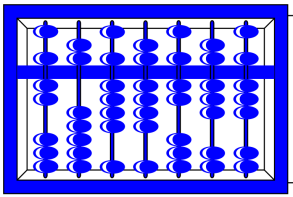

NASoftware Limited

Incorporating InfoSAR

CSIPL
Quick Reference Guide

CSIPL/Brief [3.1]

Release 3.1
May 2013

Contents

1 Introduction **1**

1.1 Types . . . 1
1.2 Symbols and Flags . . . 1
1.3 Vector Variables . . . 1
1.4 Matrix Variables . . . 2
1.5 Complex Variables . . . 2
1.6 Function Names . . . 3
1.7 Hints . . . 3
1.8 Notation . . . 3
1.9 Errors and Restrictions . . . 4

2 Getting the Best Performance **5**

2.1 Version Information . . . 5
2.2 Memory Alignment . . . 5
2.3 Vector/Matrix Format . . . 5
2.4 Error Checking and Debugging . . . 6
2.5 Support Functions . . . 7
2.6 Scalar Functions . . . 7
2.7 Random Number Generation . . . 7
2.8 Vector and Elementwise Operations . . . 7
2.9 Signal Processing Functions . . . 7
2.10 FFT Functions . . . 8
2.11 FIR Filter, Convolution and Correlation Functions . . . 9
2.12 Linear Algebra Functions . . . 9
2.13 Matrix and Vector Operations . . . 9
2.14 LU Decomposition, Cholesky and QRD Functions . . . 9
2.15 Special Linear System Solvers . . . 10
2.16 Controlling the Number of Threads . . . 10

3 Support Functions **11**

3.1 Initilialisation and Finalisation . . . 11
csipl_init . . . 11
csipl_finalize . . . 11
3.2 Sundry Functions . . . 11
csipl_complete . . . 11

csipl_cstorage 11

4 Scalar Functions **12**

4.1 Real Scalar Functions 12

csipl_acos_f 12
csipl_asin_f 12
csipl_atan_f 12
csipl_atan2_f 12
csipl_ceil_f 12
csipl_cos_f 12
csipl_cosh_f 12
csipl_exp_f 12
csipl_floor_f 12
csipl_log_f 12
csipl_log10_f 12
csipl_mag_f 12
csipl_pow_f 12
csipl_sin_f 12
csipl_sinh_f 12
csipl_sqrt_f 12
csipl_tan_f 12
csipl_tanh_f 13

4.2 Complex Scalar Functions 13

csipl_arg_f 13
csipl_cadd_f 13
csipl_rcadd_f 13
csipl_cdiv_f 13
csipl_crdiv_f 13
csipl_cexp_f 13
csipl_cjmul_f 13
csipl_cmag_f 13
csipl_cmagsq_f 13
csipl_cmplx_f 13
csipl_cmul_f 13
csipl_rcmul_f 13
csipl_cneg_f 13
csipl_conj_f 13
csipl_crecip_f 13

csipl_csqrt_f . . . 14
csipl_csub_f . . . 14
csipl_rcsub_f . . . 14
csipl_crsub_f . . . 14
csipl_imag_f . . . 14
csipl_polar_f . . . 14
csipl_real_f . . . 14
csipl_rect_f . . . 14
4.3 Index Scalar Functions . . . 14
csipl_matindex . . . 14
csipl_mcolindex . . . 14
csipl_mrowindex . . . 14

5 Random Number Generation 15
5.1 Random Number Functions . . . 15
csipl_randcreate . . . 15
csipl_randdestroy . . . 15
csipl_randu_f . . . 15
csipl_crandu_f . . . 15
csipl_vrandu_f . . . 15
csipl_cvrandu_inter_f . . . 15
csipl_cvrandu_split_f . . . 15
csipl_randn_f . . . 15
csipl_crandn_f . . . 16
csipl_vrandn_f . . . 16
csipl_cvrandn_inter_f . . . 16
csipl_cvrandn_split_f . . . 16

