cvview_d* b, const vsip_cvview_d* r);
```

Description

This function performs element-wise subtraction of the real vector view `a` and the complex vector view `b` and stores the result in the complex vector view `r`.

Parameters

- `const vsip_vview_p* a`: Pointer to the source real vector view.
- `const vsip_cvview_p* b`: Pointer to the source complex vector view.
- `const vsip_cvview_p* r`: Pointer to the destination complex vector view.

Example

```
vsip_vview_f *real_vector;  
vsip_cvview_f *complex_vector;  
vsip_cvview_f *result_vector;  
  
// Assuming real_vector, complex_vector, and result_vector have been properly initialized  
vsip_rcvsub_f(real_vector, complex_vector, result_vector);
```


4.3.8 vsip_rcvmul_p - Element-wise Real-Complex Multiplication

```
void vsip_rcvmul_f(const vsip_vview_f* a, const vsip_cvview_f* b, const vsip_cvview_f* r);  
void vsip_rcvmul_d(const vsip_vview_d* a, const vsip_cvview_d* b, const vsip_cvview_d* r);
```

Description

This function performs element-wise multiplication of the real vector view a and the complex vector view b and stores the result in the complex vector view r.

Parameters

- const vsip_vview_p* a: Pointer to the source real vector view.
- const vsip_cvview_p* b: Pointer to the source complex vector view.
- const vsip_cvview_p* r: Pointer to the destination complex vector view.

Example

```
vsip_vview_f *real_vector;  
vsip_cvview_f *complex_vector;  
vsip_cvview_f *result_vector;  
  
// Assuming real_vector, complex_vector, and result_vector have been properly initialized  
vsip_rcvmul_f(real_vector, complex_vector, result_vector);
```

4.3.9 vsip_crvsub_p - Element-wise Complex-Real Subtraction

```
void vsip_crvsub_f(const vsip_vview_f* a, const vsip_cvview_f* b, const vsip_cvview_f* r);  
void vsip_crvsub_d(const vsip_vview_d* a, const vsip_cvview_d* b, const vsip_cvview_d* r);
```

Description

This function performs element-wise subtraction of the complex vector view a and the real vector view b and stores the result in the complex vector view r.

Parameters

- const vsip_cvview_p* a: Pointer to the source complex vector view.
- const vsip_vview_p* b: Pointer to the source real vector view.
- const vsip_cvview_p* r: Pointer to the destination complex vector view.

Example

```
vsip_vview_f *real_vector;  
vsip_cvview_f *complex_vector;  
vsip_cvview_f *result_vector;  
  
// Assuming real_vector, complex_vector, and result_vector have been properly initialized  
vsip_rcvsub_f(complex_vector, real_vector, result_vector);
```

4.3.10 vsip_dsvadd_p - Add a Scalar to a Vector View

```

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_svadd_d(vsip_scalar_d alpha, const vsip_vview_d* b, const vsip_vview_d* r);

void vsip_csvadd_f(vsip_cscalar_f alpha, const vsip_cvview_f* b, const vsip_cvview_f* r);
void vsip_csvadd_d(vsip_cscalar_d alpha, const vsip_cvview_d* b, const vsip_cvview_d* r);

```

Description

This function adds the scalar alpha to each element of the vector view b and stores the result in the vector view r.

Parameters

- vsip_dscalar_p alpha: The scalar value to add.
- const vsip_dvview_p* b: Pointer to the source vector view.
- const vsip_dvview_p* r: Pointer to the destination vector view.

Example

```

vsip_vview_f *src_vector_view;
vsip_vview_f *dst_vector_view;
vsip_scalar_f scalar = 2.0;

// Assuming src_vector_view and dst_vector_view have been properly initialized
vsip_svadd_f(scalar, src_vector_view, dst_vector_view);

```

4.3.11 vsip_dsvsub_p - Subtract a Scalar to a Vector View

```
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_svsub_d(vsip_scalar_d alpha, const vsip_vview_d* b, const vsip_vview_d* r);
void vsip_csvsub_f(vsip_cscalar_f alpha, const vsip_cvview_f* b, const vsip_cvview_f* r);
void vsip_csvsub_d(vsip_cscalar_d alpha, const vsip_cvview_d* b, const vsip_cvview_d* r);
```

Description

This function subtracts the scalar alpha to each element of the vector view b and stores the result in the vector view r.

Parameters

- vsip_dscalar_p alpha: The scalar value to add.
- const vsip_dvview_p* b: Pointer to the source vector view.
- const vsip_dvview_p* r: Pointer to the destination vector view.

Example

```
vsip_vview_f *src_vector_view;
vsip_vview_f *dst_vector_view;
vsip_scalar_f scalar = 2.0;

// Assuming src_vector_view and dst_vector_view have been properly initialized
vsip_svsub_f(scalar, src_vector_view, dst_vector_view);
```

4.3.12 vsip_dsvmul_p - Multiply a Scalar by a Vector View

```
void vsip_svmul_i(vsip_scalar_i alpha, const vsip_vview_i* b, const vsip_vview_i* r);
void vsip_svmul_f(vsip_scalar_f alpha, const vsip_vview_f* b, const vsip_vview_f* r);
void vsip_svmul_d(vsip_scalar_d alpha, const vsip_vview_d* b, const vsip_vview_d* r);
void vsip_csvm_f(vsip_cscalar_f alpha, const vsip_cvview_f* b, const vsip_cvview_f* r);
void vsip_csvm_d(vsip_cscalar_d alpha, const vsip_cvview_d* b, const vsip_cvview_d* r);
```

Description

This function multiplies each element of the vector view `b` by the scalar `alpha` and stores the result in the vector view `r`.

Parameters

- `vsip_dscalar_p alpha`: The scalar value to multiply by.
- `const vsip_dvview_p* b`: Pointer to the source vector view.
- `const vsip_dvview_p* r`: Pointer to the destination vector view.

Example

```
vsip_vview_f *src_vector_view;
vsip_vview_f *dst_vector_view;
vsip_scalar_f scalar = 2.0;

// Assuming src_vector_view and dst_vector_view have been properly initialized
vsip_svmul_f(scalar, src_vector_view, dst_vector_view);
```

4.3.13 vsip_dsvdiv_p - Divide a Scalar by a Vector View

```
void vsip_svdiv_f(vsip_scalar_f alpha, const vsip_vview_f* b, const vsip_vview_f* r);
void vsip_svdiv_d(vsip_scalar_d alpha, const vsip_vview_d* b, const vsip_vview_d* r);

void vsip_csvdiv_f(vsip_cscalar_f alpha, const vsip_cvview_f* b, const vsip_cvview_f* r);
void vsip_csvdiv_d(vsip_cscalar_d alpha, const vsip_cvview_d* b, const vsip_cvview_d* r);
```

Description

This function divides the scalar alpha by each element of the vector view b and stores the result in the vector view r.

Parameters

- vsip_dscalar_p alpha: The scalar value to divide.
- const vsip_dvview_p* b: Pointer to the source vector view.
- const vsip_dvview_p* r: Pointer to the destination vector view.

Example

```
vsip_vview_f *src_vector_view;
vsip_vview_f *dst_vector_view;
vsip_scalar_f scalar = 2.0;

// Assuming src_vector_view and dst_vector_view have been properly initialized
vsip_svdiv_f(scalar, src_vector_view, dst_vector_view);
```

4.3.14 vsip_rscvadd_p - Element-wise Real-Scalar-Complex Addition

```
void vsip_rscvadd_f(vsip_scalar_f alpha, const vsip_cvview_f* b, const vsip_cvview_f* r);  
void vsip_rscvadd_d(vsip_scalar_d alpha, const vsip_cvview_d* b, const vsip_cvview_d* r);
```

Description

This function performs element-wise addition of the scalar alpha and the complex vector view b and stores the result in the complex vector view r.

Parameters

- vsip_scalar_p alpha: The scalar value add.
- const vsip_cvview_p* b: Pointer to the source complex vector view.
- const vsip_cvview_p* r: Pointer to the destination complex vector view.

Example

```
vsip_scalar_f scalar = 2.0;  
vsip_cvview_f *complex_vector;  
vsip_cvview_f *result_vector;  
  
// Assuming complex_vector and result_vector have been properly initialized  
vsip_rscvadd_f(scalar, complex_vector, result_vector);
```

4.3.15 vsip_rscvsub_p - Element-wise Real-Scalar-Complex Subtraction

```
void vsip_rscvsub_f(vsip_scalar_f alpha, const vsip_cvview_f* b, const vsip_cvview_f* r);  
void vsip_rscvsub_d(vsip_scalar_d alpha, const vsip_cvview_d* b, const vsip_cvview_d* r);
```

Description

This function performs element-wise subtraction of the scalar alpha and the complex vector view b and stores the result in the complex vector view r.

Parameters

- vsip_scalar_p alpha: The scalar value to subtract.
- const vsip_cvview_p* b: Pointer to the source complex vector view.
- const vsip_cvview_p* r: Pointer to the destination complex vector view.

Example

```
vsip_scalar_f scalar = 2.0;  
vsip_cvview_f *complex_vector;  
vsip_cvview_f *result_vector;  
  
// Assuming complex_vector and result_vector have been properly initialized  
vsip_rscvsub_f(scalar, complex_vector, result_vector);
```


4.3.16 vsip_rscvmul_p - Element-wise Real-Scalar-Complex Multiplication

```
void vsip_rscvmul_f(vsip_scalar_f alpha, const vsip_cvview_f* b, const vsip_cvview_f* r);  
void vsip_rscvmul_d(vsip_scalar_d alpha, const vsip_cvview_d* b, const vsip_cvview_d* r);
```

Description

This function performs element-wise multiplication of the scalar alpha and the complex vector view b and stores the result in the complex vector view r.

Parameters

- vsip_scalar_p alpha: The scalar value to multiply by.
- const vsip_cvview_p* b: Pointer to the source complex vector view.
- const vsip_cvview_p* r: Pointer to the destination complex vector view.

Example

```
vsip_scalar_f scalar = 2.0;  
vsip_cvview_f *complex_vector;  
vsip_cvview_f *result_vector;  
  
// Assuming complex_vector and result_vector have been properly initialized  
vsip_rscvmul_f(scalar, complex_vector, result_vector);
```

4.3.17 vsip_d vmmul_p - Vector-Matrix Multiplication

```

void vsip_vmmul_f(const vsip_vview_f *a, const vsip_mview_f *b,
                  vsip_major major, const vsip_mview_f *r);
void vsip_vmmul_d(const vsip_vview_d *a, const vsip_mview_d *b,
                  vsip_major major, const vsip_mview_d *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_cvmmul_d(const vsip_cvview_d *a, const vsip_cmview_d *b,
                  vsip_major major, const vsip_cmview_d *r);

```

Description

This function performs a vector-matrix multiplication. The operation performs either:

- Row-wise multiplication if `major = VSIP_ROW`:

$$r_{i,j} = a_j \cdot b_{i,j}$$

- Column-wise multiplication if `major = VSIP_COL`:

$$r_{i,j} = a_i \cdot b_{i,j}$$

for all i from 0 to $m - 1$ and j from 0 to $n - 1$, where m is the number of rows and n is the number of columns in the matrix.

Parameters

- `const vsip_d vview_p *` `a`: Input vector containing the multiplier elements.
- `const vsip_d mview_p *` `b`: Input matrix to be multiplied.
- `vsip_major major`: Storage order specifying the multiplication direction:
 - `VSIP_ROW`: Multiply each row of the matrix by the corresponding vector element
 - `VSIP_COL`: Multiply each column of the matrix by the corresponding vector element
- `const vsip_d mview_p *` `r`: Output matrix that will store the results.

Notes

- The length of the vector must match either the number of columns (for `VSIP_COL`) or the number of rows (for `VSIP_ROW`) in the matrix.

4.3.18 vsip_rvcmmul_p - Real Vector-Complex Matrix Multiplication

```
void vsip_rvcmmul_f(const vsip_vview_f* a, const vsip_cmview_f* b,
                   vsip_major major, const vsip_cmview_f* r);
void vsip_rvcmmul_d(const vsip_vview_d* a, const vsip_cmview_d* b,
                   vsip_major major, const vsip_cmview_d* r);
```

Description

This function performs a real vector-complex matrix multiplication. The operation performs either:

- Row-wise multiplication if `major = VSIP_ROW`:

$$r_{i,j} = a_j \cdot b_{i,j}$$

- Column-wise multiplication if `major = VSIP_COL`:

$$r_{i,j} = a_i \cdot b_{i,j}$$

for all i from 0 to $m - 1$ and j from 0 to $n - 1$, where m is the number of rows and n is the number of columns in the matrix.

Parameters

- `const vsip_vview_p* a`: Input real vector containing the multiplier elements.
- `const vsip_cmview_p* b`: Input complex matrix to be multiplied.
- `vsip_major major`: Storage order specifying the multiplication direction:
 - `VSIP_ROW`: Multiply each row of the matrix by the corresponding vector element
 - `VSIP_COL`: Multiply each column of the matrix by the corresponding vector element
- `const vsip_cmview_p* r`: Output complex matrix that will store the results.

Notes

- The length of the real vector must match either the number of columns (for `VSIP_COL`) or the number of rows (for `VSIP_ROW`) in the complex matrix.

4.3.19 vsip_dvexpoavg_p - Vector Exponential Average

```

void vsip_vexpoavg_f(vsip_scalar_f alpha, const vsip_vview_f *b, const vsip_vview_f *c);
void vsip_vexpoavg_d(vsip_scalar_d alpha, const vsip_vview_d *b, const vsip_vview_d *c);
void vsip_cvexpoavg_f(vsip_cscalar_f alpha, const vsip_cvview_f *b, const vsip_cvview_f *c);
void vsip_cvexpoavg_d(vsip_cscalar_d alpha, const vsip_cvview_d *b, const vsip_cvview_d *c);

```

Description

This function computes the exponential average (also known as exponential moving average or first-order IIR filtering) of a vector. It updates the accumulator vector c using the input vector b and a smoothing factor α . The operation performs element-wise exponential averaging:

$$c_i = \alpha \cdot b_i + (1 - \alpha) \cdot c_i$$

for all i from 0 to $n - 1$, where n is the length of the vectors, α is the smoothing factor $0 < \alpha \leq 1$, b_i is the current input value, and c_i is the accumulated average value.

The smoothing factor α determines the weight of the new data relative to the accumulated average:

- Smaller α values (closer to 0) give more weight to the accumulated average (more smoothing)
- Larger α values (closer to 1) give more weight to the new data (less smoothing)

Parameters

- `vsip_dscalar_p alpha`: Smoothing factor.
- `const vsip_dvview_p* b`: Input vector containing the new data values.
- `const vsip_dvview_p* c`: Input/Output vector that contains the accumulated average values on input and will store the updated averages on output.

Notes

- Both vectors must have the same length.
- The smoothing factor α should be in the range (0,1]. Values outside this range may lead to numerical instability.

4.3.20 vsip_vhypot_p - Vector Hypotenuse (Euclidean Norm)

```
void vsip_vhypot_f(const vsip_vview_f *a, const vsip_vview_f *b, const vsip_vview_f *r);  
void vsip_vhypot_d(const vsip_vview_d *a, const vsip_vview_d *b, const vsip_vview_d *r);
```

Description

This function computes the element-wise hypotenuse (Euclidean norm) of corresponding elements from two vectors. The operation performs element-wise computation:

$$r_i = \sqrt{a_i^2 + b_i^2}$$

for all i from 0 to $n - 1$, where n is the length of the vectors.

Parameters

- `const vsip_vview_p * a`: First input vector containing one set of components.
- `const vsip_vview_p * b`: Second input vector containing the other set of components.
- `const vsip_vview_p * r`: Output vector that will store the hypotenuse results.

Notes

- All three vectors must have the same length.

4.4 Ternary Operations

4.4.1 vsip_dvam_p - Vector Add-Multiply

```

void vsip_vam_f(vsip_vview_f const *a, vsip_vview_f const *b,
               vsip_vview_f const *c, vsip_vview_f const *r);
void vsip_vam_d(vsip_vview_d const *a, vsip_vview_d const *b,
               vsip_vview_d const *c, vsip_vview_d const *r);
void vsip_cvam_f(vsip_cvview_f const *a, vsip_cvview_f const *b,
               vsip_cvview_f const *c, vsip_cvview_f const *r);
void vsip_cvam_d(vsip_cvview_d const *a, vsip_cvview_d const *b,
               vsip_cvview_d const *c, vsip_cvview_d const *r);

```

Description

This function performs an element-wise add-multiply operation on three input vectors. The operation performs element-wise computation:

$$r_i = (a_i + b_i) \cdot c_i$$

for all i from 0 to $n - 1$, where n is the length of the vectors.

Parameters

- vsip_dvview_p const* a: First input vector.
- vsip_dvview_p const* b: Second input vector.
- vsip_dvview_p const* c: Third input vector.
- vsip_dvview_p const* r: Output vector that will store the results.

4.4.2 vsip_dvma_p - Vector Multiply-Add

```

void vsip_vma_f(vsip_vview_f const *a, vsip_vview_f const *b,
               vsip_vview_f const *c, vsip_vview_f const *r);
void vsip_vma_d(vsip_vview_d const *a, vsip_vview_d const *b,
               vsip_vview_d const *c, vsip_vview_d const *r);

void vsip_cvma_f(vsip_cvview_f const *a, vsip_cvview_f const *b,
               vsip_cvview_f const *c, vsip_cvview_f const *r);
void vsip_cvma_d(vsip_cvview_d const *a, vsip_cvview_d const *b,
               vsip_cvview_d const *c, vsip_cvview_d const *r);

```

Description

This function performs an element-wise multiply-add operation on three input vectors. The operation performs element-wise computation:

$$r_i = (a_i \cdot b_i) + c_i$$

for all i from 0 to $n - 1$, where n is the length of the vectors.

Parameters

- vsip_dvview_p const* a: First input vector.
- vsip_dvview_p const* b: Second input vector.
- vsip_dvview_p const* c: Third input vector.
- vsip_dvview_p const* r: Output vector that will store the results.

4.4.3 vsip_dvmsb_p - Vector Multiply-Subtract

```
void vsip_vmsb_f(vsip_vview_f const *a, vsip_vview_f const *b,
                vsip_vview_f const *c, vsip_vview_f const *r);
void vsip_vmsb_d(vsip_vview_d const *a, vsip_vview_d const *b,
                vsip_vview_d const *c, vsip_vview_d const *r);
void vsip_cvmsb_f(vsip_cvview_f const *a, vsip_cvview_f const *b,
                vsip_cvview_f const *c, vsip_cvview_f const *r);
void vsip_cvmsb_d(vsip_cvview_d const *a, vsip_cvview_d const *b,
                vsip_cvview_d const *c, vsip_cvview_d const *r);
```

Description

This function performs an element-wise multiply-subtract operation on three input vectors. The operation performs element-wise computation:

$$r_i = (a_i \cdot b_i) - c_i$$

for all i from 0 to $n - 1$, where n is the length of the vectors.

Parameters

- vsip_dvview_p const* a: First input vector.
- vsip_dvview_p const* b: Second input vector.
- vsip_dvview_p const* c: Third input vector.
- vsip_dvview_p const* r: Output vector that will store the results.

4.4.4 vsip_dvmsa_p - Vector Multiply-Scalar-Add

```

void vsip_vmsa_f(vsip_vview_f const *a, vsip_vview_f const *b,
                vsip_scalar_f alpha, vsip_vview_f const *r);
void vsip_vmsa_d(vsip_vview_d const *a, vsip_vview_d const *b,
                vsip_scalar_d alpha, vsip_vview_d const *r);

void vsip_cvmsa_f(vsip_cvview_f const *a, vsip_cvview_f const *b,
                vsip_cscalar_f alpha, vsip_cvview_f const *r);
void vsip_cvmsa_d(vsip_cvview_d const *a, vsip_cvview_d const *b,
                vsip_cscalar_d alpha, vsip_cvview_d const *r);

```

Description

This function performs an element-wise multiply-scalar-add operation on two input vectors and a scalar constant. The operation performs element-wise computation:

$$r_i = (a_i \cdot b_i) + \alpha$$

for all i from 0 to $n - 1$, where n is the length of the vectors.

Parameters

- vsip_dvview_p const* a: First input vector.
- vsip_dvview_p const* b: Second input vector.
- vsip_dscalar_p alpha: Scalar argument α
- vsip_dvview_p const* r: Output vector that will store the results.

4.4.5 vsip_dvsam_p - Vector Vector-Add-Scalar-Multiply

```

void vsip_vsam_f(vsip_vview_f const *a, vsip_scalar_f beta,
                 vsip_vview_f const *c, vsip_vview_f const *r);
void vsip_vsam_d(vsip_vview_d const *a, vsip_scalar_d beta,
                 vsip_vview_d const *c, vsip_vview_d const *r);

void vsip_cvsam_f(vsip_cvview_f const *a, vsip_cscalar_f beta,
                 vsip_cvview_f const *c, vsip_cvview_f const *r);
void vsip_cvsam_d(vsip_cvview_d const *a, vsip_cscalar_d beta,
                 vsip_cvview_d const *c, vsip_cvview_d const *r);

```

Description

This function performs an element-wise vector-add-scalar-multiply operation on two input vectors and a scalar constant. The operation performs element-wise computation:

$$r_i = \beta(a_i + b_i)$$

for all i from 0 to $n - 1$, where n is the length of the vectors.

Parameters

- vsip_dvview_p const* a: First input vector.
- vsip_dscalar_p beta: Scalar constant β .
- vsip_dvview_p const* c: Third input vector.
- vsip_dvview_p const* r: Output vector that will store the results.

4.4.6 vsip_dvsbm_p - Vector Subtract-Multiply

```

void vsip_vsbm_f(vsip_vview_f const *a, vsip_vview_f const *b,
                vsip_vview_f const *c, vsip_vview_f const *r);
void vsip_vsbm_d(vsip_vview_d const *a, vsip_vview_d const *b,
                vsip_vview_d const *c, vsip_vview_d const *r);
void vsip_cvsbm_f(vsip_cvview_f const *a, vsip_cvview_f const *b,
                vsip_cvview_f const *c, vsip_cvview_f const *r);
void vsip_cvsbm_d(vsip_cvview_d const *a, vsip_cvview_d const *b,
                vsip_cvview_d const *c, vsip_cvview_d const *r);

```

Description

This function performs an element-wise subtract-multiply operation on three input vectors. The operation performs element-wise computation:

$$r_i = (a_i - b_i) \cdot c_i$$

for all i from 0 to $n - 1$, where n is the length of the vectors.

Parameters

- vsip_dvview_p const* a: First input vector.
- vsip_dvview_p const* b: Second input vector.
- vsip_dvview_p const* c: Third input vector.
- vsip_dvview_p const* r: Output vector that will store the results.

4.4.7 vsip_dvsma_p - Vector Vector-Scalar-Multiply-Add

```

void vsip_vsma_f(vsip_vview_f const *a, vsip_scalar_f beta,
                vsip_vview_f const *c, vsip_vview_f const *r);
void vsip_vsma_d(vsip_vview_d const *a, vsip_scalar_d beta,
                vsip_vview_d const *c, vsip_vview_d const *r);
void vsip_cvsma_f(vsip_cvview_f const *a, vsip_cscalar_f beta,
                vsip_cvview_f const *c, vsip_cvview_f const *r);
void vsip_cvsma_d(vsip_cvview_d const *a, vsip_cscalar_d beta,
                vsip_cvview_d const *c, vsip_cvview_d const *r);

```

Description

This function performs an element-wise operation on two input vectors and a scalar constant. The operation performs element-wise computation:

$$r_i = \beta a_i + c_i$$

for all i from 0 to $n - 1$, where n is the length of the vectors.

Parameters

- vsip_dvview_p const* a: First input vector.
- vsip_dscalar_p beta: Scalar constant β .
- vsip_dvview_p const* c: Third input vector.
- vsip_dvview_p const* r: Output vector that will store the results.

4.4.8 vsip_dvsmsa_p - Vector Vector-Scalar-Multiply-Scalar-Add

```
void vsip_vsmsa_f(vsip_vview_f const *a, vsip_scalar_f beta,
                 vsip_scalar_f gamma, vsip_vview_f const *r);
void vsip_vsmsa_d(vsip_vview_d const *a, vsip_scalar_d beta,
                 vsip_scalar_d gamma, vsip_vview_d const *r);
void vsip_cvmsa_f(vsip_cvview_f const *a, vsip_cscalar_f beta,
                 vsip_cscalar_f gamma, vsip_cvview_f const *r);
void vsip_cvmsa_d(vsip_cvview_d const *a, vsip_cscalar_d beta,
                 vsip_cscalar_d gamma, vsip_cvview_d const *r);
```

Description

This function performs an element-wise operation on the input vector and two scalar constants. The operation performs element-wise computation:

$$r_i = \beta a_i + \gamma$$

for all i from 0 to $n - 1$, where n is the length of the vector.

Parameters

- vsip_dvview_p const* a: First input vector.
- vsip_dscalar_p beta: Scalar constant β .
- vsip_dscalar_p gamma: Scalar constant γ .
- vsip_dvview_p const* r: Output vector that will store the results.

4.5 Logical Operations

4.5.1 vsip_valltrue_p - Check if All Elements in Boolean Vector are True

```
vsip_scalar_bl vsip_valltrue_bl(const vsip_vview_bl *a);
```

Description

This function checks whether all elements in a boolean vector are true. It returns a single boolean value that is true if and only if every element in the input vector is true.

The function performs the following logical operation:

$$\text{result} = a_0 \wedge a_1 \wedge a_2 \wedge \dots \wedge a_{n-1}$$

where a_i are the elements of the input vector and n is the length of the vector.

Parameters

- `const vsip_vview_p * a`: Input boolean vector to check.

Return Value

- Returns true if all elements in the vector are true.
- Returns false if any element in the vector is false or if the vector is empty.

Example

```
vsip_vview_bl *conditions;
vsip_length n = 10;
vsip_scalar_bl all_valid;

// Create and initialize a boolean vector
conditions = vsip_vcreate_bl(n, VSIP_MEM_NONE);

// Set all elements to true (for demonstration)
vsip_vfill_bl(conditions, true);

// Check if all conditions are true
all_valid = vsip_valltrue_bl(conditions);
if (all_valid) {
    printf("All conditions are satisfied.\n");
} else {
    printf("Some conditions are not satisfied.\n");
}

// For a more practical example:
for (vsip_length i = 0; i < n; i++) {
    // Set based on some actual conditions in your algorithm
    vsip_vput_bl(conditions, i, (i % 2) == 0); // Only even indices are true
}

all_valid = vsip_valltrue_bl(conditions);
// all_valid will be false in this case

// Clean up
vsip_valldestroy_bl(conditions);
```


4.5.2 vsip_vanytrue_p - Check if Any Element in Boolean Vector is True

```
vsip_scalar_bl vsip_vanytrue_bl(const vsip_vview_bl *a);
```

Description

This function checks whether any element in a boolean vector is true. It returns a single boolean value that is true if at least one element in the input vector is true.

The function performs the following logical operation:

$$\text{result} = a_0 \vee a_1 \vee a_2 \vee \dots \vee a_{n-1}$$

where a_i are the elements of the input vector and n is the length of the vector.

Parameters

- `const vsip_vview_p * a`: Input boolean vector to check.

Return Value

- Returns true if at least one element in the vector is true.
- Returns false if all elements in the vector are false or if the vector is empty.

Example

```
vsip_vview_bl *flags;
vsip_length n = 100;
vsip_scalar_bl any_flag_set;

// Create and initialize a boolean vector
flags = vsip_vcreate_bl(n, VSIP_MEM_NONE);

// Set all elements to false initially
vsip_vfill_bl(flags, false);

// Set some flags based on your algorithm's conditions
// For example, set flag at index 42 to true
vsip_vput_bl(flags, 42, true);

// Check if any flag is set
any_flag_set = vsip_vanytrue_bl(flags);
if (any_flag_set) {
    printf("At least one flag is set. Processing required.\n");
    // Perform necessary processing for your application
} else {
    printf("No flags are set. Skipping processing.\n");
}

// For a more practical example with actual conditions:
for (vsip_length i = 0; i < n; i++) {
    // Set based on some actual conditions in your algorithm
    vsip_vput_bl(flags, i, (i % 7) == 0); // Set flags for indices divisible by 7
}

any_flag_set = vsip_vanytrue_bl(flags);
// any_flag_set will be true in this case

// Clean up
vsip_valldestroy_bl(flags);
```

4.5.3 vsip_vleq_p - Element-wise Equal Comparison

```
void vsip_vleq_f(const vsip_vview_f *a, const vsip_vview_f *b, const vsip_vview_bl *r);  
void vsip_vleq_d(const vsip_vview_d *a, const vsip_vview_d *b, const vsip_vview_bl *r);
```

Description

This function performs an element-wise *equal* comparison between two vectors, storing the results in a boolean vector. The operation performs element-wise comparison:

$$r_i = (a_i = b_i)$$

for all i from 0 to $n - 1$, where n is the length of the vectors, and r_i is a boolean value that is true if a_i is equal to b_i , and false otherwise.

Parameters

- `const vsip_vview_p* a`: First input vector.
- `const vsip_vview_p* b`: Second input vector.
- `const vsip_vview_bl* r`: Output boolean vector that will store the comparison results.

Notes

- All three vectors must have the same length.

4.5.4 vsip_vlne_p - Element-wise Not-Equal Comparison

```
void vsip_vlne_f(const vsip_vview_f *a, const vsip_vview_f *b, const vsip_vview_bl *r);
void vsip_vlne_d(const vsip_vview_d *a, const vsip_vview_d *b, const vsip_vview_bl *r);
```

Description

This function performs an element-wise *not-equal* comparison between two vectors, storing the results in a boolean vector. The operation performs element-wise comparison:

$$r_i = (a_i \neq b_i)$$

for all i from 0 to $n - 1$, where n is the length of the vectors, and r_i is a boolean value that is true if a_i is not equal to b_i , and false otherwise.

Parameters

- `const vsip_vview_p* a`: First input vector.
- `const vsip_vview_p* b`: Second input vector.
- `const vsip_vview_bl* r`: Output boolean vector that will store the comparison results.

Notes

- All three vectors must have the same length.

4.5.5 vsip_vllt_p - Element-wise Less-Than Comparison

```
void vsip_vllt_f(const vsip_vview_f *a, const vsip_vview_f *b, const vsip_vview_bl *r);
void vsip_vllt_d(const vsip_vview_d *a, const vsip_vview_d *b, const vsip_vview_bl *r);
```

Description

This function performs an element-wise *less-than* comparison between two vectors, storing the results in a boolean vector. The operation performs element-wise comparison:

$$r_i = (a_i < b_i)$$

for all i from 0 to $n - 1$, where n is the length of the vectors, and r_i is a boolean value that is true if a_i is less than b_i , and false otherwise.

Parameters

- const vsip_vview_p* a: First input vector.
- const vsip_vview_p* b: Second input vector.
- const vsip_vview_bl* r: Output boolean vector that will store the comparison results.

Notes

- All three vectors must have the same length.

4.5.6 vsip_vlle_p - Element-wise Lesser-Or-Equal-Than Comparison

```
void vsip_vlle_f(const vsip_vview_f *a, const vsip_vview_f *b, const vsip_vview_bl *r);
void vsip_vlle_d(const vsip_vview_d *a, const vsip_vview_d *b, const vsip_vview_bl *r);
```

Description

This function performs an element-wise *lesser-or-equal-than* comparison between two vectors, storing the results in a boolean vector. The operation performs element-wise comparison:

$$r_i = (a_i \leq b_i)$$

for all i from 0 to $n - 1$, where n is the length of the vectors, and r_i is a boolean value that is true if a_i is lesser or equal than b_i , and false otherwise.

Parameters

- `const vsip_vview_p* a`: First input vector.
- `const vsip_vview_p* b`: Second input vector.
- `const vsip_vview_bl* r`: Output boolean vector that will store the comparison results.

Notes

- All three vectors must have the same length.

4.5.7 vsip_vlgt_p - Element-wise Greater-Than Comparison

```
void vsip_vlgt_f(const vsip_vview_f *a, const vsip_vview_f *b, const vsip_vview_bl *r);
void vsip_vlgt_d(const vsip_vview_d *a, const vsip_vview_d *b, const vsip_vview_bl *r);
```

Description

This function performs an element-wise *greater-than* comparison between two vectors, storing the results in a boolean vector. The operation performs element-wise comparison:

$$r_i = (a_i > b_i)$$

for all i from 0 to $n-1$, where n is the length of the vectors, and r_i is a boolean value that is true if a_i is greater than b_i , and false otherwise.

Parameters

- const vsip_vview_p* a: First input vector.
- const vsip_vview_p* b: Second input vector.
- const vsip_vview_bl* r: Output boolean vector that will store the comparison results.

Notes

- All three vectors must have the same length.

4.5.8 vsip_vlge_p - Element-wise Greater-Or-Equal-Than Comparison

```
void vsip_vlge_f(const vsip_vview_f *a, const vsip_vview_f *b, const vsip_vview_bl *r);
void vsip_vlge_d(const vsip_vview_d *a, const vsip_vview_d *b, const vsip_vview_bl *r);
```

Description

This function performs an element-wise *greater-or-equal-than* comparison between two vectors, storing the results in a boolean vector. The operation performs element-wise comparison:

$$r_i = (a_i \geq b_i)$$

for all i from 0 to $n-1$, where n is the length of the vectors, and r_i is a boolean value that is true if a_i is greater or equal than b_i , and false otherwise.

Parameters

- `const vsip_vview_p* a`: First input vector.
- `const vsip_vview_p* b`: Second input vector.
- `const vsip_vview_bl* r`: Output boolean vector that will store the comparison results.

Notes

- All three vectors must have the same length.

4.6 Selection Operations

4.6.1 vsip_vclip_p - Clip Vector Elements Between Thresholds

```
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_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_d(const vsip_vview_d *a, vsip_scalar_d t1, vsip_scalar_d t2,
                 vsip_scalar_d c1, vsip_scalar_d c2, const vsip_vview_d *r);
```

Description

This function clips the elements of a vector between specified thresholds, replacing values outside the threshold range with corresponding clip values. The clipping operation is defined as:

$$r_i = \begin{cases} c1 & \text{if } a_i < t1 \\ a_i & \text{if } t1 \leq a_i \leq t2 \\ c2 & \text{if } a_i > t2 \end{cases}$$

for all i from 0 to $n - 1$, where n is the length of the vectors.

Parameters

- `const vsip_vview_p* a`: Input vector containing the elements to clip.
- `vsip_scalar_p t1`: Lower threshold value.
- `vsip_scalar_p t2`: Upper threshold value.
- `vsip_scalar_p c1`: Value to substitute for elements below the lower threshold.
- `vsip_scalar_p c2`: Value to substitute for elements above the upper threshold.
- `const vsip_vview_p* r`: Output vector that will store the clipped results.

Example

```
vsip_vview_f *signal, *clipped_signal;
vsip_length n = 1024;

// Create vectors
signal = vsip_vcreate_f(n, VSIP_MEM_NONE);
clipped_signal = vsip_vcreate_f(n, VSIP_MEM_NONE);

// Initialize signal with some values (e.g., sine wave with noise)
for (vsip_length i = 0; i < n; i++) {
    float val = 10.0f * sin(2 * M_PI * i / n) + 5.0f * ((float)rand()/RAND_MAX - 0.5f);
    vsip_vput_f(signal, i, val);
}

// Clip the signal between -5.0 and 5.0, replacing outliers with these bounds
vsip_vclip_f(signal, -5.0f, 5.0f, -5.0f, 5.0f, clipped_signal);

// The clipped_signal vector now contains the original values where they were
// within [-5.0, 5.0], and -5.0 or 5.0 where they were outside this range

// Clean up
vsip_valldestroy_f(signal);
vsip_valldestroy_f(clipped_signal);
```

Notes

- The input and output vectors must have the same length.
- The thresholds and clip values can be in any order, but typically $t1 \leq t2$ and $c1 \leq c2$.

4.6.2 vsip_vinvclip_p - Inverse Clip Vector Elements

```
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_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_d(const vsip_vview_d *a, vsip_scalar_d t1, vsip_scalar_d t2,
                    vsip_scalar_d t3, vsip_scalar_d c1, vsip_scalar_d c2,
                    const vsip_vview_d *r);
```

Description

This function performs an inverse clipping operation on a vector, replacing values within a specified range with corresponding clip values while preserving values outside this range. The inverse clipping operation is defined as:

$$r_i = \begin{cases} a_i & \text{if } a_i < t1 \text{ or } a_i > t3 \\ c1 & \text{if } t1 \leq a_i < t2 \\ c2 & \text{if } t2 \leq a_i \leq t3 \end{cases}$$

for all i from 0 to $n - 1$, where n is the length of the vectors.

Parameters

- `const vsip_vview_p* a`: Input vector containing the elements to process.
- `vsip_scalar_p t1`: Lower threshold of the clipping range.
- `vsip_scalar_p t2`: Middle threshold separating the two clip values.
- `vsip_scalar_p t3`: Upper threshold of the clipping range.
- `vsip_scalar_p c1`: Value to substitute for elements in the lower part of the clipping range.
- `vsip_scalar_p c2`: Value to substitute for elements in the upper part of the clipping range.
- `const vsip_vview_p* r`: Output vector that will store the processed results.

Example

```
vsip_vview_f *signal, *processed_signal;
vsip_length n = 1024;

// Create vectors
signal = vsip_vcreate_f(n, VSIP_MEM_NONE);
processed_signal = vsip_vcreate_f(n, VSIP_MEM_NONE);

// Initialize signal with some values (e.g., audio signal with noise)
for (vsip_length i = 0; i < n; i++) {
    float val = 10.0f * sin(2 * M_PI * i / n) + 3.0f * sin(20 * 2 * M_PI * i / n);
    vsip_vput_f(signal, i, val);
}

// Apply inverse clipping to remove values in the [-1.0, 1.0] range,
// replacing them with -2.0 and 2.0 respectively
vsip_vinvclip_f(signal, -1.0f, 0.0f, 1.0f, -2.0f, 2.0f, processed_signal);

// The processed_signal vector now contains:
// - Original values where they were < -1.0 or > 1.0
// - -2.0 where values were between -1.0 and 0.0
// - 2.0 where values were between 0.0 and 1.0
```

```
// Clean up  
vsip_valldestroy_f(signal);  
vsip_valldestroy_f(processed_signal);
```

Notes

- The input and output vectors must have the same length.
- The thresholds should typically satisfy $t1 \leq t2 \leq t3$.

4.6.3 vsip_vindexbool - Find Indices of True Elements in Boolean Vector

```
vsip_length vsip_vindexbool(const vsip_vview_bl *a, vsip_vview_vi *index);
```

Description

This function finds the indices of all true elements in a boolean vector and stores them in an integer index vector. It returns the number of true elements found.

The function scans the input boolean vector *a* and records the positions of all elements that are true in the output index vector. The function returns the count of true elements found.

Parameters

- `const vsip_vview_bl* a`: Input boolean vector to search.
- `vsip_vview_vi* index`: Output integer vector that will store the indices of true elements. This vector must be large enough to hold all potential true indices (i.e., its length should be at least equal to the length of the input boolean vector).

Return Value

- Returns the number of true elements found in the input vector.
- Returns 0 if no true elements are found or if the input vector is empty.

Example

```
vsip_vview_bl *conditions;
vsip_vview_vi *indices;
vsip_length n = 100;
vsip_length true_count;

// Create boolean vector
conditions = vsip_vcreate_bl(n, VSIP_MEM_NONE);

// Create index vector (same length as conditions)
indices = vsip_vcreate_vi(n, VSIP_MEM_NONE);

// Set some conditions to true (for example, every 5th element)
for (vsip_length i = 0; i < n; i++) {
    vsip_vput_bl(conditions, i, (i % 5) == 0);
}

// Find indices of true elements
true_count = vsip_vindexbool(conditions, indices);

printf("Found %lu true elements at positions:\n", true_count);
for (vsip_length i = 0; i < true_count; i++) {
    printf("%ld ", vsip_vget_vi(indices, i));
}
printf("\n");

// Use the indices for further processing in your algorithms
// For example, you could use these indices to select specific elements
// from another vector that corresponds to your conditions

// Clean up
vsip_valldestroy_bl(conditions);
vsip_valldestroy_vi(indices);
```

4.6.4 vsip_vmaxval_p - Find the Maximum Value in a Vector View

```
vsip_scalar_f vsip_vmaxval_f(const vsip_vview_f* a, vsip_index* j);  
vsip_scalar_d vsip_vmaxval_d(const vsip_vview_d* a, vsip_index* j);
```

Description

This function finds the maximum value in the vector view `a` and returns it. The index of the maximum value is stored in the variable pointed to by `j`.

Parameters

- `const vsip_vview_p* a`: Pointer to the vector view.
- `vsip_index* j`: Pointer to a variable where the index of the maximum value will be stored.

Return Value

- The maximum value in the vector view.

Example

```
vsip_vview_f *vector_view;  
vsip_index index;  
vsip_scalar_f max_value;  
  
// Assuming vector_view has been properly initialized  
max_value = vsip_vmaxval_f(vector_view, &index);
```

4.6.5 vsip_vminval_p - Find the Minimum Value in a Vector View

```
vsip_scalar_f vsip_vminval_f(const vsip_vview_f* a, vsip_index* j);  
vsip_scalar_d vsip_vminval_d(const vsip_vview_d* a, vsip_index* j);
```

Description

This function finds the minimum value in the vector view `a` and returns it. The index of the minimum value is stored in the variable pointed to by `j`.

Parameters

- `const vsip_vview_p* a`: Pointer to the vector view.
- `vsip_index* j`: Pointer to a variable where the index of the minimum value will be stored.

Return Value

- The minimum value in the vector view.

Example

```
vsip_vview_f *vector_view;  
vsip_index index;  
vsip_scalar_f min_value;  
  
// Assuming vector_view has been properly initialized  
min_value = vsip_vminval_f(vector_view, &index);
```

4.6.6 vsip_vmax_p - Element-wise Maximum of Two Vector Views

```
void vsip_vmax_f(const vsip_vview_f* a, const vsip_vview_f* b, const vsip_vview_f* w);  
void vsip_vmax_d(const vsip_vview_d* a, const vsip_vview_d* b, const vsip_vview_d* w);
```

Description

This function performs element-wise maximum comparison of the vector views *a* and *b* and stores the result in the vector view *w*. Each element in *w* is the maximum of the corresponding elements in *a* and *b*.

$$w_i = \max(a_i, b_i)$$

Parameters

- `const vsip_vview_p* a`: Pointer to the first source vector view.
- `const vsip_vview_p* b`: Pointer to the second source vector view.
- `const vsip_vview_p* w`: Pointer to the destination vector view.

Example

```
vsip_vview_f *vector_view_a;  
vsip_vview_f *vector_view_b;  
vsip_vview_f *result_vector_view;
```

```
// Assuming vector_view_a, vector_view_b, and result_vector_view have been properly initialized  
vsip_vmax_f(vector_view_a, vector_view_b, result_vector_view);
```

4.6.7 vsip_vmin_p - Element-wise Minimum of Two Vector Views

```
void vsip_vmin_f(const vsip_vview_f* a, const vsip_vview_f* b, const vsip_vview_f* w);  
void vsip_vmin_d(const vsip_vview_d* a, const vsip_vview_d* b, const vsip_vview_d* w);
```

Description

This function performs element-wise minimum comparison of the vector views a and b and stores the result in the vector view w. Each element in w is the minimum of the corresponding elements in a and b.

$$w_i = \min(a_i, b_i)$$

Parameters

- const vsip_vview_p* a: Pointer to the first source vector view.
- const vsip_vview_p* b: Pointer to the second source vector view.
- const vsip_vview_p* w: Pointer to the destination vector view.

Example

```
vsip_vview_f *vector_view_a;  
vsip_vview_f *vector_view_b;  
vsip_vview_f *result_vector_view;
```

```
// Assuming vector_view_a, vector_view_b, and result_vector_view have been properly initialized  
vsip_vmin_f(vector_view_a, vector_view_b, result_vector_view);
```


4.6.8 vsip_vmaxmg_p - Element-wise Maximum of Magnitudes

```
void vsip_vmaxmg_f(const vsip_vview_f *a, const vsip_vview_f *b, const vsip_vview_f *r);
void vsip_vmaxmg_d(const vsip_vview_d *a, const vsip_vview_d *b, const vsip_vview_d *r);
```

Description

This function computes the element-wise maximum of the magnitudes (absolute values) of corresponding elements from two vectors. The operation performs element-wise comparison of magnitudes:

$$r_i = \max(|a_i|, |b_i|)$$

for all i from 0 to $n - 1$, where n is the length of the vectors.

Parameters

- `const vsip_vview_p* a`: First input vector.
- `const vsip_vview_p* b`: Second input vector.
- `const vsip_vview_p* r`: Output vector that will store the results.

Example

```
vsip_vview_f *signal1, *signal2, *result;
vsip_length n = 1024;

// Create vectors
signal1 = vsip_vcreate_f(n, VSIP_MEM_NONE);
signal2 = vsip_vcreate_f(n, VSIP_MEM_NONE);
result = vsip_vcreate_f(n, VSIP_MEM_NONE);

// Initialize vectors with some signal data
// For example, two different signal measurements
for (vsip_length i = 0; i < n; i++) {
    vsip_vput_f(signal1, i, 5.0f * sin(2 * M_PI * i / n));
    vsip_vput_f(signal2, i, 3.0f * cos(2 * M_PI * i / n) + 2.0f);
}

// Compute element-wise maximum of magnitudes
vsip_vmaxmg_f(signal1, signal2, result);

// The result vector now contains the maximum magnitude at each position
// from the two input signals

// Clean up
vsip_valldestroy_f(signal1);
vsip_valldestroy_f(signal2);
vsip_valldestroy_f(result);
```

Notes

- All three vectors must have the same length.

4.6.9 vsip_vminmg_p - Element-wise Minimum of Magnitudes

```
void vsip_vminmg_f(const vsip_vview_f *a, const vsip_vview_f *b, const vsip_vview_f *r);
void vsip_vminmg_d(const vsip_vview_d *a, const vsip_vview_d *b, const vsip_vview_d *r);
```

Description

This function computes the element-wise minimum of the magnitudes (absolute values) of corresponding elements from two vectors. The operation performs element-wise comparison of magnitudes:

$$r_i = \min(|a_i|, |b_i|)$$

for all i from 0 to $n - 1$, where n is the length of the vectors.

Parameters

- const vsip_vview_p* a: First input vector.
- const vsip_vview_p* b: Second input vector.
- const vsip_vview_p* r: Output vector that will store the results.

Example

```
vsip_vview_f *signal1, *signal2, *result;
vsip_length n = 1024;

// Create vectors
signal1 = vsip_vcreate_f(n, VSIP_MEM_NONE);
signal2 = vsip_vcreate_f(n, VSIP_MEM_NONE);
result = vsip_vcreate_f(n, VSIP_MEM_NONE);

// Initialize vectors with some signal data
// For example, two different signal measurements
for (vsip_length i = 0; i < n; i++) {
    vsip_vput_f(signal1, i, 5.0f * sin(2 * M_PI * i / n));
    vsip_vput_f(signal2, i, 3.0f * cos(2 * M_PI * i / n) + 2.0f);
}

// Compute element-wise minimum of magnitudes
vsip_vminmg_f(signal1, signal2, result);

// The result vector now contains the maximum magnitude at each position
// from the two input signals

// Clean up
vsip_valldestroy_f(signal1);
vsip_valldestroy_f(signal2);
vsip_valldestroy_f(result);
```

Notes

- All three vectors must have the same length.

4.6.10 vsip_vmaxmgval_p - Find Maximum Magnitude Value in Vector

```
vsip_scalar_f vsip_vmaxmgval_f(const vsip_vview_f *a, vsip_scalar_vi *index);
vsip_scalar_d vsip_vmaxmgval_d(const vsip_vview_d *a, vsip_scalar_vi *index);
```

Description

This function finds the maximum magnitude (absolute) value in a vector and returns its value while storing its index in the provided output parameter.

The function scans the input vector *a* and finds the element with the largest absolute value, returning this value and storing its index in the output parameter.

Parameters

- `const vsip_vview_p* a`: Input vector to search for the maximum magnitude value.
- `vsip_scalar_vi* index`: Pointer to an integer that will store the index of the element with the maximum magnitude.

Return Value

- Returns the maximum magnitude (absolute) value found in the vector.

Example

```
vsip_vview_f *signal;
vsip_scalar_f max_magnitude;
vsip_scalar_vi max_index;
vsip_length n = 1024;

// Create vector
signal = vsip_vcreate_f(n, VSIP_MEM_NONE);

// Initialize signal with some values (e.g., a sine wave with noise)
for (vsip_length i = 0; i < n; i++) {
    float val = 10.0f * sin(2 * M_PI * i / n) + 2.0f * ((float)rand()/RAND_MAX - 0.5f);
    vsip_vput_f(signal, i, val);
}

// Add a spike at position 42 for demonstration
vsip_vput_f(signal, 42, 15.0f);

// Find the maximum magnitude value and its index
max_magnitude = vsip_vmaxmgval_f(signal, &max_index);

printf("Maximum magnitude value: %.4f\n", max_magnitude);
printf("Found at index: %ld\n", max_index);

// Clean up
vsip_valldestroy_f(signal);
```

4.6.11 vsip_vminmgval_p - Find Minimum Magnitude Value in Vector

```
vsip_scalar_f vsip_vminmgval_f(const vsip_vview_f *a, vsip_scalar_vi *index);
vsip_scalar_d vsip_vminmgval_d(const vsip_vview_d *a, vsip_scalar_vi *index);
```

Description

This function finds the minimum magnitude (absolute) value in a vector and returns its value while storing its index in the provided output parameter.

The function scans the input vector *a* and finds the element with the smallest absolute value, returning this value and storing its index in the output parameter.

Parameters

- `const vsip_vview_p* a`: Input vector to search for the minimum magnitude value.
- `vsip_scalar_vi* index`: Pointer to an integer that will store the index of the element with the minimum magnitude.

Return Value

- Returns the minimum magnitude (absolute) value found in the vector.

Example

```
vsip_vview_f *signal;
vsip_scalar_f min_magnitude;
vsip_scalar_vi min_index;
vsip_length n = 1024;

// Create vector
signal = vsip_vcreate_f(n, VSIP_MEM_NONE);

// Initialize signal with some values (e.g., a sine wave with noise)
for (vsip_length i = 0; i < n; i++) {
    float val = 10.0f * sin(2 * M_PI * i / n) + 0.1f * ((float)rand()/RAND_MAX - 0.5f);
    vsip_vput_f(signal, i, val);
}

// Add a near-zero value at position 123 for demonstration
vsip_vput_f(signal, 123, 0.001f);

// Find the minimum magnitude value and its index
min_magnitude = vsip_vminmgval_f(signal, &min_index);

printf("Minimum magnitude value: %.6f\n", min_magnitude);
printf("Found at index: %ld\n", min_index);

// Clean up
vsip_valldestroy_f(signal);
```

4.6.12 vsip_vcmaxmg_p - Element-wise Maximum of Complex Vector Magnitudes

```
void vsip_vcmaxmg_f(const vsip_cvview_f *a, const vsip_cvview_f *b, const vsip_vview_f *r);
void vsip_vcmaxmg_d(const vsip_cvview_d *a, const vsip_cvview_d *b, const vsip_vview_d *r);
```

Description

This function computes the element-wise maximum of the magnitudes of corresponding elements from two complex vectors, storing the results in a real output vector. The operation performs element-wise comparison of complex magnitudes:

$$r_i = \max(|a_i|, |b_i|)$$

for all i from 0 to $n-1$, where n is the length of the vectors, and $|a_i|$ and $|b_i|$ are the magnitudes of the complex numbers.

Parameters

- `const vsip_cvview_p*` `a`: First input complex vector.
- `const vsip_cvview_p*` `b`: Second input complex vector.
- `const vsip_vview_p*` `r`: Output real vector that will store the maximum magnitude results.

Example

```
vsip_cvview_f *signal1, *signal2;
vsip_vview_f *max_magnitudes;
vsip_length n = 1024;

// Create vectors
signal1 = vsip_cvcreate_f(n, VSIP_MEM_NONE);
signal2 = vsip_cvcreate_f(n, VSIP_MEM_NONE);
max_magnitudes = vsip_vcreate_f(n, VSIP_MEM_NONE);

// Initialize complex vectors with some signal data
for (vsip_length i = 0; i < n; i++) {
    // Create two different complex signals
    float angle1 = 2 * M_PI * i / n;
    float angle2 = 2 * M_PI * i / n + M_PI/4;
    vsip_cscalar_f val1 = VSIP_CMPLX_F(5.0f * cos(angle1), 5.0f * sin(angle1));
    vsip_cscalar_f val2 = VSIP_CMPLX_F(3.0f * cos(angle2), 3.0f * sin(angle2));

    vsip_cvput_f(signal1, i, val1);
    vsip_cvput_f(signal2, i, val2);
}

// Compute element-wise maximum of magnitudes
vsip_vcmaxmg_f(signal1, signal2, max_magnitudes);

// The max_magnitudes vector now contains the maximum magnitude at each position
// from the two input complex signals

// Clean up
vsip_cvalldestroy_f(signal1);
vsip_cvalldestroy_f(signal2);
vsip_valldestroy_f(max_magnitudes);
```

Notes

- All three vectors must have the same length.

4.6.13 vsip_vcminmg_p - Element-wise Minimum of Complex Vector Magnitudes

```
void vsip_vcminmg_f(const vsip_cvview_f *a, const vsip_cvview_f *b, const vsip_vview_f *r);
void vsip_vcminmg_d(const vsip_cvview_d *a, const vsip_cvview_d *b, const vsip_vview_d *r);
```

Description

This function computes the element-wise minimum of the magnitudes of corresponding elements from two complex vectors, storing the results in a real output vector. The operation performs element-wise comparison of complex magnitudes:

$$r_i = \min(|a_i|, |b_i|)$$

for all i from 0 to $n-1$, where n is the length of the vectors, and $|a_i|$ and $|b_i|$ are the magnitudes of the complex numbers.

Parameters

- `const vsip_cvview_p* a`: First input complex vector.
- `const vsip_cvview_p* b`: Second input complex vector.
- `const vsip_vview_p* r`: Output real vector that will store the minimum magnitude results.

Example

```
vsip_cvview_f *reference, *noisy_signal;
vsip_vview_f *min_magnitudes;
vsip_length n = 1024;

// Create vectors
reference = vsip_cvcreate_f(n, VSIP_MEM_NONE);
noisy_signal = vsip_cvcreate_f(n, VSIP_MEM_NONE);
min_magnitudes = vsip_vcreate_f(n, VSIP_MEM_NONE);

// Initialize complex vectors with signal data
for (vsip_length i = 0; i < n; i++) {
    // Create a reference signal and a noisy version
    float angle = 2 * M_PI * i / n;
    vsip_cscalar_f ref_val = VSIP_CMPLX_F(5.0f * cos(angle), 5.0f * sin(angle));

    // Add some noise to create the noisy signal
    float noise_real = 0.5f * ((float)rand())/RAND_MAX - 0.5f;
    float noise_imag = 0.5f * ((float)rand())/RAND_MAX - 0.5f;
    vsip_cscalar_f noisy_val = VSIP_CMPLX_F(ref_val.r + noise_real, ref_val.i + noise_imag);

    vsip_cvput_f(reference, i, ref_val);
    vsip_cvput_f(noisy_signal, i, noisy_val);
}

// Compute element-wise minimum of magnitudes
vsip_vcminmg_f(reference, noisy_signal, min_magnitudes);

// The min_magnitudes vector now contains the minimum magnitude at each position
// from the two input complex signals, which can be useful for noise analysis

// Clean up
vsip_cvalldestroy_f(reference);
vsip_cvalldestroy_f(noisy_signal);
vsip_valldestroy_f(min_magnitudes);
```

Notes

- All three vectors must have the same length.

4.6.14 vsip_vcmaxmgsqval_p - Find Maximum Magnitude Squared Value in Complex Vector

```
vsip_scalar_f vsip_vcmaxmgsqval_f(const vsip_cvview_f *a, vsip_scalar_vi *index);
vsip_scalar_d vsip_vcmaxmgsqval_d(const vsip_cvview_d *a, vsip_scalar_vi *index);
```

Description

This function finds the maximum magnitude squared value in a complex vector and returns its value while storing its index in the provided output parameter.

The function scans the input complex vector a and finds the element with the largest magnitude squared value (real part squared plus imaginary part squared), returning this value and storing its index in the output parameter.

The magnitude squared is calculated as:

$$|a_i|^2 = \text{real}(a_i)^2 + \text{imag}(a_i)^2$$

Parameters

- `const vsip_cvview_p* a`: Input complex vector to search for the maximum magnitude squared value.
- `vsip_scalar_vi* index`: Pointer to an integer that will store the index of the element with the maximum magnitude squared.

Return Value

- Returns the maximum magnitude squared value found in the vector.

Example

```
vsip_cvview_f *spectrum;
vsip_scalar_f max_magnitude_sq;
vsip_scalar_vi max_index;
vsip_length n = 1024;

// Create complex vector for spectrum data
spectrum = vsip_cvcreate_f(n, VSIP_MEM_NONE);

// Initialize with some complex spectrum data (e.g., FFT results)
for (vsip_length i = 0; i < n; i++) {
    // Simulate a spectrum with a peak at position 42
    float magnitude = (i == 42) ? 10.0f : 0.1f + 0.9f * (float)rand()/RAND_MAX;
    float phase = 2 * M_PI * (float)rand()/RAND_MAX;
    vsip_cscalar_f val = VSIP_CMPLX_F(magnitude * cos(phase), magnitude * sin(phase));
    vsip_cvput_f(spectrum, i, val);
}

// Find the maximum magnitude squared value and its index
max_magnitude_sq = vsip_vcmaxmgsqval_f(spectrum, &max_index);

printf("Maximum magnitude squared value: %.4f\n", max_magnitude_sq);
printf("Found at index: %ld\n", max_index);
printf("Actual magnitude: %.4f\n", sqrt(max_magnitude_sq));

// Clean up
vsip_cvalldestroy_f(spectrum);
```

4.6.15 vsip_vcminmgsqval_p - Find Minimum Magnitude Squared Value in Complex Vector

```
vsip_scalar_f vsip_vcminmgsqval_f(const vsip_cvview_f *a, vsip_scalar_vi *index);
vsip_scalar_d vsip_vcminmgsqval_d(const vsip_cvview_d *a, vsip_scalar_vi *index);
```

Description

This function finds the minimum magnitude squared value in a complex vector and returns its value while storing its index in the provided output parameter.

The function scans the input complex vector a and finds the element with the smallest magnitude squared value (real part squared plus imaginary part squared), returning this value and storing its index in the output parameter.

The magnitude squared is calculated as:

$$|a_i|^2 = \text{real}(a_i)^2 + \text{imag}(a_i)^2$$

Parameters

- `const vsip_cvview_p* a`: Input complex vector to search for the minimum magnitude squared value.
- `vsip_scalar_vi* index`: Pointer to an integer that will store the index of the element with the minimum magnitude squared.

Return Value

- Returns the minimum magnitude squared value found in the vector.

Example

```
vsip_cvview_f *spectrum;
vsip_scalar_f min_magnitude_sq;
vsip_scalar_vi min_index;
vsip_length n = 1024;

// Create complex vector for spectrum data
spectrum = vsip_cvcreate_f(n, VSIP_MEM_NONE);

// Initialize with some complex spectrum data (e.g., FFT results)
for (vsip_length i = 0; i < n; i++) {
    // Simulate a spectrum with mostly small values and one very small value
    float magnitude = (i == 123) ? 0.001f : 0.1f + 0.9f * (float)rand()/RAND_MAX;
    float phase = 2 * M_PI * (float)rand()/RAND_MAX;
    vsip_cscalar_f val = VSIP_CMPLX_F(magnitude * cos(phase), magnitude * sin(phase));
    vsip_cvput_f(spectrum, i, val);
}

// Find the minimum magnitude squared value and its index
min_magnitude_sq = vsip_vcminmgsqval_f(spectrum, &min_index);

printf("Minimum magnitude squared value: %.6f\n", min_magnitude_sq);
printf("Found at index: %ld\n", min_index);
printf("Actual magnitude: %.6f\n", sqrt(min_magnitude_sq));

// Clean up
vsip_cvalldestroy_f(spectrum);
```


4.7 Bitwise and Boolean Logical Operations

4.7.1 vsip_vnot_p - Boolean Vector Logical NOT

```
void vsip_vnot_bl(const vsip_vview_bl *a, const vsip_vview_bl *r);
void vsip_vnot_i(const vsip_vview_i *a, const vsip_vview_i *r);
```

Description

This function performs a logical NOT operation between corresponding elements of boolean vector a , storing the result in the output vector r . The operation performs element-wise logical NOT:

$$r_i = \neg a_i$$

for all i from 0 to $n - 1$, where n is the length of the vectors.

Parameters

- const vsip_vview_p* a: Input boolean vector.
- const vsip_vview_p* r: Output boolean vector that will store the result.

Example

```
vsip_vview_bl *a, *r;
vsip_length n = 10;

// Create boolean vectors
a = vsip_vcreate_bl(n, VSIP_MEM_NONE);
r = vsip_vcreate_bl(n, VSIP_MEM_NONE);

// Initialize vectors with some boolean values
// For example, set alternating true/false patterns
for (vsip_length i = 0; i < n; i++) {
    vsip_vput_bl(a, i, (i % 2) == 0); // true for even indices
}

// Perform logical AND operation
vsip_vnot_bl(a, r);

// The result vector r will now contain true only where both
// a and b had true values (indices 0, 6)

// Clean up
vsip_valldestroy_bl(a);
vsip_valldestroy_bl(r);
```

4.7.2 vsip_vand_p - Boolean Vector Logical AND

```
void vsip_vand_bl(const vsip_vview_bl *a, const vsip_vview_bl *b, const vsip_vview_bl *r);
void vsip_vand_i(const vsip_vview_i *a, const vsip_vview_i *b, const vsip_vview_i *r);
```

Description

This function performs a logical AND operation between corresponding elements of two boolean vectors a and b , storing the result in the output vector r . The operation performs element-wise logical AND:

$$r_i = a_i \wedge b_i$$

for all i from 0 to $n - 1$, where n is the length of the vectors.

Parameters

- `const vsip_vview_p* a`: First input boolean vector.
- `const vsip_vview_p* b`: Second input boolean vector.
- `const vsip_vview_p* r`: Output boolean vector that will store the result.

Example

```
vsip_vview_bl *a, *b, *r;
vsip_length n = 10;

// Create boolean vectors
a = vsip_vcreate_bl(n, VSIP_MEM_NONE);
b = vsip_vcreate_bl(n, VSIP_MEM_NONE);
r = vsip_vcreate_bl(n, VSIP_MEM_NONE);

// Initialize vectors with some boolean values
// For example, set alternating true/false patterns
for (vsip_length i = 0; i < n; i++) {
    vsip_vput_bl(a, i, (i % 2) == 0); // true for even indices
    vsip_vput_bl(b, i, (i % 3) == 0); // true for indices divisible by 3
}

// Perform logical AND operation
vsip_vand_bl(a, b, r);

// The result vector r will now contain true only where both
// a and b had true values (indices 0, 6)

// Clean up
vsip_valldestroy_bl(a);
vsip_valldestroy_bl(b);
vsip_valldestroy_bl(r);
```

4.7.3 vsip_vor_p - Boolean Vector Logical OR

```
void vsip_vor_bl(const vsip_vview_bl *a, const vsip_vview_bl *b, const vsip_vview_bl *r);
void vsip_vor_i(const vsip_vview_i *a, const vsip_vview_i *b, const vsip_vview_i *r);
```

Description

This function performs a logical OR operation between corresponding elements of two boolean vectors a and b , storing the result in the output vector r . The operation performs element-wise logical OR:

$$r_i = a_i \vee b_i$$

for all i from 0 to $n - 1$, where n is the length of the vectors.

Parameters

- const vsip_vview_p* a: First input boolean vector.
- const vsip_vview_p* b: Second input boolean vector.
- const vsip_vview_p* r: Output boolean vector that will store the result.

Example

```
vsip_vview_bl *a, *b, *r;
vsip_length n = 10;

// Create boolean vectors
a = vsip_vcreate_bl(n, VSIP_MEM_NONE);
b = vsip_vcreate_bl(n, VSIP_MEM_NONE);
r = vsip_vcreate_bl(n, VSIP_MEM_NONE);

// Initialize vectors with some boolean values
// For example, set alternating true/false patterns
for (vsip_length i = 0; i < n; i++) {
    vsip_vput_bl(a, i, (i % 2) == 0); // true for even indices
    vsip_vput_bl(b, i, (i % 3) == 0); // true for indices divisible by 3
}

// Perform logical AND operation
vsip_vor_bl(a, b, r);

// The result vector r will now contain true only where both
// a and b had true values (indices 0, 6)

// Clean up
vsip_valldestroy_bl(a);
vsip_valldestroy_bl(b);
vsip_valldestroy_bl(r);
```

4.7.4 vsip_vxor_p - Boolean Vector Logical XOR

```
void vsip_vxor_bl(const vsip_vview_bl *a, const vsip_vview_bl *b, const vsip_vview_bl *r);
void vsip_vxor_i(const vsip_vview_i *a, const vsip_vview_i *b, const vsip_vview_i *r);
```

Description

This function performs a logical XOR operation between corresponding elements of two boolean vectors a and b , storing the result in the output vector r . The operation performs element-wise logical XOR:

$$r_i = a_i \oplus b_i$$

for all i from 0 to $n - 1$, where n is the length of the vectors.

Parameters

- const vsip_vview_p* a: First input boolean vector.
- const vsip_vview_p* b: Second input boolean vector.
- const vsip_vview_p* r: Output boolean vector that will store the result.

Example

```
vsip_vview_bl *a, *b, *r;
vsip_length n = 10;

// Create boolean vectors
a = vsip_vcreate_bl(n, VSIP_MEM_NONE);
b = vsip_vcreate_bl(n, VSIP_MEM_NONE);
r = vsip_vcreate_bl(n, VSIP_MEM_NONE);

// Initialize vectors with some boolean values
// For example, set alternating true/false patterns
for (vsip_length i = 0; i < n; i++) {
    vsip_vput_bl(a, i, (i % 2) == 0); // true for even indices
    vsip_vput_bl(b, i, (i % 3) == 0); // true for indices divisible by 3
}

// Perform logical AND operation
vsip_vxor_bl(a, b, r);

// The result vector r will now contain true only where both
// a and b had true values (indices 0, 6)

// Clean up
vsip_valldestroy_bl(a);
vsip_valldestroy_bl(b);
vsip_valldestroy_bl(r);
```

4.8 Element Generation Functions

4.8.1 vsip_dvfill_p - Fill a Vector View with a Scalar Value

```
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_vfill_d(vsip_scalar_d alpha, const vsip_vview_d* r);

void vsip_cvfill_f(vsip_cscalar_f alpha, const vsip_cvview_f* r);
void vsip_cvfill_d(vsip_cscalar_d alpha, const vsip_cvview_d* r);
```

Description

This function fills the vector view `r` with the scalar value `alpha`.

$$\mathbf{r} = \alpha$$

Parameters

- `vsip_scalar_p alpha`: The scalar value to fill the vector view with.
- `const vsip_vview_p* r`: Pointer to the destination vector view.

Example

```
vsip_vview_f *vector_view;
vsip_scalar_f scalar_value = 2.0;

// Assuming vector_view has been properly initialized
vsip_vfill_f(scalar_value, vector_view);
```

4.8.2 vsip_vramp_p - Fill a Vector View with a Ramp

```
void vsip_vramp_i(vsip_scalar_i z, vsip_scalar_i d, const vsip_vview_i* r);  
void vsip_vramp_f(vsip_scalar_f z, vsip_scalar_f d, const vsip_vview_f* r);  
void vsip_vramp_d(vsip_scalar_d z, vsip_scalar_d d, const vsip_vview_d* r);
```

Description

This function fills the vector view `r` with a ramp starting at `z` and incrementing by `d`.

$$r_i = z + di$$

Parameters

- `vsip_scalar_p z`: The starting value of the ramp.
- `vsip_scalar_p d`: The increment value of the ramp.
- `const vsip_vview_p* r`: Pointer to the destination vector view.

Example

```
vsip_vview_f *vector_view;  
vsip_scalar_f start_value = 0.0;  
vsip_scalar_f increment = 1.0;  
  
// Assuming vector_view has been properly initialized  
vsip_vramp_f(start_value, increment, vector_view);
```


4.9 Copying Functions

4.9.1 vsip_dvcopy_p_p - Copy Vector Views

```

void vsip_vcopy_f_f(const vsip_vview_f* a, const vsip_vview_f* r);
void vsip_vcopy_d_d(const vsip_vview_d* a, const vsip_vview_d* r);
void vsip_vcopy_i_i(const vsip_vview_i* a, const vsip_vview_i* r);
void vsip_vcopy_i_f(const vsip_vview_i* a, const vsip_vview_f* r);
void vsip_vcopy_i_d(const vsip_vview_i* a, const vsip_vview_d* r);
void vsip_vcopy_f_i(const vsip_vview_f* a, const vsip_vview_i* r);
void vsip_vcopy_d_i(const vsip_vview_d* a, const vsip_vview_i* r);
void vsip_cvcopy_f_f(const vsip_cvview_f* a, const vsip_cvview_f* r);
void vsip_cvcopy_d_d(const vsip_cvview_d* a, const vsip_cvview_d* r);
void vsip_vcopy_vi_vi(const vsip_vview_vi* a, const vsip_vview_vi* r);
void vsip_vcopy_i_vi(const vsip_vview_i* a, const vsip_vview_vi* r);
void vsip_vcopy_vi_i(const vsip_vview_vi* a, const vsip_vview_i* r);
void vsip_vcopy_mi_mi(const vsip_vview_mi* a, const vsip_vview_mi* r);
void vsip_vcopy_bl_bl(const vsip_vview_bl* a, const vsip_vview_bl* r);
void vsip_vcopy_bl_f(const vsip_vview_bl* a, const vsip_vview_f* r);
void vsip_vcopy_f_bl(const vsip_vview_f* a, const vsip_vview_bl* r);

```

Description

These functions copy the contents of one vector view to another. The source and destination vector views can be of different types (float or integer), and the functions handle the necessary type conversions.

Parameters

- `const vsip_dvview_p* a`: Pointer to the source vector view.
- `const vsip_dvview_p* r`: Pointer to the destination vector view.

Functions

- `vsip_vcopy_f_f`: Copies from a float vector view to another float vector view.
- `vsip_vcopy_d_d`: Copies from a double vector view to another double vector view.
- `vsip_vcopy_i_i`: Copies from an integer vector view to another integer vector view.
- `vsip_vcopy_i_f`: Copies from an integer vector view to a float vector view.
- `vsip_vcopy_i_d`: Copies from an integer vector view to a double vector view.
- `vsip_vcopy_f_i`: Copies from a float vector view to an integer vector view.
- `vsip_vcopy_d_i`: Copies from a double vector view to an integer vector view.
- `vsip_cvcopy_f_f`: Copies from a complex float vector view to another complex float vector view.
- `vsip_cvcopy_d_d`: Copies from a complex double vector view to another complex double vector view.
- `vsip_vcopy_vi_vi`: Copies from a vector index vector view to another vector index vector view.
- `vsip_vcopy_vi_i`: Copies from a vector index vector view to integer vector view.
- `vsip_vcopy_i_vi`: Copies from a integer vector view to a vector index vector view.
- `vsip_vcopy_mi_mi`: Copies from a matrix index vector view to another matrix index vector view.
- `vsip_vcopy_bl_bl`: Copies from a boolean vector view to another boolean vector view.
- `vsip_vcopy_bl_f`: Copies from a boolean vector view to a real vector view.
- `vsip_vcopy_f_bl`: Copies from a real vector view to a boolean vector view.

Example

```
vsip_vview_f *src_float_view;  
vsip_vview_f *dst_float_view;  
  
// Assuming all views have been properly initialized  
  
// Copy from float vector view to float vector view  
vsip_vcopy_f_f(src_float_view, dst_float_view);
```

4.9.2 vsip_dmcopy_p - Copy Matrix Views

```
void vsip_mcopy_f_f(const vsip_mview_f* A, const vsip_mview_f* B);
void vsip_mcopy_d_d(const vsip_mview_d* A, const vsip_mview_d* B);
void vsip_cmcopy_f_f(const vsip_cmview_f* A, const vsip_cmview_f* B);
void vsip_cmcopy_d_d(const vsip_cmview_d* A, const vsip_cmview_d* B);
```

Description

These functions copy the contents of one matrix view to another, with optional type conversion. The functions handle both real and complex matrices of various precision levels:

- vsip_mcopy_f_f: Copy from float matrix to float matrix
- vsip_cmcopy_f_f: Copy from complex float matrix to complex float matrix

Parameters

- const vsip_dmview_p* A: Pointer to the source matrix view
- const vsip_dmview_p* B: Pointer to the destination matrix view

Example

```
vsip_mview_f *src_matrix_f;
vsip_mview_f *dst_matrix_f;

// Copy float matrix to float matrix
vsip_mcopy_f_f(src_matrix_f, dst_matrix_f);
```

Notes

- The source and destination matrices must have the same dimensions.
- For type conversion functions, appropriate rounding or truncation is applied when converting to integer types.
- When converting from higher precision to lower precision (e.g., double to float), values may be truncated or rounded.
- The matrices can be views of larger matrices or blocks, allowing for copying of submatrices.

4.10 Manipulation Operations

4.10.1 vsip_vreal_p - Extract Real Part of a Complex Vector View

```
void vsip_vreal_f(const vsip_cvview_f* a, const vsip_vview_f* r);  
void vsip_vreal_d(const vsip_cvview_d* a, const vsip_vview_d* r);
```

Description

This function extracts the real part of the complex vector view `a` and stores the result in the real vector view `r`.

Parameters

- `const vsip_cvview_p* a`: Pointer to the source complex vector view.
- `const vsip_vview_p* r`: Pointer to the destination real vector view.

Example

```
vsip_cvview_f *complex_vector;  
vsip_vview_f *real_vector;  
  
// Assuming complex_vector and real_vector have been properly initialized  
vsip_vreal_f(complex_vector, real_vector);
```

4.10.2 vsip_vimag_p - Extract Imaginary Part of a Complex Vector View

```
void vsip_vimag_f(const vsip_cvview_f* a, const vsip_vview_f* r);  
void vsip_vimag_d(const vsip_cvview_d* a, const vsip_vview_d* r);
```

Description

This function extracts the imaginary part of the complex vector view `a` and stores the result in the real vector view `r`.

Parameters

- `const vsip_cvview_p* a`: Pointer to the source complex vector view.
- `const vsip_vview_p* r`: Pointer to the destination real vector view.

Example

```
vsip_cvview_f *complex_vector;  
vsip_vview_f *imag_vector;  
  
// Assuming complex_vector and imag_vector have been properly initialized  
vsip_vimag_f(complex_vector, imag_vector);
```

4.10.3 vsip_vcmlx_p - Create a Complex Vector View from Real and Imaginary Parts

```
void vsip_vcmlx_f(const vsip_vview_f* a, const vsip_vview_f* b, const vsip_cvview_f* r);  
void vsip_vcmlx_d(const vsip_vview_d* a, const vsip_vview_d* b, const vsip_cvview_d* r);
```

Description

This function creates a complex vector view `r` from the real vector view `a` and the imaginary vector view `b`.

Parameters

- `const vsip_vview_p* a`: Pointer to the source real vector view.
- `const vsip_vview_p* b`: Pointer to the source imaginary vector view.
- `const vsip_cvview_p* r`: Pointer to the destination complex vector view.

Example

```
vsip_vview_f *real_vector;  
vsip_vview_f *imag_vector;  
vsip_cvview_f *complex_vector;  
  
// Assuming real_vector, imag_vector, and complex_vector have been properly initialized  
vsip_vcmlx_f(real_vector, imag_vector, complex_vector);
```


4.10.4 vsip_dvgather_p - Gather Elements from a Vector

```
void vsip_vgather_i(const vsip_vview_i *a, const vsip_vview_vi *b, const vsip_vview_i *r);
void vsip_vgather_f(const vsip_vview_f *a, const vsip_vview_vi *b, const vsip_vview_f *r);
void vsip_vgather_d(const vsip_vview_d *a, const vsip_vview_vi *b, const vsip_vview_d *r);
void vsip_cvgather_f(const vsip_cvview_f *a, const vsip_vview_vi *b, const vsip_cvview_f *r);
void vsip_cvgather_d(const vsip_cvview_d *a, const vsip_vview_vi *b, const vsip_cvview_d *r);
```

Description

This function gathers elements from an input integer vector a according to the indices specified in vector b , and stores the results in output vector r . The operation performs:

$$r_i = a_{b_i}$$

for all i from 0 to $n - 1$, where n is the length of the index and output vectors.

Parameters

- `const vsip_dvview_p* a`: Input integer vector from which elements are gathered.
- `const vsip_vview_vi* b`: Index vector containing the positions of elements to gather from a .
- `const vsip_dvview_p* r`: Output integer vector that will store the gathered elements.

Example

```
vsip_vview_i *data, *result;
vsip_vview_vi *indices;
vsip_length n = 10; // Number of elements to gather
vsip_length data_size = 100; // Size of input data vector

// Create vectors
data = vsip_vcreate_i(data_size, VSIP_MEM_NONE);
result = vsip_vcreate_i(n, VSIP_MEM_NONE);
indices = vsip_vcreate_vi(n, VSIP_MEM_NONE);

// Initialize data vector with some values
for (vsip_length i = 0; i < data_size; i++) {
    vsip_vput_i(data, i, i * 10); // Example data
}

// Set up indices to gather (e.g., every 10th element)
for (vsip_length i = 0; i < n; i++) {
    vsip_vput_vi(indices, i, i * 10);
}

// Gather elements from data vector
vsip_vgather_i(data, indices, result);

// The result vector now contains elements from data at positions:
// 0, 10, 20, 30, 40, 50, 60, 70, 80, 90

// Print results
printf("Gathered elements:\n");
for (vsip_length i = 0; i < n; i++) {
    printf("%d ", vsip_vget_i(result, i));
}
printf("\n");

// Clean up
vsip_valldestroy_i(data);
```

```
vsip_valldestroy_i(result);  
vsip_valldestroy_vi(indices);
```

Notes

- The index vector b must contain valid indices for the input vector a (i.e., $0 \leq b_i < \text{length}(a)$).
- The output vector r must have the same length as the index vector b .

4.10.5 vsip_dvscatter_p - Scatter Elements to a Vector

```
void vsip_vscatter_i(const vsip_vview_i *a, const vsip_vview_i *r, const vsip_vview_vi *b);
void vsip_vscatter_f(const vsip_vview_f *a, const vsip_vview_f *r, const vsip_vview_vi *b);
void vsip_vscatter_d(const vsip_vview_d *a, const vsip_vview_d *r, const vsip_vview_vi *b);
void vsip_cvscatter_f(const vsip_cvview_f *a, const vsip_cvview_f *r, const vsip_vview_vi *b);
void vsip_cvscatter_d(const vsip_cvview_d *a, const vsip_cvview_d *r, const vsip_vview_vi *b);
```

Description

This function scatters elements from an input integer vector a into specific positions of an output vector r , with the positions specified by the index vector b . The operation performs:

$$r_{b_i} = a_i$$

for all i from 0 to $n - 1$, where n is the length of the input and index vectors.

Parameters

- `const vsip_dvview_p* a`: Input integer vector containing elements to scatter.
- `const vsip_dvview_p* r`: Output integer vector that will receive the scattered elements.
- `const vsip_vview_vi* b`: Index vector containing the positions in r where elements from a should be placed.

Example

```
vsip_vview_i *data, *result;
vsip_vview_vi *indices;
vsip_length n = 10; // Number of elements to scatter
vsip_length result_size = 100; // Size of output vector

// Create vectors
data = vsip_vcreate_i(n, VSIP_MEM_NONE);
result = vsip_vcreate_i(result_size, VSIP_MEM_NONE);
indices = vsip_vcreate_vi(n, VSIP_MEM_NONE);

// Initialize data vector with values to scatter
for (vsip_length i = 0; i < n; i++) {
    vsip_vput_i(data, i, i * i); // Example: square numbers
}

// Initialize result vector (e.g., with zeros)
vsip_vfill_i(result, 0);

// Set up indices where to scatter elements (e.g., every 10th position)
for (vsip_length i = 0; i < n; i++) {
    vsip_vput_vi(indices, i, i * 10);
}

// Scatter elements to result vector
vsip_vscatter_i(data, result, indices);

// The result vector now has non-zero values at positions:
// 0, 10, 20, 30, 40, 50, 60, 70, 80, 90
// containing the values from the data vector

// Print some results
printf("Scattered elements at positions:\n");
for (vsip_length i = 0; i < n; i++) {
    vsip_length pos = vsip_vget_vi(indices, i);
    printf("Position %ld: %d\n", pos, vsip_vget_i(result, pos));
}
```

```
}  
  
// Clean up  
vsip_valldestroy_i(data);  
vsip_valldestroy_i(result);  
vsip_valldestroy_vi(indices);
```

Notes

- The index vector b must contain valid indices for the output vector r (i.e., $0 \leq b_i < \text{length}(r)$).
- The input vector a and index vector b must have the same length.

4.10.6 vsip_dvswap_p - Swap Elements Between two Vectors

```
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_vswap_d(const vsip_vview_d *a, const vsip_vview_d *b);
void vsip_cvswap_f(const vsip_cvview_f *a, const vsip_cvview_f *b);
void vsip_cvswap_d(const vsip_cvview_d *a, const vsip_cvview_d *b);
```

Description

This function swaps the elements between two vectors a and b . After the operation, vector a will contain the elements that were originally in vector b , and vice versa. The operation performs an element-wise swap:

$$\text{temp} = a_i$$

$$a_i = b_i$$

$$b_i = \text{temp}$$

for all i from 0 to $n - 1$, where n is the length of the vectors.

Parameters

- `const vsip_dvview_p* a`: First floating-point vector.
- `const vsip_dvview_p* b`: Second floating-point vector.

Example

```
vsip_vview_f *signal1, *signal2;
vsip_length n = 1024; // Vector length

// Create vectors for your signal processing
signal1 = vsip_vcreate_f(n, VSIP_MEM_NONE);
signal2 = vsip_vcreate_f(n, VSIP_MEM_NONE);

// Initialize vectors with some data
// For example, fill with sample data for your algorithms
vsip_vramp_f(0.0f, 1.0f, signal1); // signal1 = [0, 1, 2, ..., 1023]
vsip_vramp_f(10.0f, -0.5f, signal2); // signal2 = [10, 9.5, 9, ..., -502]

// Print some values before swap
printf("Before swap:\n");
printf("signal1[0:4] = %.2f, %.2f, %.2f, %.2f\n",
       vsip_vget_f(signal1, 0), vsip_vget_f(signal1, 1),
       vsip_vget_f(signal1, 2), vsip_vget_f(signal1, 3));
printf("signal2[0:4] = %.2f, %.2f, %.2f, %.2f\n",
       vsip_vget_f(signal2, 0), vsip_vget_f(signal2, 1),
       vsip_vget_f(signal2, 2), vsip_vget_f(signal2, 3));

// Swap the vectors
vsip_vswap_f(signal1, signal2);

// Now signal1 contains the original signal2 data and vice versa
printf("\nAfter swap:\n");
printf("signal1[0:4] = %.2f, %.2f, %.2f, %.2f\n",
       vsip_vget_f(signal1, 0), vsip_vget_f(signal1, 1),
       vsip_vget_f(signal1, 2), vsip_vget_f(signal1, 3));
printf("signal2[0:4] = %.2f, %.2f, %.2f, %.2f\n",
       vsip_vget_f(signal2, 0), vsip_vget_f(signal2, 1),
       vsip_vget_f(signal2, 2), vsip_vget_f(signal2, 3));
```

```
// Clean up  
vsip_valldestroy_f(signal1);  
vsip_valldestroy_f(signal2);
```

Notes

- Both vectors must have the same length.
- The operation is performed in-place on both vectors.
- This operation is more efficient than manually copying elements between vectors using a temporary buffer.
- Be cautious when using this function with vectors that might be views of the same underlying data, as this could lead to unexpected results.

4.10.7 vsip_vrect_p - Convert Cartesian Coordinates to Complex Numbers

```
void vsip_vrect_f(const vsip_vview_f *a, const vsip_vview_f *b, const vsip_cvview_f *r);
void vsip_vrect_d(const vsip_vview_d *a, const vsip_vview_d *b, const vsip_cvview_d *r);
```

Description

This function converts pairs of real vectors representing Cartesian coordinates (real and imaginary parts) into a complex vector.

The operation performs element-wise conversion:

$$r_i = a_i + j \cdot b_i$$

for all i from 0 to $n - 1$, where n is the length of the vectors, a_i is the real part, b_i is the imaginary part, and r_i is the resulting complex number.

Parameters

- `const vsip_vview_p*` `a`: Input vector containing real parts.
- `const vsip_vview_p*` `b`: Input vector containing imaginary parts.
- `const vsip_cvview_p*` `r`: Output complex vector that will store the results.

Example

```
vsip_vview_f *real_parts, *imag_parts;
vsip_cvview_f *complex_numbers;
vsip_length n = 10;

// Create vectors
real_parts = vsip_vcreate_f(n, VSIP_MEM_NONE);
imag_parts = vsip_vcreate_f(n, VSIP_MEM_NONE);
complex_numbers = vsip_cvcreate_f(n, VSIP_MEM_NONE);

// Initialize real and imaginary parts
// For example, create a complex signal
for (vsip_length i = 0; i < n; i++) {
    vsip_vput_f(real_parts, i, cos(2 * M_PI * i / n)); // Real parts
    vsip_vput_f(imag_parts, i, sin(2 * M_PI * i / n)); // Imaginary parts
}

// Convert to complex numbers
vsip_vrect_f(real_parts, imag_parts, complex_numbers);

// The complex_numbers vector now contains the complex representation
// of your signal, which can be used in further complex operations

// Print some results
printf("Complex numbers (first 3 elements):\n");
for (vsip_length i = 0; i < 3; i++) {
    vsip_cscalar_f val = vsip_cvget_f(complex_numbers, i);
    printf("%.4f, %.4f ", val.r, val.i);
}
printf("\n");

// Clean up
vsip_valldestroy_f(real_parts);
vsip_valldestroy_f(imag_parts);
vsip_cvalldestroy_f(complex_numbers);
```

Notes

- All three vectors must have the same length.
- This operation is the inverse of `vsip_vpolar_p` which converts from polar to Cartesian coordinates.

4.10.8 vsip_vpolar_p - Convert Polar Coordinates to Cartesian

```
void vsip_vpolar_f(const vsip_cvview_f *a, const vsip_vview_f *r, const vsip_vview_f *s);
void vsip_vpolar_d(const vsip_cvview_d *a, const vsip_vview_d *r, const vsip_vview_d *s);
```

Description

This function converts complex numbers from a complex vector into their polar coordinate representation (magnitude and phase). The operation performs element-wise conversion from Cartesian to polar coordinates:

$$r_i = |a_i|$$

$$s_i = \arg(a_i)$$

for all i from 0 to $n - 1$, where n is the length of the vectors, r_i is the magnitude, and s_i is the phase (angle in radians) of the complex number a_i .

Parameters

- `const vsip_cvview_p* a`: Input complex vector.
- `const vsip_vview_p* r`: Output vector that will store the magnitudes.
- `const vsip_vview_p* s`: Output vector that will store the phases (in radians).

Example

```
vsip_cvview_f *complex_signal;
vsip_vview_f *magnitudes, *phases;
vsip_length n = 10;

// Create vectors
complex_signal = vsip_cvcreate_f(n, VSIP_MEM_NONE);
magnitudes = vsip_vcreate_f(n, VSIP_MEM_NONE);
phases = vsip_vcreate_f(n, VSIP_MEM_NONE);

// Initialize complex signal (e.g., with some complex values)
for (vsip_length i = 0; i < n; i++) {
    float real = cos(2 * M_PI * i / n);
    float imag = sin(2 * M_PI * i / n);
    vsip_cvput_f(complex_signal, i, VSIP_CMPLX_F(real, imag));
}

// Convert to polar coordinates
vsip_vpolar_f(complex_signal, magnitudes, phases);

// The magnitudes and phases vectors now contain the polar representation
// of your complex signal

// Print some results
printf("Magnitude and Phase (first 3 elements):\n");
for (vsip_length i = 0; i < 3; i++) {
    printf("Element %ld: Magnitude = %.4f, Phase = %.4f radians\n",
        i, vsip_vget_f(magnitudes, i), vsip_vget_f(phases, i));
}

// Clean up
vsip_cvalldestroy_f(complex_signal);
vsip_valldestroy_f(magnitudes);
vsip_valldestroy_f(phases);
```

Notes

- All three vectors must have the same length.
- The phase values are returned in radians in the range $[-\pi, \pi]$.
- This operation is the inverse of `vsip_vrect_p` which converts from Cartesian to polar coordinates.

Chapter 5

Signal Processing Functions

5.1 FFT Functions

5.1.1 vsip_ddfftop_create_p - Create FFT Objects (Out-of-Place)

```

typedef enum _vsip_fft_dir {
    VSIP_FFT_FWD = -1,
    VSIP_FFT_INV = +1
} vsip_fft_dir;

typedef enum _vsip_alg_hint {
    VSIP_ALG_TIME = 0,
    VSIP_ALG_SPACE = 1,
    VSIP_ALG_NOISE = 2
} vsip_alg_hint;

vsip_fft_f* vsip_ccfftop_create_f(vsip_length length, vsip_scalar_f scale,
                                vsip_fft_dir sign, vsip_length ntimes,
                                vsip_alg_hint hint);
vsip_fft_f* vsip_rcfftop_create_f(vsip_length length, vsip_scalar_f scale,
                                vsip_fft_dir sign, vsip_length ntimes,
                                vsip_alg_hint hint);
vsip_fft_f* vsip_crfftop_create_f(vsip_length length, vsip_scalar_f scale,
                                vsip_fft_dir sign, vsip_length ntimes,
                                vsip_alg_hint hint);

vsip_fft_d* vsip_ccfftop_create_d(vsip_length length, vsip_scalar_d scale,
                                vsip_fft_dir sign, vsip_length ntimes,
                                vsip_alg_hint hint);
vsip_fft_d* vsip_rcfftop_create_d(vsip_length length, vsip_scalar_d scale,
                                vsip_fft_dir sign, vsip_length ntimes,
                                vsip_alg_hint hint);
vsip_fft_d* vsip_crfftop_create_d(vsip_length length, vsip_scalar_d scale,
                                vsip_fft_dir sign, vsip_length ntimes,
                                vsip_alg_hint hint);

```

Description

These functions create FFT (Fast Fourier Transform) objects for different types of FFT operations:

- `vsip_ccfftop_create_p`: Creates an FFT object for complex-to-complex out-of-place FFT.
- `vsip_rcfftop_create_p`: Creates an FFT object for real-to-complex out-of-place FFT.
- `vsip_crfftop_create_p`: Creates an FFT object for complex-to-real out-of-place FFT.

Each function initializes the FFT object with the specified length, scale factor, direction, number of times to apply the FFT, and algorithm hint.

The performance for supported FFT sizes is standardized as $O(n \log n)$. For sizes not directly supported by the FFT kernels a DFT fallback with a performance of $O(n^2)$ is standardized.

Parameters

- `vsip_length length`: The length of the FFT.
- `vsip_scalar_f scale`: The scale factor to apply to the FFT result.
- `vsip_fft_dir sign`: The direction of the FFT.
 - `VSIP_FFT_FWD` - Forward
 - `VSIP_FFT_INV` - Inverse
- `vsip_length ntimes`: The number of times to apply the FFT.
- `vsip_alg_hint hint`: Algorithm hint for the FFT.

- VSIP_ALG_TIME - Optimize for time
- VSIP_ALG_SPACE - Optimize for memory usage
- VSIP_ALG_NOISE - Optimize for noise

Return Value

- On success, a pointer to the newly created FFT object is returned.
- On error, NULL is returned.

Example

```
vsip_length length = 1024;
vsip_scalar_f scale = 1.0;
vsip_fft_dir direction = VSIP_FFT_FWD; // Forward FFT
vsip_length ntimes = 1;
vsip_alg_hint hint = VSIP_ALG_TIME;

vsip_fft_f *fft_cc;
vsip_fft_f *fft_rc;
vsip_fft_f *fft_cr;

// Create complex-to-complex FFT object
fft_cc = vsip_ccfftop_create_f(length, scale, direction, ntimes, hint);
if (fft_cc == NULL) {
    // Handle error
}
```

5.1.2 vsip_ccfftip_create_p - Create FFT Object (In-Place)

```
typedef enum _vsip_fft_dir {
    VSIP_FFT_FWD = -1,
    VSIP_FFT_INV = +1
} vsip_fft_dir;

typedef enum _vsip_alg_hint {
    VSIP_ALG_TIME = 0,
    VSIP_ALG_SPACE = 1,
    VSIP_ALG_NOISE = 2
} vsip_alg_hint;

vsip_fft_f* vsip_ccfftip_create_f(vsip_length length, vsip_scalar_f scale,
                                vsip_fft_dir sign, vsip_length ntimes,
                                vsip_alg_hint hint);
vsip_fft_d* vsip_ccfftip_create_d(vsip_length length, vsip_scalar_d scale,
                                vsip_fft_dir sign, vsip_length ntimes,
                                vsip_alg_hint hint);
```

Description

These functions create FFT (Fast Fourier Transform) object for a complex-to-complex in-place FFT. The functions initialize a FFT object with the specified length, scale factor, direction, number of times to apply the FFT, and algorithm hint.

The performance for supported FFT sizes is standardized as $O(n \log n)$. For sizes not directly supported by the FFT kernels a DFT fallback with a performance of $O(n^2)$ is standardized.

Parameters

- `vsip_length length`: The length of the FFT.
- `vsip_scalar_f scale`: The scale factor to apply to the FFT result.
- `vsip_fft_dir sign`: The direction of the FFT.
 - `VSIP_FFT_FWD` - Forward
 - `VSIP_FFT_INV` - Inverse
- `vsip_length ntimes`: The number of times to apply the FFT.
- `vsip_alg_hint hint`: Algorithm hint for the FFT.
 - `VSIP_ALG_TIME` - Optimize for time
 - `VSIP_ALG_SPACE` - Optimize for memory usage
 - `VSIP_ALG_NOISE` - Optimize for noise

Return Value

- On success, a pointer to the newly created FFT object is returned.
- On error, `NULL` is returned.

Example

```
vsip_length length = 1024;
vsip_scalar_f scale = 1.0;
vsip_fft_dir direction = VSIP_FFT_FWD; // Forward FFT
vsip_length ntimes = 1;
vsip_alg_hint hint = VSIP_ALG_TIME;

vsip_fft_f *fft_cc;
```

```
vsip_fft_f *fft_rc;  
vsip_fft_f *fft_cr;  
  
// Create complex-to-complex FFT object  
fft_cc = vsip_ccfftip_create_f(length, scale, direction, ntimes, hint);  
if (fft_cc == NULL) {  
    // Handle error  
}
```


5.1.3 vsip_fft_destroy_p - Destroy an FFT Object

```
int vsip_fft_destroy_f(vsip_fft_f *fft);  
int vsip_fft_destroy_d(vsip_fft_d *fft);
```

Description

This function destroys the specified FFT object and frees associated resources.

Parameters

- `vsip_fft_p* fft`: Pointer to the FFT object to be destroyed.

Return Value

- Returns 0 on success.
- Returns a non-zero value on error.

Example

```
vsip_fft_f *fft;  
int result;  
  
// Assuming fft has been properly initialized  
result = vsip_fft_destroy_f(fft);  
  
if (result != 0) {  
    // Handle error  
}
```

5.1.4 vsip_fft_getattr_p - Get FFT Object Attributes

```
typedef struct _vsip_fft_attr_f {  
    vsip_scalar_vi input;  
    vsip_scalar_vi output;  
    vsip_fft_place place;  
    vsip_scalar_f scale;  
    vsip_fft_dir dir;  
} vsip_fft_attr_f;  
/* Similar for the other data types */  
  
void vsip_fft_getattr_f(const vsip_fft_f *fft, vsip_fft_attr_f *attr);  
void vsip_dft_getattr_d(const vsip_dft_d *fft, vsip_dft_attr_d *attr);
```

Description

This function retrieves the attributes of an FFT (Fast Fourier Transform) object and stores them in the provided attribute structure.

Parameters

- `const vsip_fft_p* fft`: Pointer to the FFT object.
- `vsip_fft_attr_p* attr`: Pointer to the attribute structure where the FFT object attributes will be stored.

5.1.5 vsip_ddfftop_p - Perform FFT Operations (Out-of-Place)

```

void vsip_ccfftop_f(const vsip_fft_f *fft, const vsip_cvview_f *x, const vsip_cvview_f *y);
void vsip_rcfftop_f(const vsip_fft_f *fft, const vsip_vview_f *x, const vsip_cvview_f *y);
void vsip_crfftop_f(const vsip_fft_f *fft, const vsip_cvview_f *x, const vsip_vview_f *y);

void vsip_ccfftop_d(const vsip_fft_d *fft, const vsip_cvview_d *x, const vsip_cvview_d *y);
void vsip_rcfftop_d(const vsip_fft_d *fft, const vsip_vview_d *x, const vsip_cvview_d *y);
void vsip_crfftop_d(const vsip_fft_d *fft, const vsip_cvview_d *x, const vsip_vview_d *y);

```

Description

These functions perform FFT (Fast Fourier Transform) operations using the specified FFT object. Each function handles a different type of FFT:

- `vsip_ccfftop_p`: Performs a out-of-place complex-to-complex FFT.
- `vsip_rcfftop_p`: Performs a out-of-place real-to-complex FFT.
- `vsip_crfftop_p`: Performs a out-of-place complex-to-real FFT.

The performance for supported FFT sizes is standardized as $O(n \log n)$. For sizes not directly supported by the FFT kernels a DFT fallback with a performance of $O(n^2)$ is standardized.

Parameters

- `const vsip_fft_p* fft`: Pointer to the FFT object.
- `const vsip_dvview_p* x`: Pointer to the input complex vector view
- `const vsip_dvview_p* y`: Pointer to the output complex vector view

Example

```

vsip_fft_f *fft_cc;
vsip_fft_f *fft_rc;
vsip_fft_f *fft_cr;
vsip_cvview_f *complex_input;
vsip_cvview_f *complex_output;
vsip_vview_f *real_input;
vsip_vview_f *real_output;

```

```
// Assuming fft_cc, fft_rc, fft_cr, complex_input, complex_output, real_input, and real_output have been p
```

```
// Perform complex-to-complex FFT
```

```
vsip_ccfftop_f(fft_cc, complex_input, complex_output);
```

5.1.6 vsip_ccfftip_p - Perform FFT Operations (In-Place)

```
void vsip_ccfftip_f(const vsip_fft_f *fft, const vsip_cvview_f *y);
```

```
void vsip_ccfftip_d(const vsip_fft_d *fft, const vsip_cvview_d *y);
```

Description

These functions perform FFT (Fast Fourier Transform) operations using the specified FFT object in-place.

The performance for supported FFT sizes is standardized as $O(n \log n)$. For sizes not directly supported by the FFT kernels a DFT fallback with a performance of $O(n^2)$ is standardized.

Parameters

- `const vsip_fft_p* fft`: Pointer to the FFT object.
- `const vsip_cvview_p* y`: Pointer to the complex input and output vector view.

5.1.7 vsip_ddffmop_create_p - Create Multiple-FFT Objects (Out-of-Place)

```

typedef enum _vsip_fft_dir {
    VSIP_FFT_FWD = -1,
    VSIP_FFT_INV = +1
} vsip_fft_dir;

typedef enum _vsip_alg_hint {
    VSIP_ALG_TIME = 0,
    VSIP_ALG_SPACE = 1,
    VSIP_ALG_NOISE = 2
} vsip_alg_hint;

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_crfftmop_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_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_d* vsip_ccfftmop_create_d(vsip_length m, vsip_length n,
                                   vsip_scalar_d scale, vsip_dft_dir dir,
                                   vsip_major major, vsip_length ntimes,
                                   vsip_alg_hint hint);
vsip_fftm_d* vsip_crfftmop_create_d(vsip_length m, vsip_length n,
                                   vsip_scalar_d scale, vsip_major major,
                                   vsip_length ntimes, vsip_alg_hint hint);
vsip_fftm_d* vsip_rcfftmop_create_d(vsip_length m, vsip_length n,
                                   vsip_scalar_d scale, vsip_major major,
                                   vsip_length ntimes, vsip_alg_hint hint);

```

Description

These functions create Multiple-FFT (Fast Fourier Transform) objects for different types of FFT operations:

- `vsip_ccffmop_create_p`: Creates an Multiple-FFT object for complex-to-complex out-of-place FFT.
- `vsip_rcffmop_create_p`: Creates an Multiple-FFT object for real-to-complex out-of-place FFT.
- `vsip_crffmop_create_p`: Creates an Multiple-FFT object for complex-to-real out-of-place FFT.

Each function initializes the FFT object with the specified length, scale factor, direction, number of times to apply the FFT, and algorithm hint.

The performance for supported FFT sizes is standardized as $O(n \log n)$. For sizes not directly supported by the FFT kernels a DFT fallback with a performance of $O(n^2)$ is standardized.

Parameters

- `vsip_length m`: The length of columns or rows, depending on the given major.
- `vsip_length n`: The length of rows or columns, depending on the given major.
- `vsip_scalar_f scale`: The scale factor to apply to the FFT result.
- `vsip_fft_dir sign`: The direction of the FFT.
 - `VSIP_FFT_FWD` - Forward
 - `VSIP_FFT_INV` - Inverse

- `vsip_major major`: Direction of the multiple-FFT:
 - `VSIP_ROW` - Row Major
 - `VSIP_Col` - Column Major
- `vsip_length ntimes`: The number of times to apply the FFT.
- `vsip_alg_hint hint`: Algorithm hint for the FFT.
 - `VSIP_ALG_TIME` - Optimize for time
 - `VSIP_ALG_SPACE` - Optimize for memory usage
 - `VSIP_ALG_NOISE` - Optimize for noise

Return Value

- On success, a pointer to the newly created FFT object is returned.
- On error, `NULL` is returned.

5.1.8 vsip_ccfftmip_create_p - Create Multipe-FFT Object (In-Place)

```

typedef enum _vsip_fft_dir {
    VSIP_FFT_FWD = -1,
    VSIP_FFT_INV = +1
} vsip_fft_dir;

typedef enum _vsip_alg_hint {
    VSIP_ALG_TIME = 0,
    VSIP_ALG_SPACE = 1,
    VSIP_ALG_NOISE = 2
} vsip_alg_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_d* vsip_ccfftmip_create_d(vsip_length m, vsip_length n,
                                   vsip_scalar_d scale, vsip_fft_dir dir,
                                   vsip_major major, vsip_length ntimes,
                                   vsip_alg_hint hint);

```

Description

These functions create a Multiple-FFT (Fast Fourier Transform) object for a complex-to-complex in-place FFT. The functions initialize a FFT object with the specified length, scale factor, direction, number of times to apply the FFT, and algorithm hint.

Parameters

- `vsip_length m`: The length of columns or rows, depending on the given major.
- `vsip_length n`: The length of rows or columns, depending on the given major.
- `vsip_scalar_f scale`: The scale factor to apply to the FFT result.
- `vsip_fft_dir sign`: The direction of the FFT.
 - `VSIP_FFT_FWD` - Forward
 - `VSIP_FFT_INV` - Inverse
- `vsip_major major`: Direction of the multiple-FFT:
 - `VSIP_ROW` - Row Major
 - `VSIP_Col` - Column Major
- `vsip_length ntimes`: The number of times to apply the FFT.
- `vsip_alg_hint hint`: Algorithm hint for the FFT.
 - `VSIP_ALG_TIME` - Optimize for time
 - `VSIP_ALG_SPACE` - Optimize for memory usage
 - `VSIP_ALG_NOISE` - Optimize for noise

Return Value

- On success, a pointer to the newly created FFT object is returned.
- On error, NULL is returned.

5.1.9 vsip_fftm_destroy_p - Destroy a Multiple-FFT Object

```
int vsip_fftm_destroy_f(vsip_fftm_f *fft);  
int vsip_fftm_destroy_d(vsip_fftm_d *fft);
```

Description

This function destroys the specified Multiple-FFT object and frees associated resources.

Parameters

- `vsip_fftm_p * fft`: Pointer to the Multiple-FFT object to be destroyed.

Return Value

- Returns 0 on success.
- Returns a non-zero value on error.

5.1.10 vsip_fftm_getattr_p - Get Multiple-FFT Object Attributes

```
typedef struct _vsip_fftm_attr_f {
    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;
/* Same for the other vsip_fftm_attr_d types */

void vsip_fftm_getattr_f(const vsip_fftm_f *fft, vsip_fftm_attr_f *attr);
void vsip_dftm_getattr_d(const vsip_dftm_d *fft, vsip_dftm_attr_d *attr);
```

Description

This function retrieves the attributes of an Multiple-FFT (Fast Fourier Transform) object and stores them in the provided attribute structure.

Parameters

- `const vsip_fftm_p* fft`: Pointer to the FFT object.
- `vsip_fftm_attr_p* attr`: Pointer to the attribute structure where the FFT object attributes will be stored.

5.1.11 vsip_ddffmop_p - Perform Multiple-FFT Operations (Out-of-Place)

```

void vsip_ccfftmop_f(const vsip_fftm_f *fft, const vsip_cmview_f *x, const vsip_cmview_f *y);
void vsip_crfftmop_f(const vsip_fftm_f *fft, const vsip_cmview_f *x, const vsip_mview_f *y);
void vsip_rcfftmop_f(const vsip_fftm_f *fft, const vsip_mview_f *x, const vsip_cmview_f *y);

void vsip_ccfftmop_d(const vsip_fftm_d *fft, const vsip_cmview_d *x, const vsip_cmview_d *y);
void vsip_crfftmop_d(const vsip_fftm_d *fft, const vsip_cmview_d *x, const vsip_mview_d *y);
void vsip_rcfftmop_d(const vsip_fftm_d *fft, const vsip_mview_d *x, const vsip_cmview_d *y);

```

Description

These functions perform Multiple-FFT (Fast Fourier Transform) operations using the specified FFT object. Each function handles a different type of FFT:

- `vsip_ccfftmop_p`: Performs a out-of-place complex-to-complex Multiple-FFT.
- `vsip_rcfftmop_p`: Performs a out-of-place real-to-complex Multiple-FFT.
- `vsip_crfftmop_p`: Performs a out-of-place complex-to-real Multiple-FFT.

Parameters

- `const vsip_fftm_p * fft`: Pointer to the FFT object.
- `const vsip_dmview_p * x`: Pointer to the input complex matrix view
- `const vsip_dmview_p * y`: Pointer to the output complex matrix view

5.1.12 vsip_ccffmip_p - Perform Multiple-FFT Operations (In-Place)

```
void vsip_ccffmip_f(const vsip_fftm_f *fft, const vsip_cmview_f *y);
```

```
void vsip_ccffmip_d(const vsip_fftm_d *fft, const vsip_cmview_d *y);
```

Description

These functions perform Multiple-FFT (Fast Fourier Transform) operations using the specified FFT object in-place.

Parameters

- `const vsip_fftm_p* fft`: Pointer to the FFT object.
- `const vsip_cmview_p* y`: Pointer to the complex input and output matrix view.

5.2 Convolution and Correlation Functions

5.2.1 vsip_dconv1d_create_p - Create 1D Convolution Object

```
typedef enum _vsip_alg_hint {
    VSIP_ALG_TIME = 0,
    VSIP_ALG_SPACE = 1,
    VSIP_ALG_NOISE = 2
} vsip_alg_hint;

typedef enum _vsip_support_region {
    VSIP_SUPPORT_FULL = 0,
    VSIP_SUPPORT_SAME = 1,
    VSIP_SUPPORT_MIN = 2,
} vsip_support_region;

typedef enum _vsip_symmetry {
    VSIP_NONSYM = 0,
    VSIP_SYM_EVEN_LEN_ODD = 1,
    VSIP_SYM_ODD_LEN_EVEN = 2
} vsip_symmetry;

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_conv1d_d* vsip_conv1d_create_d(const vsip_vview_d *h, vsip_symmetry symm,
                                   vsip_length n, vsip_length d, vsip_support_region support,
                                   vsip_length ntimes, vsip_alg_hint hint);
```

Description

This function creates a one-dimensional convolution object. The convolution object can handle various types of impulse responses and supports different output regions and decimation factors.

Parameters

- `const vsip_d vview_p* h`: Vector containing the impulse response (filter coefficients).
- `vsip_symmetry symm`: Symmetry of the impulse response:
 - `VSIP_SYM_EVEN_LEN_ODD`: Even symmetry, odd length
 - `VSIP_SYM_ODD_LEN_EVEN`: Odd symmetry, even length
 - `VSIP_NOSYM`: No symmetry
- `vsip_length n`: Length of the input signal.
- `vsip_length d`: Decimation factor (1 for no decimation).
- `vsip_support_region support`: Support region for the convolution:
 - `VSIP_SUPPORT_FULL`: Full convolution. Output length is $\lfloor (n + m - 2)/d \rfloor + 1$
 - `VSIP_SUPPORT_SAME`: Same-length output. Output length is $\lfloor (n - 1)/d \rfloor + 1$
 - `VSIP_SUPPORT_MIN`: Minimum-length output. Output length is $\lfloor (n - 1)/d \rfloor - \lfloor (m - 1)/d \rfloor + 1$
- `vsip_length ntimes`: Number of times the convolution will be applied.
- `vsip_alg_hint hint`: Algorithm hint for optimization:
 - `VSIP_ALG_TIME`: Optimize for computation time
 - `VSIP_ALG_SPACE`: Optimize for memory usage
 - `VSIP_ALG_NOHINT`: No specific optimization

Return Value

- On success: Pointer to the newly created 1D convolution object.
- On error (e.g., memory allocation failure): NULL.

Example

```

vsip_conv1d_f *conv;
vsip_vview_f *h;
vsip_length h_len = 31; // Impulse response length
vsip_length n = 1024;   // Input signal length
vsip_length d = 1;      // No decimation

// Create impulse response vector
h = vsip_vcreate_f(h_len, VSIP_MEM_NONE);

// Initialize impulse response (e.g., Gaussian kernel)
// vsip_vramp_f(0.0f, 1.0f, h);
// Apply window function or other modifications to h...

// Create convolution object for full convolution
conv = vsip_conv1d_create_f(h, VSIP_SYM_NONE, n, d,
                           VSIP_SUPPORT_FULL, 100, VSIP_ALG_TIME);
if (conv == NULL) {
    fprintf(stderr, "Error: Could not create convolution object\n");
    return;
}

// Use the convolution object for your signal processing
// vsip_vview_f *input = vsip_vcreate_f(n, VSIP_MEM_NONE);
// vsip_vview_f *output = vsip_vcreate_f(n + h_len - 1, VSIP_MEM_NONE);
// vsip_conv1d_f(conv, input, output);

// Clean up when done
vsip_conv1d_destroy_f(conv);
vsip_vdestroy_f(h);

```

Notes

- The convolution object should be destroyed with `vsip_d conv1d_destroy_p` when no longer needed.
- The decimation factor d allows for downsampling the output.

5.2.2 vsip_dconv1d_destroy_p - Destroy 1D Convolution Object

```
vsip_length vsip_conv1d_destroy_f(vsip_conv1d_f *conv1d);
vsip_length vsip_conv1d_destroy_d(vsip_conv1d_d *conv1d);
```

Description

This function releases all memory and resources associated with a 1D convolution object that was previously created with `vsip_dconv1d_create_p`.

Parameters

- `vsip_dconv1d_p* conv1d`: Pointer to the 1D convolution object to be destroyed.

Return Value

- Returns 0.

Example

```
vsip_conv1d_f *conv;
vsip_vview_f *h;
vsip_length h_len = 31; // Impulse response length
vsip_length n = 1024;   // Input signal length

// Create impulse response vector
h = vsip_vcreate_f(h_len, VSIP_MEM_NONE);

// Create convolution object
conv = vsip_conv1d_create_f(h, VSIP_NOSYM, n, 1,
                           VSIP_SUPPORT_FULL, 100, VSIP_ALG_TIME);

if (conv == NULL) {
    fprintf(stderr, "Error: Could not create convolution object\n");
    return;
}

// Use the convolution object for your signal processing
// ... your convolution operations ...

// Destroy convolution object when done
vsip_conv1d_destroy_f(conv);
vsip_valldestroy_f(h);
```

5.2.3 vsip_dconv1d_getattr_p - Get 1D Convolution Object Attributes

```
typedef struct _vsip_conv1d_attr_f {
    vsip_scalar_vi    kernel_len; // Kernel length
    vsip_symmetry      symm;      // Symmetry
    vsip_scalar_vi    data_len;   // Data length
    vsip_support_region support;   // Support
    vsip_scalar_vi    out_len;    // Output length
    vsip_length        decimation; // Decimation
} vsip_conv1d_attr_f;

void vsip_conv1d_getattr_f(const vsip_conv1d_f *conv1d, vsip_conv1d_attr_f *attr);
void vsip_conv1d_getattr_d(const vsip_conv1d_d *conv1d, vsip_conv1d_attr_d *attr);
```

Description

This function retrieves the attributes of a 1D convolution object and stores them in the provided attribute structure.

Parameters

- `const vsip_dconv1d_p* conv1d`: Pointer to the 1D convolution object created with `vsip_dconv1d_create_p`.
- `vsip_dconv1d_attr_p* attr`: Pointer to the attribute structure where the convolution object attributes will be stored.

Example

```
vsip_conv1d_f *conv;
vsip_conv1d_attr_f attr;
vsip_vview_f *h;
vsip_length h_len = 31; // Impulse response length
vsip_length n = 1024;   // Input signal length

// Create impulse response vector
h = vsip_vcreate_f(h_len, VSIP_MEM_NONE);

// Create convolution object
conv = vsip_conv1d_create_f(h, VSIP_SYM_NONE, n, 1,
                           VSIP_SUPPORT_FULL, 100, VSIP_ALG_TIME);
if (conv == NULL) {
    fprintf(stderr, "Error: Could not create convolution object\n");
    return;
}

// Get the attributes of the convolution object
vsip_conv1d_getattr_f(conv, &attr);

// Clean up
vsip_conv1d_destroy_f(conv);
vsip_valldestroy_f(h);
```


5.2.4 vsip_dconvolve1d_p - Perform 1D Convolution

```
void vsip_convolve1d_f(const vsip_conv1d_f *conv, const vsip_vview_f *x, const vsip_vview_f *y);
void vsip_convolve1d_d(const vsip_conv1d_d *conv, const vsip_vview_d *x, const vsip_vview_d *y);
```

Description

This function performs one-dimensional convolution between an input signal x and the impulse response (filter kernel) stored in the convolution object, storing the result in the output vector y . The convolution operation computes:

$$y_n = \sum_k h_n \cdot x_{n-k}$$

where h is the impulse response stored in the convolution object, and x is the input signal.

Parameters

- `const vsip_dconv1d_p * conv`: Pointer to the 1D convolution object created with `vsip_dconv1d_create_p`.
- `const vsip_dvview_p * x`: Input signal vector of length n (as specified when creating the convolution object).
- `const vsip_dvview_p * y`: Output convolution vector. Its length depends on the support region specified on the creation of the convolution object:
 - `VSIP_SUPPORT_FULL`: Full convolution. Output length is $\lfloor (n + m - 2)/d \rfloor + 1$
 - `VSIP_SUPPORT_SAME`: Same-length output. Output length is $\lfloor (n - 1)/d \rfloor + 1$
 - `VSIP_SUPPORT_MIN`: Minimum-length output. Output length is $\lfloor (n - 1)/d \rfloor - \lfloor (m - 1)/d \rfloor + 1$

Notes

- The input vector x must have length n as specified when creating the convolution object.
- The output vector y must have the appropriate length based on the support region (see Parameters section).
- The convolution object can be reused for multiple convolution operations with different input signals.

5.2.5 vsip_dcorr1d_create_p - Create 1D Correlation Object

```

typedef enum _vsip_support_region {
    VSIP_SUPPORT_FULL = 0,
    VSIP_SUPPORT_SAME = 1,
    VSIP_SUPPORT_MIN = 2,
} vsip_support_region;

typedef enum _vsip_alg_hint {
    VSIP_ALG_TIME = 0,
    VSIP_ALG_SPACE = 1,
    VSIP_ALG_NOISE = 2
} vsip_alg_hint;

vsip_corr1d_f* vsip_corr1d_create_f(vsip_length m, vsip_length n,
                                   vsip_support_region support,
                                   vsip_length ntimes, vsip_alg_hint hint);
vsip_corr1d_d* vsip_corr1d_create_d(vsip_length m, vsip_length n,
                                   vsip_support_region support,
                                   vsip_length ntimes, vsip_alg_hint hint);
vsip_ccorr1d_f* vsip_ccorr1d_create_f(vsip_length m, vsip_length n,
                                       vsip_support_region support,
                                       vsip_length ntimes, vsip_alg_hint hint);
vsip_ccorr1d_d* vsip_ccorr1d_create_d(vsip_length m, vsip_length n,
                                       vsip_support_region support,
                                       vsip_length ntimes, vsip_alg_hint hint);

```

Description

This function creates a one-dimensional correlation object that can be used to compute the correlation between an input signal and a reference signal.

The correlation object is optimized for repeated use, allowing efficient computation of correlations between signals of length m and reference signals of length n .

Parameters

- `vsip_length m`: Length of the input signal.
- `vsip_length n`: Length of the reference signal.
- `vsip_support_region support`: Specifies the support region for the correlation:
 - `VSIP_SUPPORT_FULL`: Compute full correlation. Output length is $n + m - 1$.
 - `VSIP_SUPPORT_SAME`: Compute correlation with same-length output n .
 - `VSIP_SUPPORT_MIN`: Compute minimum-length correlation. Output length is $|n - m| - 1$.
- `vsip_length ntimes`: Number of times the correlation will be applied.
- `vsip_alg_hint hint`: Algorithm hint for optimization:
 - `VSIP_ALG_TIME`: Optimize for computation time
 - `VSIP_ALG_SPACE`: Optimize for memory usage
 - `VSIP_ALG_NOHINT`: No specific optimization

Return Value

- On success: Pointer to the newly created 1D correlation object.
- On error (e.g., memory allocation failure): `NULL`.

Example

```

vsip_corr1d_f *corr;
vsip_length m = 1024; // Input signal length
vsip_length n = 64;   // Reference signal length
vsip_length ntimes = 100; // Number of times to reuse object

// Create correlation object for full correlation
corr = vsip_corr1d_create_f(m, n, VSIP_SUPPORT_FULL, ntimes, VSIP_ALG_TIME);
if (corr == NULL) {
    fprintf(stderr, "Error: Could not create correlation object\n");
    return -1;
}

// Use the correlation object for your signal processing
// vsip_vview_f *input = vsip_vcreate_f(m, VSIP_MEM_NONE);
// vsip_vview_f *reference = vsip_vcreate_f(n, VSIP_MEM_NONE);
// vsip_vview_f *result = vsip_vcreate_f(m + n - 1, VSIP_MEM_NONE);
//
// vsip_corr1d_f(corr, input, reference, result);

// Clean up when done
vsip_corr1d_destroy_f(corr);

```

Notes

- The correlation object should be destroyed with `vsip_dcorr1d_destroy_p` when no longer needed.
- The choice of support affects the length of the output correlation vector:
 - `VSIP_SUPPORT_FULL`: Output length is $n + m - 1$
 - `VSIP_SUPPORT_SAME`: Output length is n
 - `VSIP_SUPPORT_MIN`: Output length is $|n - m| + 1$
- The `ntimes` parameter helps the library optimize memory allocation for repeated use.

5.2.6 vsip_dcorr1d_destroy_p - Destroy 1D Correlation Object

```
int vsip_corr1d_destroy_f(vsip_corr1d_f *cor);
int vsip_corr1d_destroy_d(vsip_corr1d_d *cor);
int vsip_ccorr1d_destroy_f(vsip_ccorr1d_f *cor);
int vsip_ccorr1d_destroy_d(vsip_ccorr1d_d *cor);
```

Description

This function releases all memory and resources associated with a 1D correlation object that was previously created with `vsip_dcorr1d_create_p`. It is essential to call this function when you no longer need the correlation object to prevent memory leaks in your signal processing applications.

Parameters

- `vsip_dcorr1d_p* cor`: Pointer to the 1D correlation object to be destroyed.

Return Value

- Returns 0.

Example

```
vsip_corr1d_f *corr;
int status;
vsip_length m = 1024; // Input signal length
vsip_length n = 64;   // Reference signal length

// Create correlation object
corr = vsip_corr1d_create_f(m, n, VSIP_SUPPORT_FULL, 100, VSIP_ALG_TIME);
if (corr == NULL) {
    fprintf(stderr, "Error: Could not create correlation object\n");
    return -1;
}

// Use the correlation object for your signal processing
// vsip_vview_f *input = vsip_vcreate_f(m, VSIP_MEM_NONE);
// vsip_vview_f *reference = vsip_vcreate_f(n, VSIP_MEM_NONE);
// vsip_vview_f *result = vsip_vcreate_f(m + n - 1, VSIP_MEM_NONE);
// vsip_corr1d_f(corr, input, reference, result);

// Destroy correlation object when done
vsip_corr1d_destroy_f(corr);
```

5.2.7 vsip_dcorr1d_getattr_p - Get 1D Correlation Object Attributes

```
typedef struct _vsip_corr1d_attr_f {
    vsip_scalar_vi    ref_len;  // Reference length
    vsip_scalar_vi    data_len; // Data length
    vsip_support_region support; // Support type
    vsip_scalar_vi    lag_len;  // Lag length
} vsip_corr1d_attr_f;

/* same for ccorr1d */
typedef vsip_corr1d_attr_f vsip_ccorr1d_attr_f;

void vsip_corr1d_getattr_f(const vsip_corr1d_f *cor, vsip_corr1d_attr_f *attr);
void vsip_corr1d_getattr_d(const vsip_corr1d_d *cor, vsip_corr1d_attr_d *attr);
void vsip_ccorr1d_getattr_f(const vsip_ccorr1d_f *cor, vsip_ccorr1d_attr_f *attr);
void vsip_ccorr1d_getattr_d(const vsip_ccorr1d_d *cor, vsip_ccorr1d_attr_d *attr);
```

Description

This function retrieves the attributes of a 1D correlation object and stores them in the provided attribute structure.

Parameters

- `const vsip_dcorr1d_p* cor`: Pointer to the 1D correlation object created with `vsip_dcorr1d_create_p`.
- `vsip_dcorr1d_attr_p* attr`: Pointer to the attribute structure where the correlation object attributes will be stored.

Example

```
vsip_corr1d_f *corr;
vsip_corr1d_attr_f attr;
vsip_length m = 1024; // Input signal length
vsip_length n = 64;   // Reference signal length

// Create correlation object
corr = vsip_corr1d_create_f(m, n, VSIP_SUPPORT_FULL, 100, VSIP_ALG_TIME);
if (corr == NULL) {
    fprintf(stderr, "Error: Could not create correlation object\n");
    return;
}

// Get the attributes of the correlation object
vsip_corr1d_getattr_f(corr, &attr);

printf("1D Correlation Object Attributes:\n");
printf("  Ref length: %lu\n", attr.ref_len);
printf("  Data length: %lu\n", attr.data_len);
printf("  Support region: %d\n", attr.support);
printf("  Lag length: %lu\n", attr.lag_len);

// Clean up
vsip_corr1d_destroy_f(corr);
```

5.2.8 vsip_dcorrelate1d_p - Compute 1D Correlation

```
typedef enum _visp_bias {
    VSIP_BIASED    = 0,
    VSIP_UNBIASED  = 1
} visp_bias;

void vsip_correlate1d_f(const vsip_corr1d_f *cor, visp_bias bias, const vsip_vview_f *h,
                      const vsip_vview_f *x, const vsip_vview_f *y);
void vsip_correlate1d_d(const vsip_corr1d_d *cor, visp_bias bias, const vsip_vview_d *h,
                      const vsip_vview_d *x, const vsip_vview_d *y);
void vsip_ccorrelate1d_f(const vsip_ccorr1d_f *cor, visp_bias bias, const vsip_cvview_f *h,
                      const vsip_cvview_f *x, const vsip_cvview_f *y);
void vsip_ccorrelate1d_d(const vsip_ccorr1d_d *cor, visp_bias bias, const vsip_cvview_d *h,
                      const vsip_cvview_d *x, const vsip_cvview_d *y);
```

Description

This function computes the one-dimensional correlation between an input signal x and a reference signal h using the pre-configured correlation object. The result is stored in the output vector y . The correlation operation computes:

$$y_n = \sum_k h_k \cdot x_{n+k}$$

The exact form depends on the support region specified when creating the correlation object and the bias option.

Parameters

- `const vsip_dcorr1d_p* cor`: Pointer to the 1D correlation object created with `vsip_dcorr1d_create_p`.
- `visp_bias bias`: Bias option for the correlation:
 - `VSIP_NOBIAS`: No bias applied
 - `VSIP_BIASED`: Bias applied (normalization)
- `const vsip_dvview_p* h`: Reference signal vector of length n .
- `const vsip_dvview_p* x`: Input signal vector of length m .
- `const vsip_dvview_p* y`: Output correlation vector. Its length depends on the support region specified in the correlation object.

Example

```
vsip_corr1d_f *corr;
vsip_vview_f *h, *x, *y;
vsip_length m = 1024; // Input signal length
vsip_length n = 64;   // Reference signal length
vsip_length y_len;    // Output length

// Create correlation object for full correlation
corr = vsip_corr1d_create_f(m, n, VSIP_SUPPORT_FULL, 100, VSIP_ALG_TIME);
if (corr == NULL) {
    fprintf(stderr, "Error: Could not create correlation object\n");
    return;
}

// Determine output length based on support region
vsip_corr1d_attr_f attr;
vsip_corr1d_getattr_f(corr, &attr);
y_len = (attr.support == VSIP_SUPPORT_FULL) ? m + n - 1 :
        (attr.support == VSIP_SUPPORT_SAME) ? m :
        abs(m - n) + 1;
```

```

// Create vectors
h = vsip_vcreate_f(n, VSIP_MEM_NONE);    // Reference signal
x = vsip_vcreate_f(m, VSIP_MEM_NONE);    // Input signal
y = vsip_vcreate_f(y_len, VSIP_MEM_NONE); // Output correlation

// Initialize reference and input signals
// vsip_vramp_f(0.0f, 1.0f, h); // Example: linear ramp for reference
// vsip_vramp_f(0.0f, 0.5f, x); // Example: linear ramp for input

// Compute correlation without bias
vsip_correlate1d_f(corr, VSIP_NOBIAS, h, x, y);

// Compute correlation with bias (normalized)
vsip_correlate1d_f(corr, VSIP_BIASED, h, x, y);

// Clean up
vsip_corr1d_destroy_f(corr);
vsip_valldestroy_f(h);
vsip_valldestroy_f(x);
vsip_valldestroy_f(y);

```

Notes

- The input vectors h and x must have lengths matching those specified when the correlation object was created.
- The output vector y must have the appropriate length based on the support region:
 - VSIP_SUPPORT_FULL: $n + m - 1$
 - VSIP_SUPPORT_SAME: m
 - VSIP_SUPPORT_MIN: $|n - m| + 1$
- The bias option affects the normalization of the result:
 - VSIP_NOBIAS: No normalization applied
 - VSIP_BIASED: Result is normalized

5.3 Window Functions

5.3.1 vsip_vcreate_blackman_p - Create a Blackman Window Vector

```
vsip_vview_f* vsip_vcreate_blackman_f(vsip_length n, vsip_memory_hint hint);
vsip_vview_d* vsip_vcreate_blackman_d(vsip_length n, vsip_memory_hint hint);
```

Description

This function creates and initializes a vector with coefficients of a Blackman window of length n . The Blackman window is defined by the formula:

$$w[k] = 0.42 - 0.5 \cos\left(\frac{2\pi k}{n-1}\right) + 0.08 \cos\left(\frac{4\pi k}{n-1}\right), \quad 0 \leq k < n$$

Parameters

- `vsip_length n`: The length of the window (number of elements in the vector).
- `vsip_memory_hint hint`: Memory allocation hint that can be used to optimize memory access:
 - `VSIP_MEM_NONE` - No memory hint
 - `VSIP_MEM_RDONLY` - The memory is to be used read-only
 - `VSIP_MEM_CONST` - The memory will hold constants
 - `VSIP_MEM_SHARED` - The memory will be shared
 - `VSIP_MEM_SHARED_RDONLY` - The memory will be shared and is read-only
 - `VSIP_MEM_SHARED_CONST` - The memory will be shared and will hold constants

Return Value

- On success, returns a pointer to the newly created and initialized vector containing the Blackman window coefficients.
- On error (e.g., if memory allocation fails), returns `NULL`.

Example

```
vsip_vview_f *blackman_window;
vsip_length i, n = 64;

// Create a Blackman window of length 64
blackman_window = vsip_vcreate_blackman_f(n, VSIP_MEM_NONE);

if (blackman_window == NULL) {
    // Handle error
}

// Print the first 10 coefficients
printf("First 10 Blackman window coefficients:\n");
for (i = 0; i < 10; i++) {
    printf("%2ld: %f\n", i, vsip_vget_f(blackman_window, i));
}

// Use the window in a signal processing application
// For example, apply it to a signal vector
vsip_vview_f *signal = vsip_vcreate_f(n, VSIP_MEM_NONE);
vsip_vview_f *windowed_signal = vsip_vcreate_f(n, VSIP_MEM_NONE);

// Initialize signal with some values...
// vsip_vfill_f(signal, 1.0f); // Example: constant signal

// Apply the window: windowed_signal = signal * blackman_window
```

```
vsip_vmul_f(signal, blackman_window, windowed_signal);
```

```
// Clean up
```

```
vsip_valldestroy_f(blackman_window);
```

```
vsip_valldestroy_f(signal);
```

```
vsip_valldestroy_f(windowed_signal);
```

Notes

- The window is symmetric for even-length vectors and nearly symmetric for odd-length vectors.

5.3.2 vsip_vcreate_kaiser_p - Create a Kaiser Window Vector

```
vsip_vview_f* vsip_vcreate_kaiser_f(vsip_length n, vsip_scalar_f beta, vsip_memory_hint hint);
vsip_vview_d* vsip_vcreate_kaiser_d(vsip_length n, vsip_scalar_d beta, vsip_memory_hint hint);
```

Description

This function creates and initializes a vector with coefficients of a Kaiser window of length n . The Kaiser window is defined by:

$$w[k] = \frac{I_0\left(\beta\sqrt{1 - \left(\frac{2k}{n-1} - 1\right)^2}\right)}{I_0(\beta)}, \quad 0 \leq k < n$$

where I_0 is the zeroth-order modified Bessel function of the first kind.

Parameters

- `vsip_length n`: Length of the window (number of elements in the vector).
- `vsip_scalar_p beta`: Shape parameter that controls the trade-off between main lobe width and side lobe attenuation, β .
- `vsip_memory_hint hint`: Memory allocation hint:
 - `VSIP_MEM_NONE` - No memory hint
 - `VSIP_MEM_RDONLY` - The memory is to be used read-only
 - `VSIP_MEM_CONST` - The memory will hold constants
 - `VSIP_MEM_SHARED` - The memory will be shared
 - `VSIP_MEM_SHARED_RDONLY` - The memory will be shared and is read-only
 - `VSIP_MEM_SHARED_CONST` - The memory will be shared and will hold constants

Return Value

- On success, returns a pointer to the newly created and initialized vector containing the Kaiser window coefficients.
- On error, returns `NULL`.

Example

```
vsip_vview_f *kaiser_win;
vsip_length n = 64;
vsip_scalar_f beta = 6.0f; // Moderate side lobe suppression

// Create a Kaiser window
kaiser_win = vsip_vcreate_kaiser_f(n, beta, VSIP_MEM_NONE);

if (kaiser_win == NULL) {
    // Handle error
}

// Use the window in an application
// For example, apply it to a signal
vsip_vview_f *signal = vsip_vcreate_f(n, VSIP_MEM_NONE);
vsip_vview_f *windowed_signal = vsip_vcreate_f(n, VSIP_MEM_NONE);

// Initialize signal...
// vsip_vramp_f(0.0f, 1.0f, signal);

// Apply the window
vsip_vmul_f(signal, kaiser_win, windowed_signal);
```

```
// Clean up  
vsip_valldestroy_f(kaiser_win);  
vsip_valldestroy_f(signal);  
vsip_valldestroy_f(windowed_signal);
```

Notes

- The Kaiser window is symmetric for even-length vectors and nearly symmetric for odd-length vectors.
- Common β values and their approximate side lobe attenuations:
 - $\beta = 0$: Rectangular window (13 dB)
 - $\beta = 3$: 30 dB side lobe attenuation
 - $\beta = 6$: 50 dB side lobe attenuation
 - $\beta = 8.6$: 60 dB side lobe attenuation

5.3.3 vsip_vcreate_cheby_p - Create a Chebyshev Window Vector

```
vsip_vview_f* vsip_vcreate_cheby_f(vsip_length n, vsip_scalar_f ripple, vsip_memory_hint hint);
vsip_vview_d* vsip_vcreate_cheby_d(vsip_length n, vsip_scalar_d ripple, vsip_memory_hint hint);
```

Description

This function creates and initializes a vector with coefficients of a Chebyshev (Dolph-Chebyshev) window of length n . The Chebyshev window is designed to have equal ripple in the passband and is optimal in the sense that it minimizes the main lobe width for a given side lobe level.

The ripple parameter specifies the side lobe level in decibels (dB), with typical values ranging from 40 to 120 dB. Higher ripple values result in better side lobe suppression but wider main lobes.

Parameters

- `vsip_length n`: Length of the window (number of elements in the vector).
- `vsip_scalar_f ripple`: Side lobe level in dB (typically 40-120 dB).
- `vsip_memory_hint hint`: Memory allocation hint:
 - `VSIP_MEM_NONE` - No memory hint
 - `VSIP_MEM_RDONLY` - The memory is to be used read-only
 - `VSIP_MEM_CONST` - The memory will hold constants
 - `VSIP_MEM_SHARED` - The memory will be shared
 - `VSIP_MEM_SHARED_RDONLY` - The memory will be shared and is read-only
 - `VSIP_MEM_SHARED_CONST` - The memory will be shared and will hold constants

Return Value

- On success, returns a pointer to the newly created and initialized vector containing the Chebyshev window coefficients.
- On error, returns `NULL`.

Example

```
vsip_vview_f *cheby_win;
vsip_length n = 64;
vsip_scalar_f ripple = 60.0f; // 60 dB side lobe attenuation

// Create a Chebyshev window
cheby_win = vsip_vcreate_cheby_f(n, ripple, VSIP_MEM_NONE);

if (cheby_win == NULL) {
    // Handle error
}

// Use the window in an application
// For example, apply it to a signal
vsip_vview_f *signal = vsip_vcreate_f(n, VSIP_MEM_NONE);
vsip_vview_f *windowed_signal = vsip_vcreate_f(n, VSIP_MEM_NONE);

// Initialize signal...
// vsip_vramp_f(0.0f, 1.0f, signal);

// Apply the window
vsip_vmul_f(signal, cheby_win, windowed_signal);

// Clean up
```

```
vsip_vdestroy_f(cheby_win);  
vsip_vdestroy_f(signal);  
vsip_vdestroy_f(windowed_signal);
```

Notes

- The Chebyshev window provides the narrowest main lobe for a given side lobe level.
- The window is symmetric for even-length vectors and nearly symmetric for odd-length vectors.
- Common ripple values:
 - 40 dB: Moderate side lobe suppression
 - 60 dB: Good side lobe suppression
 - 80 dB: Excellent side lobe suppression
 - 100 dB: Very high side lobe suppression

5.3.4 vsip_vcreate_hanning_p - Create a Hanning Window Vector

```
vsip_vview_f* vsip_vcreate_hanning_f(vsip_length n, vsip_memory_hint hint);
vsip_vview_d* vsip_vcreate_hanning_d(vsip_length n, vsip_memory_hint hint);
```

Description

This function creates and initializes a vector with coefficients of a Hanning window (also known as Hann window) of length n . The Hanning window is defined by:

$$w[k] = 0.5 \left(1 - \cos \left(\frac{2\pi k}{n-1} \right) \right), \quad 0 \leq k < n$$

Parameters

- `vsip_length n`: Length of the window (number of elements in the vector).
- `vsip_memory_hint hint`: Memory allocation hint:
 - `VSIP_MEM_NONE` - No memory hint
 - `VSIP_MEM_RDONLY` - The memory is to be used read-only
 - `VSIP_MEM_CONST` - The memory will hold constants
 - `VSIP_MEM_SHARED` - The memory will be shared
 - `VSIP_MEM_SHARED_RDONLY` - The memory will be shared and is read-only
 - `VSIP_MEM_SHARED_CONST` - The memory will be shared and will hold constants

Return Value

- On success, returns a pointer to the newly created and initialized vector containing the Hanning window coefficients.
- On error (e.g., if memory allocation fails), returns `NULL`.

Example

```
vsip_vview_f *hanning_win;
vsip_length n = 64;

// Create a Hanning window
hanning_win = vsip_vcreate_hanning_f(n, VSIP_MEM_NONE);

if (hanning_win == NULL) {
    // Handle error
}

// Print first 5 coefficients
printf("First 5 Hanning window coefficients:\n");
for (int i = 0; i < 5; i++) {
    printf("%f\n", vsip_vget_f(hanning_win, i));
}

// Use the window in a signal processing application
// For example, apply it to a signal vector
vsip_vview_f *signal = vsip_vcreate_f(n, VSIP_MEM_NONE);
vsip_vview_f *windowed_signal = vsip_vcreate_f(n, VSIP_MEM_NONE);

// Initialize signal with some values (e.g., a sine wave)
// vsip_vramp_f(0.0f, 1.0f, signal);

// Apply the window: windowed_signal = signal * hanning_win
```

```
vsip_vmul_f(signal, hanning_win, windowed_signal);  
  
// Clean up  
vsip_valldestroy_f(hanning_win);  
vsip_valldestroy_f(signal);  
vsip_valldestroy_f(windowed_signal);
```

Notes

- The Hanning window is symmetric for even-length vectors and nearly symmetric for odd-length vectors.

5.4 FIR

5.4.1 vsip_dfir_create_p - Create a FIR Filter

```
typedef enum _vsip_symmetry {
    VSIP_NONSYM      = 0,
    VSIP_SYM_EVEN_LEN_ODD  = 1,
    VSIP_SYM_EVEN_LEN_EVEN = 2
} vsip_symmetry;

typedef enum _vsip_obj_state {
    VSIP_STATE_NO_SAVE = 1,
    VSIP_STATE_SAVE    = 2
} vsip_obj_state;

typedef enum _vsip_alg_hint {
    VSIP_ALG_TIME    = 0,
    VSIP_ALG_SPACE   = 1,
    VSIP_ALG_NOISE    = 2
} vsip_alg_hint;

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);
vsip_fir_d *vsip_fir_create_d(const vsip_vview_d *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);
vsip_cfir_d *vsip_cfir_create_d(const vsip_cvview_d *kernel, vsip_symmetry symm,
                                vsip_length n, vsip_length d,
                                vsip_obj_state state,
                                vsip_length ntimes, vsip_alg_hint hint);
```

Description

This function creates a FIR (Finite Impulse Response) filter with the specified kernel, symmetry, length, decimation factor, state, number of times to apply the filter, and algorithm hint.

Parameters

- `const vsip_d vview_p* kernel`: Pointer to the kernel vector view.
- `vsip_symmetry symm`: Symmetry of the filter kernel.
 - `VSIP_NOSYM` - No symmetry
 - `VSIP_SYM_EVEN_LEN_ODD` - Odd symmetry
 - `VSIP_SYM_EVEN_LEN_EVEN` - Even symmetry
- `vsip_length n`: Length of the filter.
- `vsip_length d`: Decimation factor.
- `vsip_obj_state state`: State of the filter object.
 - `VSIP_STATE_NO_SAVE` - Do not save state
 - `VSIP_STATE_SAVE` - Save state
- `vsip_length ntimes`: Number of times to apply the filter.

- `vsip_alg_hint hint`: Algorithm hint for the filter.
 - `VSIP_ALG_TIME` - Optimize for time
 - `VSIP_ALG_SPACE` - Optimize for memory usage
 - `VSIP_ALG_NOISE` - Optimize for noise

Return Value

- On success, a pointer to the newly created FIR filter object is returned.
- On error, `NULL` is returned.

Example

```
vsip_vview_f *kernel_view;  
vsip_symmetry symm = VSIP_NONSYM;  
vsip_length length = 10;  
vsip_length decimation = 1;  
vsip_obj_state state = VSIP_STATE_SAVE;  
vsip_length ntimes = 1;  
vsip_alg_hint hint = VSIP_ALG_TIME;  
vsip_fir_f *fir_filter;  
  
// Assuming kernel_view has been properly initialized  
fir_filter = vsip_fir_create_f(kernel_view, symm, length, decimation, state, ntimes, hint);  
  
if (fir_filter == NULL) {  
    // Handle error  
}
```

5.4.2 vsip_dfir_reset_p - Reset a FIR Filter

```
void vsip_fir_reset_f(vsip_fir_f *filt);  
void vsip_fir_reset_d(vsip_fir_d *filt);  
void vsip_cfir_reset_f(vsip_cfir_f *filt);  
void vsip_cfir_reset_d(vsip_cfir_d *filt);
```

Description

This function resets the specified FIR filter to its initial state.

Parameters

- vsip_dfir_p* filt: Pointer to the FIR filter to be reset.

Example

```
vsip_fir_f *fir_filter;  
  
// Assuming fir_filter has been properly initialized  
vsip_fir_reset_f(fir_filter);
```

5.4.3 vsip_dfir_getattr_p - Get Attributes of a FIR Filter

```
typedef struct _vsip_fir_attr_f{
    vsip_scalar_vi kernel_len;
    vsip_symmetry symm;
    vsip_scalar_vi in_len;
    vsip_scalar_vi out_len;
    vsip_length decimation;
    vsip_obj_state state;
} vsip_fir_attr_f;

void vsip_fir_getattr_f(const vsip_fir_f *fir, vsip_fir_attr_f *attr);
void vsip_fir_getattr_d(const vsip_fir_d *fir, vsip_fir_attr_d *attr);
void vsip_cfir_getattr_f(const vsip_cfir_f *fir, vsip_cfir_attr_f *attr);
void vsip_cfir_getattr_d(const vsip_cfir_d *fir, vsip_cfir_attr_d *attr);
```

Description

This function retrieves the attributes of the specified FIR filter and stores them in the structure pointed to by attr.

Parameters

- const vsip_dfir_p* fir: Pointer to the FIR filter.
- vsip_dfir_attr_p* attr: Pointer to a structure where the attributes will be stored.

Example

```
vsip_fir_f *fir_filter;
vsip_fir_attr_f attributes;

// Assuming fir_filter has been properly initialized
vsip_fir_getattr_f(fir_filter, &attributes);

// The attributes of the FIR filter are now stored in 'attributes'
```

5.4.4 vsip_dfirflt_p - Apply a FIR Filter to a Vector View

```
int vsip_firflt_f(vsip_fir_f *fir, const vsip_vview_f *x, const vsip_vview_f *y);
int vsip_firflt_d(vsip_fir_d *fir, const vsip_vview_d *x, const vsip_vview_d *y);
int vsip_cfirflt_d(vsip_cfir_d *fir, const vsip_cvview_d *x, const vsip_cvview_d *y);
int vsip_cfirflt_f(vsip_cfir_f *fir, const vsip_cvview_f *x, const vsip_cvview_f *y);
```

Description

This function applies the specified FIR filter to the input vector view `x` and stores the result in the output vector view `y`.

Parameters

- `vsip_dfir_p* fir`: Pointer to the FIR filter.
- `const vsip_dvview_p* x`: Pointer to the input vector view.
- `const vsip_dvview_p* y`: Pointer to the output vector view.

Return Value

- Returns 0 on success.
- Returns a non-zero value on error.

Example

```
vsip_fir_f *fir_filter;
vsip_vview_f *input_vector;
vsip_vview_f *output_vector;
int result;

// Assuming fir_filter, input_vector, and output_vector have been properly initialized
result = vsip_firflt_f(fir_filter, input_vector, output_vector);

if (result != 0) {
    // Handle error
}
```

5.4.5 vsip_dfir_destroy_p - Destroy a FIR Filter

```
int vsip_fir_destroy_f(vsip_fir_f *filt);
int vsip_fir_destroy_d(vsip_fir_d *filt);
int vsip_cfir_destroy_f(vsip_cfir_f *filt);
int vsip_cfir_destroy_d(vsip_cfir_d *filt);
```

Description

This function destroys the specified FIR filter and frees associated resources.

Parameters

- `vsip_dfir_p * filt`: Pointer to the FIR filter to be destroyed.

Return Value

- Returns 0 on success.
- Returns a non-zero value on error.

Example

```
vsip_fir_f *fir_filter;
int result;

// Assuming fir_filter has been properly initialized
result = vsip_fir_destroy_f(fir_filter);

if (result != 0) {
    // Handle error
}
```

5.5 Miscellaneous Signal Processing Functions

5.5.1 vsip_vhisto_p - Compute Histogram of a Vector View

```
typedef enum _vsip_hist_opt {
    VSIP_HIST_RESET = 1,
    VSIP_HIST_ACCUM = 2
} vsip_hist_opt;

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_d(const vsip_vview_d *src, vsip_scalar_d min_bin,
                  vsip_scalar_d max_bin, vsip_hist_opt opt,
                  const vsip_vview_d *hist);
```

Description

This function computes the histogram of the elements in the vector view `src` and stores the result in the vector view `hist`. The histogram is computed over the range `[min_bin, max_bin]` with the specified binning options `opt`.

Parameters

- `const vsip_vview_p* src`: Pointer to the source vector view.
- `vsip_scalar_p min_bin`: The minimum value of the histogram bins.
- `vsip_scalar_p max_bin`: The maximum value of the histogram bins.
- `vsip_hist_opt opt`: Options for histogram computation.
 - `VSIP_HIST_RESET` - Reset histogram and compute new
 - `VSIP_HIST_ACCUM` - Accumulate with previous
- `const vsip_vview_f* hist`: Pointer to the destination vector view where the histogram will be stored.

Example

```
vsip_vview_f *src_vector_view;
vsip_scalar_f min_bin = 0.0;
vsip_scalar_f max_bin = 10.0;
vsip_hist_opt hist_options = VSIP_HIST_ACCUM;
vsip_vview_f *hist_vector_view;

// Assuming src_vector_view and hist_vector_view have been properly initialized
vsip_vhisto_f(src_vector_view, min_bin, max_bin, hist_options, hist_vector_view);
```


Chapter 6

Linear Algebra Functions

6.1 Matrix and Vector Operations

6.1.1 vsip_dvdot_p - Compute the Dot Product of Two Vector Views

```
vsip_scalar_f vsip_vdot_f(const vsip_vview_f* a, const vsip_vview_f* b);
vsip_scalar_d vsip_vdot_d(const vsip_vview_d* a, const vsip_vview_d* b);
vsip_cscalar_f vsip_cvdot_f(const vsip_cvview_f* a, const vsip_cvview_f* b);
vsip_cscalar_d vsip_cvdot_d(const vsip_cvview_d* a, const vsip_cvview_d* b);
```

Description

This function computes the dot product of the vector views `a` and `b` and returns it. The dot product is computed as the sum of the element-wise products of the corresponding elements in the two vectors.

$$\sum_i^n a_i b_i$$

Parameters

- `const vsip_dvview_p* a`: Pointer to the first vector view.
- `const vsip_dvview_p* b`: Pointer to the second vector view.

Return Value

- The dot product of the two vector views.

Example

```
vsip_cvview_f *complex_vector_a;
vsip_cvview_f *complex_vector_b;
vsip_cscalar_f dot_product;
```

```
// Assuming complex_vector_a and complex_vector_b have been properly initialized
dot_product = vsip_cvdot_f(complex_vector_a, complex_vector_b);
```

6.1.2 vsip_cvjdot_p - Compute the Conjugate Dot Product of Two Complex Vector Views

```
vsip_cscalar_f vsip_cvjdot_f(const vsip_cvview_f* a, const vsip_cvview_f* b);
vsip_cscalar_d vsip_cvjdot_d(const vsip_cvview_d* a, const vsip_cvview_d* b);
```

Description

This function computes the conjugate dot product of the complex vector views `a` and `b` and returns it. The conjugate dot product is computed as the sum of the element-wise products of the corresponding elements in the first vector and the conjugate of the elements in the second vector.

$$\sum_i^n a_i \overline{b_i}$$

Parameters

- `const vsip_cvview_p*` `a`: Pointer to the first complex vector view.
- `const vsip_cvview_p*` `b`: Pointer to the second complex vector view.

Return Value

- The conjugate dot product of the two complex vector views.

Example

```
vsip_cvview_f *complex_vector_a;
vsip_cvview_f *complex_vector_b;
vsip_cscalar_f conjugate_dot_product;
```

```
// Assuming complex_vector_a and complex_vector_b have been properly initialized
conjugate_dot_product = vsip_cvjdot_f(complex_vector_a, complex_vector_b);
```

6.1.3 vsip_dvouter_p - Outer Product of Two Vectors

```
void vsip_vouter_f(vsip_scalar_f alpha, const vsip_vview_f *x, const vsip_vview_f *y, const vsip_mview_f *r)
void vsip_vouter_d(vsip_scalar_d alpha, const vsip_vview_d *x, const vsip_vview_d *y, const vsip_mview_d *r)
void vsip_cvouter_f(vsip_cscalar_f alpha, const vsip_cvview_f *x, const vsip_cvview_f *y, const vsip_cmview_f *r)
void vsip_cvouter_d(vsip_cscalar_d alpha, const vsip_cvview_d *x, const vsip_cvview_d *y, const vsip_cmview_d *r)
```

Description

This function computes the outer product of two vectors x and y , scaled by α , and stores the result in matrix r . The outer product is defined as:

$$r_{i,j} = \alpha \cdot x_i \cdot y_j$$

for all i and j , where x_i is the i -th element of vector x and y_j is the j -th element of vector y .

Parameters

- vsip_dscalar_p alpha: Scalar multiplier for the outer product.
- const vsip_dvview_p* x: Pointer to the first input vector of length m .
- const vsip_dvview_p* y: Pointer to the second input vector of length n .
- const vsip_dmview_p* r: Pointer to the output matrix of size $m \times n$ that will store the result.

Example

```
vsip_vview_f *x, *y;
vsip_mview_f *r;
vsip_length m = 5, n = 4;

// Create vectors and matrix
x = vsip_vcreate_f(m, VSIP_MEM_NONE);
y = vsip_vcreate_f(n, VSIP_MEM_NONE);
r = vsip_mcreate_f(m, n, VSIP_ROW, VSIP_MEM_NONE);

// Initialize vectors
vsip_vramp_f(1.0f, 1.0f, x); // x = [1, 2, 3, 4, 5]
vsip_vramp_f(0.5f, 0.5f, y); // y = [0.5, 1.0, 1.5, 2.0]

// Compute outer product: r = x * y^T
vsip_vouter_f(1.0f, x, y, r);

// Print the resulting matrix
printf("Outer product result:\n");
for (vsip_index i = 0; i < m; i++) {
    for (vsip_index j = 0; j < n; j++) {
        printf("%8.2f ", vsip_mget_f(r, i, j));
    }
    printf("\n");
}

// Compute scaled outer product: r = 2.0 * x * y^T
vsip_vouter_f(2.0f, x, y, r);

// Clean up
vsip_valldestroy_f(x);
vsip_valldestroy_f(y);
vsip_malldestroy_f(r);
```

Notes

- The output matrix r must have dimensions $m \times n$ where m is the length of vector x and n is the length of vector y .
- The outer product is not commutative: $x \otimes y \neq y \otimes x$.
- If $\alpha = 0$, the result will be a zero matrix regardless of the input vectors.

6.1.4 vsip_dmtrans_p - Matrix Transposition

```
void vsip_mtrans_f(const vsip_mview_f *a, const vsip_mview_f *c);
void vsip_mtrans_d(const vsip_mview_d *a, const vsip_mview_d *c);
void vsip_cmtrans_f(const vsip_cmview_f *a, const vsip_cmview_f *c);
void vsip_cmtrans_d(const vsip_cmview_d *a, const vsip_cmview_d *c);
```

Description

This function computes the transpose of matrix A and stores the result in matrix C . The transpose operation exchanges the rows and columns of the matrix, such that element $c_{i,j}$ of the output matrix is equal to element $a_{j,i}$ of the input matrix.

For an $m \times n$ input matrix A , the output matrix C must be of size $n \times m$.

Parameters

- const vsip_dmview_p* a: Pointer to the input matrix of size $m \times n$.
- const vsip_dmview_p* c: Pointer to the output matrix of size $n \times m$ that will store the transposed result.

Example

```
vsip_mview_f *A, *C;
vsip_length m = 3, n = 4;

// Create input matrix (3x4)
A = vsip_mcreate_f(m, n, VSIP_ROW, VSIP_MEM_NONE);

// Initialize matrix A with some values
for (vsip_index i = 0; i < m; i++) {
    for (vsip_index j = 0; j < n; j++) {
        vsip_mput_f(A, i, j, (float)(i*n + j + 1));
    }
}

// Create output matrix (4x3) for the transpose
C = vsip_mcreate_f(n, m, VSIP_ROW, VSIP_MEM_NONE);

// Compute the transpose: C = A^T
vsip_mtrans_f(A, C);

// Print the original and transposed matrices
printf("Original matrix A (%lux%lu):\n", m, n);
for (vsip_index i = 0; i < m; i++) {
    for (vsip_index j = 0; j < n; j++) {
        printf("%6.1f ", vsip_mget_f(A, i, j));
    }
    printf("\n");
}

printf("\nTransposed matrix C (%lux%lu):\n", n, m);
for (vsip_index i = 0; i < n; i++) {
    for (vsip_index j = 0; j < m; j++) {
        printf("%6.1f ", vsip_mget_f(C, i, j));
    }
    printf("\n");
}

// Clean up
vsip_malldestroy_f(A);
vsip_malldestroy_f(C);
```

Notes

- The output matrix C must have dimensions $n \times m$ where the input matrix A has dimensions $m \times n$.
- For in-place transposition (when $m = n$), consider using `vsip_dmtransview_p` to create a transposed view.

6.1.5 vsip_cmherm_p - Matrix Hermitian

```
void vsip_cmherm_f(const vsip_cmview_f *a, const vsip_cmview_f *c);
void vsip_cmherm_d(const vsip_cmview_d *a, const vsip_cmview_d *c);
```

Description

This function computes the hermitian of a complex matrix A and stores the result in matrix C . The Hermitian operation exchanges the rows and columns of the matrix, such that element $c_{i,j}$ of the output matrix is equal to the conjugate of element $\overline{a_{j,i}}$ of the input matrix.

For an $m \times n$ input matrix A , the output matrix C must be of size $n \times m$.

Parameters

- `const vsip_cmview_p*` a : Pointer to the input matrix of size $m \times n$.
- `const vsip_cmview_p*` c : Pointer to the output matrix of size $n \times m$ that will store the Hermitian result.

Notes

- The output matrix C must have dimensions $n \times m$ where the input matrix A has dimensions $m \times n$.
- For in-place transposition (when $m = n$), consider using `vsip_dmtransview_p` to create a transposed view and use the conjugate of the elements.

6.1.6 vsip_dgemp_p - General Matrix Product

```
typedef enum _vsip_mat_op {
    VSIP_MAT_NTRANS = 0, // op(A) = A
    VSIP_MAT_TRANS  = 1, // op(A) = A^T
    VSIP_MAT_HERM   = 2, // op(A) = A^H (complex only)
    VSIP_MAT_CONJ   = 3  // op(X) = A^* (complex only)
} vsip_mat_op;

void vsip_gemp_f(vsip_scalar_f alpha, const vsip_mview_f *a, vsip_mat_op OpA, const vsip_mview_f *b, vsip_
void vsip_gemp_d(vsip_scalar_d alpha, const vsip_mview_d *a, vsip_mat_op OpA, const vsip_mview_d *b, vsip_
void vsip_cgemp_f(vsip_cscalar_f alpha, const vsip_cmview_f *a, vsip_mat_op OpA, const vsip_cmview_f *b, v
void vsip_cgemp_d(vsip_cscalar_d alpha, const vsip_cmview_d *a, vsip_mat_op OpA, const vsip_cmview_d *b, v
```

Description

This function performs a generalized matrix-matrix operation of the form:

$$R = \alpha \cdot \text{op}(A) \cdot \text{op}(B) + \beta \cdot R$$

where $\text{op}(X)$ can be X , X^T , or X^H .

Parameters

- `vsip_dscalar_p alpha`: Scalar multiplier for the matrix product.
- `const vsip_dmview_p* a`: First input matrix.
- `vsip_mat_op OpA`: Operation to perform on matrix A:
 - `VSIP_MAT_NTRANS`: Use A as is
 - `VSIP_MAT_TRANS`: Use the transpose of, A^T
 - `VSIP_MAT_HERM`: Use the conjugate transpose of A^H
 - `VSIP_MAT_CONJ`: Use the conjugate of A^*
- `const vsip_dmview_p* b`: Second input matrix.
- `vsip_mat_op OpB`: Operation to perform on matrix B.
 - `VSIP_MAT_NTRANS`: Use A as is
 - `VSIP_MAT_TRANS`: Use the transpose of, A^T
 - `VSIP_MAT_HERM`: Use the conjugate transpose of A^H
 - `VSIP_MAT_CONJ`: Use the conjugate of A^*
- `vsip_dscalar_p beta`: Scalar multiplier for matrix R.
- `const vsip_dmview_p* r`: Input/output matrix that contains the initial values and will store the result.

Example

```
vsip_mview_f *A, *B, *R;
vsip_length m = 3, n = 2, p = 4;

// Create matrices
A = vsip_mcreate_f(m, n, VSIP_ROW, VSIP_MEM_NONE);
B = vsip_mcreate_f(n, p, VSIP_ROW, VSIP_MEM_NONE);
R = vsip_mcreate_f(m, p, VSIP_ROW, VSIP_MEM_NONE);

// Initialize matrices with some values

// Basic matrix multiplication: R = A * B
vsip_gemp_f(1.0f, A, VSIP_MAT_NTRANS, B, VSIP_MAT_NTRANS, 0.0f, R);
```

```

// Matrix multiplication with scaling:  $R = 2.0 * A * B + R$ 
vsip_gemp_f(2.0f, A, VSIP_MAT_NTRANS, B, VSIP_MAT_NTRANS, 1.0f, R);

// Transpose operations:  $R = A^T * B$ 
vsip_gemp_f(1.0f, A, VSIP_MAT_TRANS, B, VSIP_MAT_NTRANS, 0.0f, R);

// Both transposed:  $R = A^T * B^T$ 
vsip_gemp_f(1.0f, A, VSIP_MAT_TRANS, B, VSIP_MAT_TRANS, 0.0f, R);

// Clean up
vsip_malldestroy_f(A);
vsip_malldestroy_f(B);
vsip_malldestroy_f(R);

```

Notes

- The dimensions of the matrices must be compatible with the operation:
 - If $\text{OpA} = \text{VSIP_MAT_NTRANS}$, rows of A must match rows of $\text{op}(B)$.
 - If $\text{OpA} = \text{VSIP_MAT_TRANS}$ or VSIP_MAT_HERM , columns of A must match rows of $\text{op}(B)$.
- The result matrix R must have dimensions compatible with the operation.
- The operation is not commutative: $A \cdot B \neq B \cdot A$ in general.
- Setting $\beta = 0$ results in R being overwritten with the matrix product.
- Setting $\beta = 1$ results in the matrix product being added to R .

6.1.7 vsip_dgems_p - General Matrix Scaling and Addition

```
typedef enum _vsip_mat_op {
    VSIP_MAT_NTRANS = 0, // op(A) = A
    VSIP_MAT_TRANS  = 1, // op(A) = A^T
    VSIP_MAT_HERM    = 2, // op(A) = A^H (complex only)
    VSIP_MAT_CONJ    = 3  // op(X) = A^* (complex only)
} vsip_mat_op;

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 *r)
void vsip_gems_d(vsip_scalar_d alpha, const vsip_mview_d *a, vsip_mat_op OpA, vsip_scalar_d beta, const vsip_mview_d *r)
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 *r)
void vsip_cgems_d(vsip_cscalar_d alpha, const vsip_cmview_d *a, vsip_mat_op OpA, vsip_cscalar_d beta, const vsip_cmview_d *r)
```

Description

This function performs a generalized matrix scaling and addition operation of the form:

$$R = \alpha \cdot \text{op}(A) + \beta \cdot R$$

where $\text{op}(A)$ can be A , A^T , or A^H .

Parameters

- `vsip_dscalar_p alpha`: Scalar multiplier for matrix A .
- `const vsip_dmview_p* a`: Input matrix.
- `vsip_mat_op OpA`: Operation to perform on matrix A .
 - `VSIP_MAT_NTRANS`: Use A as is
 - `VSIP_MAT_TRANS`: Use the transpose of, A^T
 - `VSIP_MAT_HERM`: Use the conjugate transpose of A^H
 - `VSIP_MAT_CONJ`: Use the conjugate of A^*
- `vsip_dscalar_p beta`: Scalar multiplier for matrix R .
- `const vsip_dmview_p* r`: Input/output matrix that contains the initial values and will store the result.

Example

```
vsip_mview_f *A, *R;
vsip_length m = 3, n = 3;

// Create matrices
A = vsip_mcreate_f(m, n, VSIP_ROW, VSIP_MEM_NONE);
R = vsip_mcreate_f(m, n, VSIP_ROW, VSIP_MEM_NONE);

// Initialize matrices with some values

// Basic scaling: R = 2.0 * A
vsip_gems_f(2.0f, A, VSIP_MAT_NTRANS, 0.0f, R);

// Scale and add: R = 1.5*A + R
vsip_gems_f(1.5f, A, VSIP_MAT_NTRANS, 1.0f, R);

// Transpose operation: R = A^T
vsip_gems_f(1.0f, A, VSIP_MAT_TRANS, 0.0f, R);

// Linear combination: R = 0.5*A + 0.5*R
vsip_gems_f(0.5f, A, VSIP_MAT_NTRANS, 0.5f, R);
```

```
// Overwrite with scaled transpose:  $R = 3.0 * A^T$ 
vsip_gems_f(3.0f, A, VSIP_MAT_TRANS, 0.0f, R);

// Clean up
vsip_malldestroy_f(A);
vsip_malldestroy_f(R);
```

Notes

- The dimensions of matrices A and R must be compatible with the operation:
 - If $\text{OpA} = \text{VSIP_MAT_NTRANS}$, rows of A must match rows of R .
 - If $\text{OpA} = \text{VSIP_MAT_TRANS}$ or VSIP_MAT_HERM , columns of A must match rows of R .
- This function performs the operation in-place on matrix R .
- Setting $\beta = 0$ results in R being overwritten with the scaled matrix.
- Setting $\beta = 1$ results in the scaled matrix being added to R .

6.1.8 vsip_dvmprod_p - Vector-Matrix Product

```
void vsip_vmprod_f(const vsip_vview_f *a, const vsip_mview_f *b, const vsip_vview_f *r);
void vsip_vmprod_d(const vsip_vview_d *a, const vsip_mview_d *b, const vsip_vview_d *r);
void vsip_cvmprod_f(const vsip_cvview_f *a, const vsip_cmview_f *b, const vsip_cvview_f *r);
void vsip_cvmprod_d(const vsip_cvview_d *a, const vsip_cmview_d *b, const vsip_cvview_d *r);
```

Description

This function computes the product of a vector and a matrix, storing the result in an output vector. The operation performed is:

$$r_i = \sum_{j=1}^n a_j \cdot b_{j,i}$$

for $i = 1, 2, \dots, m$, where a is a vector of length n , b is an $n \times m$ matrix, and r is the resulting vector of length m . This operation is equivalent to the matrix-vector product $r = a^T \cdot b$, where a^T is the transpose of vector a .

Parameters

- `const vsip_dvview_p* a`: Input vector of length n .
- `const vsip_dmview_p* b`: Input matrix of size $n \times m$.
- `const vsip_dvview_p* r`: Output vector of length m that will store the result.

Example

```
vsip_vview_f *a, *r;
vsip_mview_f *b;
vsip_length n = 4, m = 3;

// Create vector and matrices
a = vsip_vcreate_f(n, VSIP_MEM_NONE);
b = vsip_mcreate_f(n, m, VSIP_ROW, VSIP_MEM_NONE);
r = vsip_vcreate_f(m, VSIP_MEM_NONE);

// Initialize vector a and matrix b with some values
vsip_vramp_f(1.0f, 1.0f, a); // a = [1, 2, 3, 4]

// Initialize matrix b (4x3)
for (vsip_index i = 0; i < n; i++) {
    for (vsip_index j = 0; j < m; j++) {
        vsip_mput_f(b, i, j, (float)(i*m + j + 1));
    }
}

// Compute vector-matrix product: r = a^T * b
vsip_vmprod_f(a, b, r);

// Print the result
printf("Result vector r:\n");
for (vsip_index i = 0; i < m; i++) {
    printf("%8.2f ", vsip_vget_f(r, i));
}
printf("\n");

// Clean up
vsip_valldestroy_f(a);
vsip_malldestroy_f(b);
vsip_valldestroy_f(r);
```


Notes

- The input vector a must have length n .
- The input matrix b must have dimensions $n \times m$.
- The output vector r must have length m .
- This operation is equivalent to the matrix-vector product $r = a^T \cdot b$.
- This operation is not commutative: $a^T \cdot b \neq b \cdot a^T$.

6.1.9 vsip_dmvprod_p - Matrix-Vector Product

```
void vsip_mvprod_f(const vsip_mview_f *a, const vsip_vview_f *b, const vsip_vview_f *r);
void vsip_mvprod_d(const vsip_mview_d *a, const vsip_vview_d *b, const vsip_vview_d *r);
void vsip_cmvpord_f(const vsip_cmview_f *a, const vsip_cvview_f *b, const vsip_cvview_f *r);
void vsip_cmvpord_d(const vsip_cmview_d *a, const vsip_cvview_d *b, const vsip_cvview_d *r);
```

Description

This function computes the product of a matrix and a vector, storing the result in an output vector. The operation performed is:

$$r_i = \sum_{j=1}^n a_{i,j} \cdot b_j$$

for $i = 1, 2, \dots, m$, where a is an $m \times n$ matrix, b is a vector of length n , and r is the resulting vector of length m . This operation is equivalent to the matrix-vector product $r = a \cdot b$.

Parameters

- `const vsip_dmview_p* a`: Input matrix of size $m \times n$.
- `const vsip_dvview_p* b`: Input vector of length n .
- `const vsip_dvview_p* r`: Output vector of length m that will store the result.

Example

```
vsip_mview_f *A;
vsip_vview_f *b, *r;
vsip_length m = 4, n = 3;

// Create matrix and vectors
A = vsip_mcreate_f(m, n, VSIP_ROW, VSIP_MEM_NONE);
b = vsip_vcreate_f(n, VSIP_MEM_NONE);
r = vsip_vcreate_f(m, VSIP_MEM_NONE);

// Initialize matrix A and vector b with some values
// Initialize A (4x3 matrix)
for (vsip_index i = 0; i < m; i++) {
    for (vsip_index j = 0; j < n; j++) {
        vsip_mput_f(A, i, j, (float)(i*n + j + 1));
    }
}

// Initialize vector b
vsip_vramp_f(1.0f, 1.0f, b); // b = [1, 2, 3]

// Compute matrix-vector product: r = A * b
vsip_mvprod_f(A, b, r);

// Print the result
printf("Matrix A (%lux%lu):\n", m, n);
for (vsip_index i = 0; i < m; i++) {
    for (vsip_index j = 0; j < n; j++) {
        printf("%8.2f ", vsip_mget_f(A, i, j));
    }
    printf("\n");
}

printf("\nVector b (%lu):\n", n);
for (vsip_index i = 0; i < n; i++) {
```

```

    printf("%8.2f ", vsip_vget_f(b, i));
}
printf("\n");

printf("\nResult vector r (%lu):\n", m);
for (vsip_index i = 0; i < m; i++) {
    printf("%8.2f ", vsip_vget_f(r, i));
}
printf("\n");

// Clean up
vsip_malldestroy_f(A);
vsip_valldestroy_f(b);
vsip_valldestroy_f(r);

```

Notes

- The input matrix a must have dimensions $m \times n$.
- The input vector b must have length n .
- The output vector r must have length m .
- This operation is equivalent to the matrix-vector product $r = a \cdot b$.
- If you need to compute $b^T \cdot a$ (vector-matrix product), use `vsip_d vmprod_p` instead.

6.1.10 vsip_dmprod_p - Matrix-Matrix Product

```

void vsip_mprod_f(const vsip_mview_f *a, const vsip_mview_f *b, const vsip_mview_f *r);
void vsip_mprod_d(const vsip_mview_d *a, const vsip_mview_d *b, const vsip_mview_d *r);
void vsip_cmprod_f(const vsip_cmview_f *a, const vsip_cmview_f *b, const vsip_cmview_f *r);
void vsip_cmprod_d(const vsip_cmview_d *a, const vsip_cmview_d *b, const vsip_cmview_d *r);

```

Description

This function computes the matrix product of two matrices A and B , storing the result in matrix R . The operation performed is:

$$r_{i,j} = \sum_{k=1}^n a_{i,k} \cdot b_{k,j}$$

for all i and j , where A is an $m \times n$ matrix, B is an $n \times p$ matrix, and R is the resulting $m \times p$ matrix.

Parameters

- const vsip_dmview_p* a: First input matrix of size $m \times n$.
- const vsip_dmview_p* b: Second input matrix of size $n \times p$.
- const vsip_dmview_p* r: Output matrix of size $m \times p$ that will store the result.

Example

```

vsip_mview_f *A, *B, *R;
vsip_length m = 3, n = 2, p = 4;

// Create matrices
A = vsip_mcreate_f(m, n, VSIP_ROW, VSIP_MEM_NONE);
B = vsip_mcreate_f(n, p, VSIP_ROW, VSIP_MEM_NONE);
R = vsip_mcreate_f(m, p, VSIP_ROW, VSIP_MEM_NONE);

// Initialize matrices A and B with some values
// Initialize matrix A (3x2)
for (vsip_index i = 0; i < m; i++) {
    for (vsip_index j = 0; j < n; j++) {
        vsip_mput_f(A, i, j, (float)(i*n + j + 1));
    }
}

// Initialize matrix B (2x4)
for (vsip_index i = 0; i < n; i++) {
    for (vsip_index j = 0; j < p; j++) {
        vsip_mput_f(B, i, j, (float)(i*p + j + 1));
    }
}

// Compute matrix product: R = A * B
vsip_mprod_f(A, B, R);

// Print the matrices
printf("Matrix A (%lux%lu):\n", m, n);
for (vsip_index i = 0; i < m; i++) {
    for (vsip_index j = 0; j < n; j++) {
        printf("%8.2f ", vsip_mget_f(A, i, j));
    }
    printf("\n");
}

```

```

printf("\nMatrix B (%lux%lu):\n", n, p);
for (vsip_index i = 0; i < n; i++) {
    for (vsip_index j = 0; j < p; j++) {
        printf("%8.2f ", vsip_mget_f(B, i, j));
    }
    printf("\n");
}

printf("\nResult matrix R (%lux%lu):\n", m, p);
for (vsip_index i = 0; i < m; i++) {
    for (vsip_index j = 0; j < p; j++) {
        printf("%8.2f ", vsip_mget_f(R, i, j));
    }
    printf("\n");
}

// Clean up
vsip_malldestroy_f(A);
vsip_malldestroy_f(B);
vsip_malldestroy_f(R);

```

Notes

- The input matrices must have compatible dimensions: A must be $m \times n$ and B must be $n \times p$.
- The output matrix R must have dimensions $m \times p$.
- The matrix product is not commutative: $A \cdot B \neq B \cdot A$ in general.
- If you need to compute $A^T \cdot B$ or other variants, consider using `vsip_dgemp_p` instead.

6.1.11 vsip_dmprod_t_p - Matrix-Matrix Product with Transposition

```
void vsip_mprod_t_f(const vsip_mview_f *a, const vsip_mview_f *b, const vsip_mview_f *r);
void vsip_mprod_t_d(const vsip_mview_d *a, const vsip_mview_d *b, const vsip_mview_d *r);
void vsip_cmprod_t_f(const vsip_cmview_f *a, const vsip_cmview_f *b, const vsip_cmview_f *r);
void vsip_cmprod_t_d(const vsip_cmview_d *a, const vsip_cmview_d *b, const vsip_cmview_d *r);
```

Description

This function computes the product of matrix A and the transpose of matrix B , storing the result in matrix R . The operation performed is:

$$r_{i,j} = \sum_{k=1}^n a_{i,k} \cdot b_{j,k}$$

for all i and j , where A is an $m \times n$ matrix, B is an $p \times n$ matrix, and R is the resulting $m \times p$ matrix.

This operation is equivalent to the matrix product $R = A \cdot B^T$.

Parameters

- const vsip_dmview_p* a: First input matrix of size $m \times n$.
- const vsip_dmview_p* b: Second input matrix of size $p \times n$ (will be transposed in the operation).
- const vsip_dmview_p* r: Output matrix of size $m \times p$ that will store the result.

Example

```
vsip_mview_f *A, *B, *R;
vsip_length m = 3, n = 4, p = 2;

// Create matrices
A = vsip_mcreate_f(m, n, VSIP_ROW, VSIP_MEM_NONE); // 3x4 matrix
B = vsip_mcreate_f(p, n, VSIP_ROW, VSIP_MEM_NONE); // 2x4 matrix
R = vsip_mcreate_f(m, p, VSIP_ROW, VSIP_MEM_NONE); // 3x2 result matrix

// Initialize matrices A and B with some values
// Initialize matrix A (3x4)
for (vsip_index i = 0; i < m; i++) {
    for (vsip_index j = 0; j < n; j++) {
        vsip_mput_f(A, i, j, (float)(i*n + j + 1));
    }
}

// Initialize matrix B (2x4)
for (vsip_index i = 0; i < p; i++) {
    for (vsip_index j = 0; j < n; j++) {
        vsip_mput_f(B, i, j, (float)(i*n + j + 1));
    }
}

// Compute matrix product with transposition: R = A * B^T
vsip_mprod_t_f(A, B, R);

// Print the matrices
printf("Matrix A (%lux%lu):\n", m, n);
for (vsip_index i = 0; i < m; i++) {
    for (vsip_index j = 0; j < n; j++) {
        printf("%8.2f ", vsip_mget_f(A, i, j));
    }
    printf("\n");
}
```

```

printf("\nMatrix B (%lux%lu):\n", p, n);
for (vsip_index i = 0; i < p; i++) {
    for (vsip_index j = 0; j < n; j++) {
        printf("%8.2f ", vsip_mget_f(B, i, j));
    }
    printf("\n");
}

printf("\nResult matrix R = A * B^T (%lux%lu):\n", m, p);
for (vsip_index i = 0; i < m; i++) {
    for (vsip_index j = 0; j < p; j++) {
        printf("%8.2f ", vsip_mget_f(R, i, j));
    }
    printf("\n");
}

// Clean up
vsip_malldestroy_f(A);
vsip_malldestroy_f(B);
vsip_malldestroy_f(R);

```

Notes

- The input matrices must have compatible dimensions: Both A and B must have the same number of columns (n).
- The output matrix R must have dimensions $m \times p$, where m is the number of rows in A and p is the number of rows in B .
- This operation is equivalent to computing the covariance matrix when A and B contain centered data.
- If you need more flexibility in choosing which matrix to transpose, consider using `vsip_dgemv_p` instead.

6.1.12 vsip_cmprodh_p - Complex Matrix Product with Hermitian Transpose

```
void vsip_cmprodh_f(const vsip_cmview_f *a, const vsip_cmview_f *b, const vsip_cmview_f *r);
void vsip_cmprodh_d(const vsip_cmview_d *a, const vsip_cmview_d *b, const vsip_cmview_d *r);
```

Description

This function computes the product of a complex matrix A with the Hermitian transpose of a complex matrix B , storing the result in complex matrix R . The operation performed is:

$$r_{i,j} = \sum_{k=1}^n a_{i,k} \cdot \overline{b_{j,k}}$$

for all i and j , where A is an $m \times n$ complex matrix, B is a $p \times n$ complex matrix, and R is the resulting $m \times p$ complex matrix. $\overline{b_{j,k}}$ denotes the complex conjugate of $b_{j,k}$.

Parameters

- const vsip_cmview_p* a: First input matrix of size $m \times n$ (complex).
- const vsip_cmview_p* b: Second input matrix of size $p \times n$ (complex), which will be Hermitian transposed in the operation.
- const vsip_cmview_p* r: Output matrix of size $m \times p$ (complex) that will store the result.

Example

```
vsip_cmview_f *A, *B, *R;
vsip_length m = 2, n = 3, p = 2;

// Create complex matrices
A = vsip_cmcreate_f(m, n, VSIP_ROW, VSIP_MEM_NONE);
B = vsip_cmcreate_f(p, n, VSIP_ROW, VSIP_MEM_NONE);
R = vsip_cmcreate_f(m, p, VSIP_ROW, VSIP_MEM_NONE);

// Initialize matrices A and B with complex values
for (vsip_index i = 0; i < m; i++) {
    for (vsip_index j = 0; j < n; j++) {
        vsip_cscalar_f val = VSIP_CMPLX_F(i*n + j + 1, -(i*n + j + 1));
        vsip_cput_f(A, i, j, val);
    }
}

for (vsip_index i = 0; i < p; i++) {
    for (vsip_index j = 0; j < n; j++) {
        vsip_cscalar_f val = VSIP_CMPLX_F(i*n + j + 1, i*n + j + 2);
        vsip_cput_f(B, i, j, val);
    }
}

// Compute matrix product with Hermitian transpose: R = A * B^H
vsip_cmprodh_f(A, B, R);

// Print the result
printf("Result matrix R = A * B^H (%lux%lu):\n", m, p);
for (vsip_index i = 0; i < m; i++) {
    for (vsip_index j = 0; j < p; j++) {
        vsip_cscalar_f val = vsip_cmget_f(R, i, j);
        printf("(%.2f%+.2fi) ", val.r, val.i);
    }
    printf("\n");
}
```



```
// Clean up  
vsip_cmalldestroy_f(A);  
vsip_cmalldestroy_f(B);  
vsip_cmalldestroy_f(R);
```

Notes

- The input matrices must have compatible dimensions: Both matrices A and B must have the same number of columns (n).
- The output matrix R must have dimensions $m \times p$, where m is the number of rows in A and p is the number of rows in B .
- If you need more flexibility in choosing which matrix to transpose, consider using `vsip_cgemp_p` instead.

6.1.13 vsip_cmprodj_p - Complex Matrix Product with Conjugate

```
void vsip_cmprodj_f(const vsip_cmview_f *a, const vsip_cmview_f *b, const vsip_cmview_f *r);
void vsip_cmprodj_d(const vsip_cmview_d *a, const vsip_cmview_d *b, const vsip_cmview_d *r);
```

Description

This function computes the product of a complex matrix A with the element-wise conjugate of a complex matrix B , storing the result in complex matrix R . The operation performed is:

$$r_{i,j} = \sum_{k=1}^n a_{i,k} \cdot \overline{b_{k,j}}$$

for all i and j , where A is an $m \times n$ complex matrix, B is an $n \times p$ complex matrix, and R is the resulting $m \times p$ complex matrix. $\overline{b_{k,j}}$ denotes the complex conjugate of $b_{k,j}$.

Parameters

- `const vsip_cmview_p* a`: First input matrix of size $m \times n$ (complex).
- `const vsip_cmview_p* b`: Second input matrix of size $n \times p$ (complex), whose elements will be conjugated in the operation.
- `const vsip_cmview_p* r`: Output matrix of size $m \times p$ (complex) that will store the result.

Example

```
vsip_cmview_f *A, *B, *R;
vsip_length m = 2, n = 3, p = 2;

// Create complex matrices
A = vsip_cmcreate_f(m, n, VSIP_ROW, VSIP_MEM_NONE);
B = vsip_cmcreate_f(n, p, VSIP_ROW, VSIP_MEM_NONE);
R = vsip_cmcreate_f(m, p, VSIP_ROW, VSIP_MEM_NONE);

// Initialize matrices A and B with complex values
for (vsip_index i = 0; i < m; i++) {
    for (vsip_index j = 0; j < n; j++) {
        vsip_cscalar_f val = VSIP_CMPLX_F(i*n + j + 1, -(i*n + j + 1));
        vsip_cput_f(A, i, j, val);
    }
}

for (vsip_index i = 0; i < n; i++) {
    for (vsip_index j = 0; j < p; j++) {
        vsip_cscalar_f val = VSIP_CMPLX_F(i*p + j + 1, i*p + j + 2);
        vsip_cput_f(B, i, j, val);
    }
}

// Compute matrix product with conjugate: R = A * conj(B)
vsip_cmprodj_f(A, B, R);

// Print the result
printf("Result matrix R = A * conj(B) (%lux%lu):\n", m, p);
for (vsip_index i = 0; i < m; i++) {
    for (vsip_index j = 0; j < p; j++) {
        vsip_cscalar_f val = vsip_cmget_f(R, i, j);
        printf("(%.2f%+.2fi) ", val.r, val.i);
    }
    printf("\n");
}
```

```
// Clean up  
vsip_cmalldestroy_f(A);  
vsip_cmalldestroy_f(B);  
vsip_cmalldestroy_f(R);
```

Notes

- The input matrices must have compatible dimensions: A must be $m \times n$ and B must be $n \times p$.
- The output matrix R must have dimensions $m \times p$.
- This operation is different from `vsip_cmprodh_p` which uses the Hermitian transpose of the second matrix.
- The element-wise conjugation of B affects only the imaginary parts of its elements, changing their sign.

6.2 Special Linear Solvers

6.2.1 vsip_dtoepsol_p - Solve a Toeplitz System of Equations

```
int vsip_toepsol_f(const vsip_vview_f* t, const vsip_vview_f* r, const vsip_vview_f* b, const vsip_vview_f* x)
int vsip_toepsol_d(const vsip_vview_d* t, const vsip_vview_d* r, const vsip_vview_d* b, const vsip_vview_d* x)
int vsip_ctoepsol_f(const vsip_cvview_f* t, const vsip_cvview_f* r, const vsip_cvview_f* b, const vsip_cvview_f* x)
int vsip_ctoepsol_d(const vsip_cvview_d* t, const vsip_cvview_d* r, const vsip_cvview_d* b, const vsip_cvview_d* x)
```

Description

This function solves a real Toeplitz system of linear equations $\mathbf{T}\mathbf{x} = \mathbf{b}$, where $\mathbf{T}$ is a symmetric Toeplitz matrix defined by its first column $\mathbf{t}$ and first row $\mathbf{r}$, $\mathbf{b}$ is the right-hand side vector, and $\mathbf{x}$ is the solution vector. The Toeplitz matrix has constant diagonals, with the first column $\mathbf{t}$ and first row $\mathbf{r}$ defining the matrix structure.

Parameters

- `const vsip_dvview_p* t`: Pointer to the vector view containing the first column of the Toeplitz matrix.
- `const vsip_dvview_p* r`: Pointer to the vector view containing the first row of the Toeplitz matrix.
- `const vsip_dvview_p* b`: Pointer to the vector view containing the right-hand side vector.
- `const vsip_dvview_p* x`: Pointer to the vector view where the solution will be stored.

Return Value

- Returns 0 on success.
- Returns a non-zero value on error (e.g., if the Toeplitz matrix is singular).

Example

```
vsip_vview_f *toeplitz_col;
vsip_vview_f *toeplitz_row;
vsip_vview_f *rhs_vector;
vsip_vview_f *solution_vector;
int result;

// Assuming all vector views have been properly initialized
result = vsip_toepsol_f(toeplitz_col, toeplitz_row, rhs_vector, solution_vector);

if (result != 0) {
    // Handle error (e.g., singular matrix)
}
```

6.2.2 vsip_dcovsol_p - Solve a Covariance System of Equations

```
int vsip_covsol_f(const vsip_vview_f* r, const vsip_vview_f* b, const vsip_vview_f* x);
int vsip_covsol_d(const vsip_vview_d* r, const vsip_vview_d* b, const vsip_vview_d* x);
int vsip_ccovsol_f(const vsip_cvview_f* r, const vsip_cvview_f* b, const vsip_cvview_f* x);
int vsip_ccovsol_d(const vsip_cvview_d* r, const vsip_cvview_d* b, const vsip_cvview_d* x);
```

Description

This function solves a covariance system of linear equations $\mathbf{T}\mathbf{x} = \mathbf{b}$, where $\mathbf{T}$ is a symmetric positive definite Toeplitz covariance matrix defined by its first column $\mathbf{r}$, $\mathbf{b}$ is the right-hand side vector, and $\mathbf{x}$ is the solution vector. The covariance matrix is a special type of Toeplitz matrix where the first column $\mathbf{r}$ completely defines the matrix structure.

This function is particularly useful in signal processing applications such as linear prediction and Wiener filtering, where covariance matrices frequently appear.

Parameters

- `const vsip_dvview_p* r`: Pointer to the vector view containing the first column of the covariance matrix (auto-correlation sequence). The length of this vector determines the size of the covariance matrix.
- `const vsip_dvview_p* b`: Pointer to the vector view containing the right-hand side vector.
- `const vsip_dvview_p* x`: Pointer to the vector view where the solution will be stored.

Return Value

- Returns 0 on success.
- Returns a non-zero value on error (e.g., if the covariance matrix is singular or not positive definite).

Example

```
vsip_vview_f *covariance_vector;
vsip_vview_f *rhs_vector;
vsip_vview_f *solution_vector;
int result;

// Assuming all vector views have been properly initialized
// and covariance_vector contains the autocorrelation sequence
result = vsip_covsol_f(covariance_vector, rhs_vector, solution_vector);

if (result != 0) {
    // Handle error (e.g., singular or non-positive definite matrix)
}
```

Notes

- The covariance matrix is assumed to be symmetric and positive definite.
- The length of the covariance vector $\mathbf{r}$ should be one more than the length of vectors $\mathbf{b}$ and $\mathbf{x}$.
- This function uses a Levinson-Durbin recursion algorithm for efficient solution of the covariance system.

6.2.3 vsip_dllsqsol_p - Solve Linear Least Squares Problem

```
int vsip_llsqsol_f(const vsip_mview_f* A, const vsip_vview_f* b, const vsip_vview_f* x);
int vsip_llsqsol_d(const vsip_mview_d* A, const vsip_vview_d* b, const vsip_vview_d* x);
int vsip_cllsqsol_f(const vsip_cmview_f* A, const vsip_cvview_f* b, const vsip_cvview_f* x);
int vsip_cllsqsol_d(const vsip_cmview_d* A, const vsip_cvview_d* b, const vsip_cvview_d* x);
```

Description

This function solves the linear least squares problem:

$$\min_x \|Ax - b\|_2$$

where A is an $M \times N$ matrix with $M \geq N$, b is an M -dimensional vector, and x is the N -dimensional solution vector that minimizes the Euclidean norm of the residual vector.

The function uses QR decomposition with column pivoting to solve the least squares problem, which provides a numerically stable solution even when matrix A is rank-deficient.

Parameters

- `const vsip_dmview_p* A`: Pointer to the $M \times N$ matrix view of coefficients.
- `const vsip_dvview_p* b`: Pointer to the M -dimensional vector view containing the right-hand side.
- `const vsip_dvview_p* x`: Pointer to the N -dimensional vector view where the least squares solution will be stored.

Return Value

- Returns 0 on success.
- Returns a non-zero value on error (e.g., if matrix dimensions are incompatible or memory allocation fails).

Example

```
vsip_mview_f *A;           // MxN coefficient matrix
vsip_vview_f *b;           // M-dimensional right-hand side vector
vsip_vview_f *x;           // N-dimensional solution vector
int result;

// Assuming A, b, and x have been properly initialized with appropriate dimensions
result = vsip_llsqsol_f(A, b, x);

if (result != 0) {
    // Handle error
}
```

Notes

- The number of rows M in matrix A must be greater than or equal to the number of columns N .
- The solution x minimizes the 2-norm of the residual vector $Ax - b$.
- If A has full column rank, the solution is unique. If A is rank-deficient, the function returns a basic solution with at most $\text{rank}(A)$ non-zero components.
- This function is particularly useful for overdetermined systems where there is no exact solution, but a best-fit solution is desired.

6.3 General Linear Square System Solver

6.3.1 vsip_dlud_create_p - Create LU Decomposition Object

```
vsip_lu_f* vsip_lud_create_f(vsip_length n);  
vsip_lu_d* vsip_lud_create_d(vsip_length n);  
vsip_clu_f* vsip_clud_create_f(vsip_length n);  
vsip_clu_d* vsip_clud_create_d(vsip_length n);
```

Description

This function creates an LU decomposition object for factoring an $n \times n$ matrix into the product of a lower triangular matrix L and an upper triangular matrix U . The object can be reused for multiple decompositions of matrices with the same dimensions.

Parameters

- vsip_length n: Number of rows and columns in the matrix to be decomposed.

Return Value

- On success, a pointer to the newly created LU decomposition object is returned.
- On error, NULL is returned.

Example

```
vsip_lu_f *lu_obj;  
vsip_length n = 100;  
  
// Create LU decomposition object  
lu_obj = vsip_lud_create_f(n);  
  
if (lu_obj == NULL) {  
    // Handle error  
}
```

6.3.2 vsip_d lud_destroy_p - Destroy LU Decomposition Object

```
int vsip_lud_destroy_f(vsip_lu_f *lu);
int vsip_lud_destroy_d(vsip_lu_d *lu);
int vsip_clud_destroy_f(vsip_clu_f *lu);
int vsip_clud_destroy_d(vsip_clu_d *lu);
```

Description

This function destroys an LU decomposition object and frees all associated resources.

Parameters

- `vsip_d lu_f * lu`: Pointer to the LU decomposition object to be destroyed.

Return Value

- Returns 0 on success.
- Returns a non-zero value on error.

Example

```
vsip_lu_f *lu_obj;
int result;

// Assuming lu_obj has been properly initialized
result = vsip_lud_destroy_f(lu_obj);

if (result != 0) {
    // Handle error
}
```

6.3.3 vsip_d lud_getattr_p - Get LU Decomposition Attributes

```
typedef struct _vsip_lu_attr_g {
    vsip_length n;
} vsip_lu_attr_g;

typedef vsip_lu_attr_g vsip_lu_attr_f;
typedef vsip_lu_attr_g vsip_clu_attr_f;

void vsip_lud_getattr_f(const vsip_lu_f *lu, vsip_lu_attr_f *attr);
void vsip_lud_getattr_d(const vsip_lu_d *lu, vsip_lu_attr_d *attr);
void vsip_clud_getattr_f(const vsip_clu_f *lu, vsip_lu_cattr_f *attr);
void vsip_clud_getattr_d(const vsip_clu_d *lu, vsip_lu_cattr_d *attr);
```

Description

This function retrieves the attributes of an LU decomposition object, this currently includes a single attribute called `n` for the row and column element count of the square matrix.

Parameters

- `const vsip_d lu_p* lu`: Pointer to the LU decomposition object.
- `vsip_lu_d attr_p* attr`: Pointer to a structure where the attributes will be stored.

Example

```
vsip_lu_f *lu_obj;
vsip_lu_attr_f attr;

// Assuming lu_obj has been properly initialized
vsip_lud_getattr_f(lu_obj, &attr);

// attr.n - Number of rows and columns of the square matrix
```

6.3.4 vsip_d lud_p - Perform LU Decomposition

```
int vsip_lud_f(const vsip_lu_f* lud, const vsip_mview_f* A);
int vsip_lud_d(const vsip_lu_d* lud, const vsip_mview_d* A);
int vsip_clud_f(const vsip_clu_f* lud, const vsip_cmview_f* A);
int vsip_clud_d(const vsip_clu_d* lud, const vsip_cmview_d* A);
```

Description

This function performs LU decomposition of matrix A using the pre-allocated LU decomposition object lud . The decomposition computes:

$$A = PLU$$

where:

- P is a permutation matrix
- L is a unit lower triangular matrix
- U is an upper triangular matrix

The function uses partial pivoting for numerical stability. The decomposed factors are stored within the LU decomposition object and can be used for subsequent operations like solving linear systems.

Parameters

- `const vsip_d lu_p* lud`: Pointer to the LU decomposition object created by `vsip_lud_create_p`.
- `const vsip_d mview_p* A`: Pointer to the $n \times n$ matrix view to be decomposed.

Return Value

- Returns 0 on success.
- Returns a non-zero value on error (e.g., if the matrix is singular or dimensions don't match the LU object).

Example

```
vsip_lu_f *lu_obj;
vsip_mview_f *matrix_A;
int result;

// Assuming lu_obj and matrix_A have been properly initialized
// with matching dimensions
result = vsip_lud_f(lu_obj, matrix_A);

if (result != 0) {
    // Handle error (e.g., singular matrix)
}
```

Notes

- The input matrix A must have full rank.
- The input matrix A must have the same dimensions as specified when creating the LU decomposition object.
- The contents of matrix A may be overwritten and must not be modified as long as factorization is required.
- The decomposed factors L and U are stored within the LU decomposition object and can be accessed through other VSIPL functions.

6.3.5 vsip_dlusol_p - Solve Linear System Using LU Decomposition

```
int vsip_lusol_f(const vsip_lu_f* lud, const vsip_vview_f* b, const vsip_vview_f* x);
int vsip_lusol_d(const vsip_lu_d* lud, const vsip_vview_d* b, const vsip_vview_d* x);
int vsip_clusol_f(const vsip_clu_f* lud, const vsip_cvview_f* b, const vsip_cvview_f* x);
int vsip_clusol_d(const vsip_clu_d* lud, const vsip_cvview_d* b, const vsip_cvview_d* x);
```

Description

This function solves a system a linear square system int the forms of:

$$\mathbf{AX} = \mathbf{B}$$

$$\mathbf{A}^T \mathbf{X} = \mathbf{B}$$

$$\mathbf{A}^H \mathbf{X} = \mathbf{B}$$

Where the matrix $\mathbf{A}$ has previously been decomposed using the function `vsip_d lud_p`. Whether the matrix $\mathbf{A}$ is transposed depends on the given argument provided.

Parameters

- `const vsip_d lu_p* lud`: Pointer to the LU decomposition object containing the decomposed factors of matrix $\mathbf{A}$.
- `vsip_mat_op OpA`: Operand for the input matrix $\mathbf{A}$.
 - `VSIP_MAT_NTRANS` - Do not transpose.
 - `VSIP_MAT_TRANS` - Transpose.
 - `VSIP_MAT_HERM` - Hermitian (Complex only).
- `const vsip_dmview_p* xb`: Pointer to the right-hand side matrix $\mathbf{B}$ of order n by k . On exit result matrix $\mathbf{X}$.

Return Value

- Returns 0 on success.
- Returns a non-zero value on error (e.g., if dimensions are incompatible or the matrix is singular).

Example

```
vsip_lu_f *lu_obj;
vsip_mview_f *a, *xb;

// Assuming all objects have been properly initialized
// First perform LU decomposition
result = vsip_lud_f(lu_obj, a);
if (result != 0) {
    // Handle decomposition error
}

// Then solve the linear system
result = vsip_lusol_f(lu_obj, VSIP_MAT_NTRANS, xb);
if (result != 0) {
    // Handle solve error
}
```

Notes

- The LU decomposition object must have been previously created and used to decompose a matrix.

6.4 Symmetric Positive Definite Linear System Solver

6.4.1 vsip_dchold_create_p - Create Cholesky Decomposition Object

```
typedef enum _vsip_mat_uplo {
    VSIP_TR_LOW = 0, // Lower triangular
    VSIP_TR_UPP = 1  // Upper triangular
} vsip_mat_uplo;

vsip_chol_f* vsip_chold_create_f(vsip_mat_uplo uplo, vsip_length n);
vsip_chol_d* vsip_chold_create_d(vsip_mat_uplo uplo, vsip_length n);
vsip_cchol_f* vsip_cchold_create_f(vsip_mat_uplo uplo, vsip_length n);
vsip_cchol_d* vsip_cchold_create_d(vsip_mat_uplo uplo, vsip_length n);
```

Description

This function creates a Cholesky decomposition object for a symmetric positive definite matrix of size $n \times n$. The Cholesky decomposition expresses a matrix A as the product of a lower triangular matrix L and its transpose: $A = LL^T$ (when `uplo = VSIP_MAT_LOWER`) or $A = U^T U$ (when `uplo = VSIP_MAT_UPPER`).

Parameters

- `vsip_mat_uplo uplo`: Specifies whether to store the upper (`VSIP_MAT_UPP`) or lower (`VSIP_MAT_LOW`) triangle of the matrix.
- `vsip_length n`: The dimension of the square matrix ($n \times n$).

Return Value

- On success: Pointer to the newly created Cholesky decomposition object
- On error (e.g., memory allocation failure): `NULL`

Example

```
vsip_chol_f *chold;
vsip_length n = 100;

// Create Cholesky decomposition object for lower triangle
chold = vsip_chold_create_f(VSIP_MAT_LOW, n);
if (chold == NULL) {
    fprintf(stderr, "Error: Could not create Cholesky object\n");
    return -1;
}
```

Notes

- The matrix must be symmetric and positive definite, otherwise the decomposition will fail.
- The object should be freed with `vsip_dchold_destroy_p` when no longer needed.

6.4.2 vsip_dchold_destroy_p - Destroy Cholesky Decomposition Object

```
int vsip_chold_destroy_f(vsip_chol_f *chold);  
int vsip_chold_destroy_d(vsip_chol_d *chold);  
int vsip_cchold_destroy_f(vsip_cchol_f *chold);  
int vsip_cchold_destroy_d(vsip_cchol_d *chold);
```

Description

This function releases the memory allocated for a Cholesky decomposition object.

Parameters

- `vsip_dchol_p* chold`: Pointer to the Cholesky decomposition object to be destroyed.

Return Value

- Returns 0

6.4.3 vsip_dchold_getattr_p - Get Cholesky Decomposition Attributes

```
typedef struct _vsip_chol_attr_f {
    vsip_length n;
    vsip_mat_uplo uplo;
} vsip_chol_attr_f;

typedef vsip_chol_attr_f vsip_cchol_attr_f;

void vsip_chold_getattr_f(const vsip_chol_f *chold, vsip_chol_attr_f *attr);
void vsip_chold_getattr_d(const vsip_chol_d *chold, vsip_chol_attr_d *attr);
void vsip_cchold_getattr_f(const vsip_cchol_f *chold, vsip_cchol_attr_f *attr);
void vsip_cchold_getattr_d(const vsip_cchol_d *chold, vsip_cchol_attr_d *attr);
```

Description

This function retrieves the attributes of a Cholesky decomposition object and stores them in the provided structure.

Parameters

- `const vsip_dchol_p* chold`: Pointer to the Cholesky decomposition object.
- `vsip_dchol_attr_p* attr`: Pointer to the structure where attributes will be stored.

Example

```
vsip_chol_f *chold;
vsip_chol_attr_f attr;

// Create object
chold = vsip_chold_create_f(VSIP_MAT_LOW, 100);

// Get attributes
vsip_chold_getattr_f(chold, &attr);

printf("Cholesky decomposition attributes:\n");
printf("  Matrix size: %lu x %lu\n", attr.n, attr.n);
printf("  Stored triangle: %s\n",
       attr.uplo == VSIP_MAT_LOW ? "lower" : "upper");

// Destroy object
vsip_chold_destroy_f(chold);
```

6.4.4 vsip_dchold_p - Perform Cholesky Decomposition

```

int vsip_chold_f(vsip_chol_f *chold, const vsip_mview_f *a);
int vsip_chold_d(vsip_chol_d *chold, const vsip_mview_d *a);
int vsip_cchold_f(vsip_cchol_f *chold, const vsip_cmview_f *a);
int vsip_cchold_d(vsip_cchol_d *chold, const vsip_cmview_d *a);

```

Description

This function performs the Cholesky decomposition of a symmetric positive definite matrix A using the provided Cholesky decomposition object. The decomposition expresses A as the product of a triangular matrix and its transpose:

When `uplo = VSIP_MAT_LOW`:

$$A = LL^T$$

$$A = LL^H$$

When `uplo = VSIP_MAT_UPP`:

$$A = U^T U$$

$$A = U^H U$$

Where L is a lower triangular matrix and U is an upper triangular matrix.

Parameters

- `vsip_dchol_p* chold`: Pointer to the Cholesky decomposition object created with `vsip_dchold_create_p`.
- `const vsip_dmview_p* a`: Pointer to the input matrix to be decomposed. The matrix must be symmetric positive definite and have dimensions matching those specified when the Cholesky object was created.

Return Value

- Returns 0 on success.
- Returns a non-zero value on error (e.g., if the matrix is not positive definite or dimensions don't match).

Notes

- The input matrix A must be symmetric and positive definite. The function will fail if the matrix is not positive definite.
- The matrix dimensions must match those specified when the Cholesky object was created.
- The decomposition overwrites the contents of the Cholesky object with the new decomposition.
- The Cholesky object can be reused for multiple decompositions by calling this function multiple times with different input matrices (as long as they have the same dimensions).

6.4.5 vsip_dcholsol_p - Solve Linear Systems Using Cholesky Decomposition

```
int vsip_cholsol_f(const vsip_chol_f *chold, const vsip_mview_f *a);
int vsip_cholsol_d(const vsip_chol_d *chold, const vsip_mview_d *a);
int vsip_ccholsol_f(const vsip_cchol_f *chold, const vsip_cmview_f *a);
int vsip_ccholsol_d(const vsip_cchol_d *chold, const vsip_cmview_d *a);
```

Description

This function solves linear systems of equations using a previously computed Cholesky decomposition. It solves the system $AX = B$ where A is a symmetric positive definite matrix that has been decomposed using `vsip_dchold_p`.

The function uses the Cholesky decomposition $A = LL^T$ (or $A = U^T U$, $A = U^H U$) to efficiently solve the linear system by performing forward and backward substitution on the triangular factors.

Parameters

- `const vsip_dchol_p* chold`: Pointer to the Cholesky decomposition object containing a previously computed decomposition.
- `const vsip_dmview_p* a`: On input, contains the right-hand side matrix B . On output, contains the solution matrix X . The matrix should have dimensions $n \times k$ where A is $n \times n$ and k is the number of right-hand sides.

Return Value

- Returns 0 on success.
- Returns a non-zero value on error

Notes

- The Cholesky decomposition must have been previously computed using `vsip_dchold_p`.

6.5 Over-determined Linear System Solver

6.5.1 vsip_dqrd_create_p - Create QR Decomposition Object

```
typedef enum _vsip_qrd_qopt {
    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;

vsip_qr_f* vsip_qrd_create_f(vsip_length m, vsip_length n, vsip_qrd_qopt qopt);
vsip_qr_d* vsip_qrd_create_d(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);
vsip_cqr_d* vsip_cqrd_create_d(vsip_length m, vsip_length n, vsip_qrd_qopt qopt);
```

Description

This function creates a QR decomposition object that can be used to compute the QR factorization of an $m \times n$ matrix. The QR decomposition expresses a matrix A as the product of an orthogonal matrix Q and an upper triangular matrix R , such that $A = QR$.

The `vsip_qrd_qopt` parameter allows you to specify how the orthogonal matrix Q should be stored.

Parameters

- `vsip_length m`: Number of rows in the matrix to be decomposed.
- `vsip_length n`: Number of columns in the matrix to be decomposed.
- `vsip_qrd_qopt qopt`: Option specifying how the Q matrix should be saved:
 - `VSIP_QRD_NOSAVEQ`: Don't save Q (only compute R)
 - `VSIP_QRD_SAVEQ1`: Save essential parts of Q (more memory efficient)
 - `VSIP_QRD_SAVEQ`: Save full Q matrix

Return Value

- On success, returns a pointer to the newly created QR decomposition object.
- On error (e.g., if memory allocation fails), returns `NULL`.

Example

```
vsip_qr_f *qrd;
vsip_length m = 100, n = 50;

// Create a QR decomposition object
// Using SAVEQ2 as a good compromise between memory and functionality
qrd = vsip_qrd_create_f(m, n, VSIP_QRD_SAVEQ2);

if (qrd == NULL) {
    fprintf(stderr, "Failed to create QR decomposition object\n");
    return;
}
```

Notes

- The QR decomposition object must be destroyed with `vsip_dqrd_destroy_p` when no longer needed.
- The choice of `qopt` affects both memory usage and the operations that can be performed with the decomposition:
 - `VSIP_QRD_SAVEQ` allows full access to Q but uses more memory
 - `VSIP_QRD_SAVEQ1` is a good compromise for most applications
 - `VSIP_QRD_NOSAVEQ` is most memory efficient but only allows operations with R

- For square matrices ($m = n$), the QR decomposition can be used to compute determinants and inverses.
- For tall matrices ($m > n$), the decomposition is useful for least squares problems.
- This function allocates internal storage for the decomposition, which is freed when the QR object is destroyed.
- For repeated decompositions of matrices with the same dimensions, you can reuse the QR object by calling `vsip_dqrd_p` multiple times with the same object.

6.5.2 vsip_dqrd_destroy_p - Destroy QR Decomposition Object

```
int vsip_qrd_destroy_f(vsip_qr_f *qrd);
int vsip_qrd_destroy_d(vsip_qr_d *qrd);
int vsip_cqrd_destroy_f(vsip_cqr_f *qrd);
int vsip_cqrd_destroy_d(vsip_cqr_d *qrd);
```

Description

This function releases the memory allocated for a QR decomposition object and destroys it.

Parameters

- `vsip_dqr_p* qrd`: Pointer to the QR decomposition object to be destroyed, which was previously created with `vsip_dqrd_create_p`.

Return Value

- Returns 0.

Example

```
vsip_qr_f *qrd;
vsip_mview_f *A;
vsip_length m = 100, n = 50;

// Create QR decomposition object
qrd = vsip_qrd_create_f(m, n, VSIP_QRD_SAVEQ);
if (qrd == NULL) {
    fprintf(stderr, "Error: Could not create QR object\n");
    return -1;
}

status = vsip_qrd_destroy_f(qrd);
```

6.5.3 vsip_dqrd_getattr_p - Get QR Decomposition Attributes

```
typedef struct _vsip_qr_attr_f {
    vsip_length n;
    vsip_length m;
    vsip_qrd_qopt Qopt;
} vsip_qr_attr_f;

typedef vsip_qr_attr_f vsip_cqr_attr_f;

void vsip_qrd_getattr_f(const vsip_qr_f *qrd, vsip_qr_attr_f *attr);
void vsip_qrd_getattr_d(const vsip_qr_d *qrd, vsip_qr_attr_d *attr);
void vsip_cqrd_getattr_f(const vsip_cqr_f *qrd, vsip_cqr_attr_f *attr);
void vsip_cqrd_getattr_d(const vsip_cqr_d *qrd, vsip_cqr_attr_d *attr);
```

Description

This function retrieves the attributes of a QR decomposition object and stores them in a `vsip_dqr_attr_p` structure. The attributes provide complete information about the QR decomposition, including the dimensions of the original matrix and the options used during creation.

Parameters

- `const vsip_dqr_p* qrd`: Pointer to the QR decomposition object.
- `vsip_dqr_attr_p* attr`: Pointer to the attribute structure where the QR decomposition attributes will be stored.

Example

```
vsip_qr_f *qrd;
vsip_qr_attr_f attr;
vsip_length m = 100, n = 50;

// Create a QR decomposition object
qrd = vsip_qrd_create_f(m, n, VSIP_QRD_SAVEQ);
if (qrd == NULL) {
    // Handle error
}

// Get the attributes of the QR decomposition object
vsip_qrd_getattr_f(qrd, &attr);

printf("QR decomposition attributes:\n");
printf("  Matrix dimensions: %lu x %lu\n", attr.m, attr.n);
printf("  QR option: %d\n", attr.qopt);

vsip_qrd_destroy_f(qrd);
```


6.5.4 vsip_dqrd_p - Perform QR Decomposition

```
int vsip_qrd_f(vsip_qr_f *qrd, const vsip_mview_f *a);
int vsip_qrd_d(vsip_qr_d *qrd, const vsip_mview_d *a);
int vsip_cqrd_f(vsip_cqr_f *qrd, const vsip_cmview_f *a);
int vsip_cqrd_d(vsip_cqr_d *qrd, const vsip_cmview_d *a);
```

Description

This function performs the QR decomposition of matrix A using the provided QR decomposition object. The QR decomposition expresses matrix A as the product of an orthogonal matrix Q and an upper triangular matrix R , such that $A = QR$.

Parameters

- `vsip_dqr_p * qrd`: Pointer to the QR decomposition object created with `vsip_dqrd_create_p`.
- `const vsip_dmview_p * a`: Pointer to the input matrix to be decomposed. The matrix must have dimensions matching those specified when the QR object was created. May be overwritten by the decomposition.

Return Value

- Returns 0 on success.
- Returns a non-zero value on error (e.g., if the matrix dimensions don't match the QR object).

Notes

- The input matrix A must have dimensions $m \times n$ that match those specified when the QR object was created.
- The decomposition overwrites the contents of the QR object with the new decomposition.
- The QR object can be reused for multiple decompositions by calling this function multiple times with different input matrices (as long as they have the same dimensions).

6.5.5 vsip_dqrsol_p - Solve Linear Systems Using QR Decomposition

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

int vsip_qrsol_f(const vsip_qr_f *qrd, vsip_qrd_prob prob, const vsip_mview_f *xb);
int vsip_qrsol_d(const vsip_qr_d *qrd, vsip_qrd_prob prob, const vsip_mview_d *xb);
int vsip_cqrsol_f(const vsip_cqr_f *qrd, vsip_qrd_prob prob, const vsip_cmview_f *xb);
int vsip_cqrsol_d(const vsip_cqr_d *qrd, vsip_qrd_prob prob, const vsip_cmview_d *xb);
```

Description

This function solves linear systems of equations using a previously computed QR decomposition for a matrix A with $m \times n$ with rank n . It can solve a covariance linear system problem

$$A^T A X = B$$

for real or

$$A^H A X = B$$

for complex. Or a linear least squares problem,

$$\min \|AX - B\|_2$$

Parameters

- `const vsip_dqr_p* qrd`: Pointer to the QR decomposition object containing a previously computed decomposition.
- `vsip_qrd_prob prob`: Type of problem to solve.
- `const vsip_dmview_p* xb`: On input, contains the right-hand side matrix B of size $n \times k$ for a covariance problem and $m \times k$ for a least squares problem. On output, contains the solution X .

Return Value

- Returns 0 on success.
- Returns a non-zero value on error

Notes

- The QR decomposition must have been previously computed using `vsip_dqrd_p`.
- The QR object must have been created with an option that saves the Q matrix to use this function.

6.5.6 vsip_dqrdslr_p - Solve Linear Systems with Modified R Matrix

```
typedef enum _vsip_mat_side {
    VSIP_MAT_LSIDE = 0,
    VSIP_MAT_RSIDE = 1
} vsip_mat_side;

int vsip_qrdslr_f(const vsip_qr_f *qrd, vsip_mat_op OpR, vsip_scalar_f alpha, const vsip_mview_f *xb);
int vsip_qrdslr_d(const vsip_qr_d *qrd, vsip_mat_op OpR, vsip_scalar_d alpha, const vsip_mview_d *xb);
int vsip_cqrdslr_f(const vsip_cqr_f *qrd, vsip_mat_op OpR, vsip_cscalar_f alpha, const vsip_cmview_f *xb);
int vsip_cqrdslr_d(const vsip_cqr_d *qrd, vsip_mat_op OpR, vsip_cscalar_d alpha, const vsip_cmview_d *xb);
```

Description

This function solves linear systems using a QR decomposition where the R matrix has been modified by a specified operation. It provides more flexibility than `vsip_dqrsol_p` by allowing operations on the R matrix before solving the system.

The function solves systems of the form:

$$\text{op}(R)X = \alpha B$$

where $\text{op}(R)$ can be R , R^T , or R^H (conjugate transpose, though for real matrices this is equivalent to R^T).

Parameters

- `const vsip_dqr_p * qrd`: Pointer to the QR decomposition object containing a previously computed decomposition.
- `vsip_mat_op OpR`: Operation to perform on R :
 - `VSIP_MAT_NTRANS`: Use R as is
 - `VSIP_MAT_TRANS`: Use the transpose of R , R^T
 - `VSIP_MAT_HERM`: Use the conjugate transpose of R , R^H
- `vsip_dscalar_p alpha`: Scalar multiplier for the right-hand side.
- `const vsip_dmview_p * xb`: On input, contains the right-hand side B . On output, contains the solution X .

Return Value

- Returns 0 on success.
- Returns a non-zero value on error.

Notes

- The QR decomposition must have been previously computed using `vsip_dqrd_p`.
- The input matrix B must have appropriate dimensions for the operation:
 - For `VSIP_MAT_NTRANS`: B should be $n \times k$ where A is $m \times n$
 - For `VSIP_MAT_TRANS` or `VSIP_MAT_HERM`: B should be $m \times k$ where A is $m \times n$
- The input matrix XB is overwritten with the solution.
- The scalar α allows scaling of the right-hand side without modifying the input matrix.
- This function is more flexible than `vsip_dqrsol_p` but requires more understanding of the underlying linear algebra.
- The QR object must have been created with an option that saves the Q matrix to use this function.

6.5.7 vsip_dqrdprodq_p - Multiply by Q Matrix from QR Decomposition

```
typedef enum _vsip_mat_op {
    VSIP_MAT_NTRANS = 0, // op(A) = A
    VSIP_MAT_TRANS = 1, // op(A) = A^T
    VSIP_MAT_HERM = 2, // op(A) = A^H (complex only)
    VSIP_MAT_CONJ = 3 // op(X) = A^* (complex only)
} vsip_mat_op;

typedef enum _vsip_mat_side {
    VSIP_MAT_LSIDE = 0,
    VSIP_MAT_RSIDE = 1
} vsip_mat_side;

int vsip_qrdprodq_f(const vsip_qr_f *qrd, vsip_mat_op opQ, vsip_mat_side apQ, const vsip_mview_f *c);
int vsip_qrdprodq_d(const vsip_qr_d *qrd, vsip_mat_op opQ, vsip_mat_side apQ, const vsip_mview_d *c);
int vsip_cqrdprodq_f(const vsip_cqr_f *qrd, vsip_mat_op opQ, vsip_mat_side apQ, const vsip_cmview_f *c);
int vsip_cqrdprodq_d(const vsip_cqr_d *qrd, vsip_mat_op opQ, vsip_mat_side apQ, const vsip_cmview_d *c);
```

Description

This function performs matrix multiplication with the orthogonal matrix Q from a QR decomposition. It computes either QC , Q^TC , Q^HC , CQ , CQ^T or CQ^H , depending on the specified parameters.

The operation performed is determined by the `opQ` and `apQ` parameters:

- `opQ` specifies whether to use Q , Q^T or Q^H
- `apQ` specifies whether Q is on the left or right of the multiplication

Parameters

- `const vsip_dqr_p* qrd`: Pointer to the QR decomposition object containing a previously computed decomposition.
- `vsip_mat_op opQ`: Operation to perform with Q :
 - `VSIP_MAT_NTRANS`: Use Q as is
 - `VSIP_MAT_TRANS`: Use the transpose of Q , Q^T
 - `VSIP_MAT_HERM`: Use the conjugate transpose of Q , Q^H
- `vsip_mat_side apQ`: Side of multiplication:
 - `VSIP_MAT_LEFT`: Q is on the left (QC , Q^TC or Q^HC)
 - `VSIP_MAT_RIGHT`: Q is on the right (CQ , CQ^T or CQ^H)
- `const vsip_dmview_p* c`: On input, contains matrix C . On output, contains the result of the multiplication.

Return Value

- Returns 0 on success.
- Returns a non-zero value on error.

Notes

- The QR decomposition must have been previously computed using `vsip_dqrd_p`.
- The QR object must have been created with an option that saves the Q matrix (`VSIP_QRD_SAVEQ` or `VSIP_QRD_SAVEQ1`).

- The input matrix C must have appropriate dimensions for the operation.

For VSIP_QRD_SAVEQ1:		Input		Output	
		MAT_LSIDE	MAT_RSIDE	MAT_LSIDE	MAT_RSIDE
	MAT_NTRANS	$n \times s$	$r \times m$	$m \times s$	$r \times n$
	MAT_TRANS	$m \times s$	$r \times n$	$n \times s$	$r \times m$
	MAT_HERM	$m \times s$	$r \times n$	$n \times s$	$r \times m$

For VSIP_QRD_SAVEQ:		Input and Output	
		MAT_LSIDE	MAT_RSIDE
	MAT_NTRANS	$m \times s$	$r \times m$
	MAT_TRANS	$m \times s$	$r \times m$
	MAT_HERM	$m \times s$	$r \times m$