6 Vector And Elementwise Operations 17
6.1 Elementary Mathematical Functions . . . 17
csipl_vacos_f . . . 17
csipl_macos_f . . . 17
csipl_vasin_f . . . 17
csipl_masin_f . . . 17
csipl_vatan_f . . . 17
csipl_matan_f . . . 17
csipl_vatan2_f . . . 17
csipl_matan2_f . . . 18
csipl_vcos_f . . . 18

csipl_mcos_f . . . 18
csipl_vcosh_f . . . 18
csipl_mcosh_f . . . 18
csipl_vexp_f . . . 18
csipl_cvexp_inter_f . . . 18
csipl_cvexp_split_f . . . 19
csipl_mexp_f . . . 19
csipl_cmexp_inter_f . . . 19
csipl_cmexp_split_f . . . 19
csipl_vexp10_f . . . 19
csipl_mexp10_f . . . 19
csipl_vfloor_f . . . 19
csipl_vlog_f . . . 20
csipl_cvlog_inter_f . . . 20
csipl_cvlog_split_f . . . 20
csipl_mlog_f . . . 20
csipl_cmlog_inter_f . . . 20
csipl_cmlog_split_f . . . 20
csipl_vlog10_f . . . 20
csipl_mlog10_f . . . 21
csipl_vsin_f . . . 21
csipl_msin_f . . . 21
csipl_vsinh_f . . . 21
csipl_msinh_f . . . 21
csipl_vsqrt_f . . . 21
csipl_cvsqrt_inter_f . . . 21
csipl_cvsqrt_split_f . . . 21
csipl_msqrt_f . . . 22
csipl_cmsqrt_inter_f . . . 22
csipl_cmsqrt_split_f . . . 22
csipl_mtan_f . . . 22
csipl_mtanh_f . . . 22
6.2 Unary Operations . . . 22
csipl_varg_f . . . 22
csipl_marg_f . . . 22
csipl_vceil_f . . . 23
csipl_cvconj_inter_f . . . 23
csipl_cvconj_split_f . . . 23

csipl_cmconj_inter_f . . . 23
csipl_cmconj_split_f . . . 23
csipl_vcumsum_P . . . 23
csipl_cvcumsum_inter_P . . . 23
csipl_cvcumsum_split_P . . . 24
csipl_mcumsum_P . . . 24
csipl_cmcumsum_inter_P . . . 24
csipl_cmcumsum_split_P . . . 24
csipl_veuler_f . . . 24
csipl_meuler_f . . . 24
csipl_vmag_P . . . 24
csipl_cvmag_inter_f . . . 24
csipl_cvmag_split_f . . . 25
csipl_mmag_f . . . 25
csipl_cmmag_inter_f . . . 25
csipl_cmmag_split_f . . . 25
csipl_vcmagsq_inter_f . . . 25
csipl_vcmagsq_split_f . . . 25
csipl_mcmagsq_f . . . 25
csipl_vmeanval_f . . . 25
csipl_cvmeanval_inter_f . . . 26
csipl_cvmeanval_split_f . . . 26
csipl_mmeanval_f . . . 26
csipl_cmmeanval_inter_f . . . 26
csipl_cmmeanval_split_f . . . 26
csipl_vmeansqval_f . . . 26
csipl_cvmeansqval_inter_f . . . 26
csipl_cvmeansqval_split_f . . . 26
csipl_mmeansqval_f . . . 26
csipl_cmmeansqval_inter_f . . . 26
csipl_cmmeansqval_split_f . . . 27
csipl_vmodulate_f . . . 27
csipl_cvmodulate_inter_f . . . 27
csipl_cvmodulate_split_f . . . 27
csipl_vneg_P . . . 27
csipl_cvneg_inter_f . . . 27
csipl_cvneg_split_f . . . 27
csipl_mneg_P . . . 28

csipl_cmneg_inter_f . . . 28
csipl_cmneg_split_f . . . 28
csipl_vrecip_f . . . 28
csipl_cvrecip_inter_f . . . 28
csipl_cvrecip_split_f . . . 28
csipl_mrecip_f . . . 28
csipl_cmrecip_inter_f . . . 29
csipl_cmrecip_split_f . . . 29
csipl_vrsqrt_f . . . 29
csipl_mrsqrt_f . . . 29
csipl_vsq_f . . . 29
csipl_msq_f . . . 29
csipl_cvsumval_inter_P . . . 29
csipl_cvsumval_split_P . . . 29
csipl_msumval_P . . . 30
csipl_cmsumval_inter_P . . . 30
csipl_cmsumval_split_P . . . 30
csipl_msumsqval_f . . . 30
6.3 Binary Operations . . . 30
csipl_vadd_P . . . 30
csipl_cvadd_inter_P . . . 30
csipl_cvadd_split_P . . . 30
csipl_madd_P . . . 31
csipl_cmadd_inter_f . . . 31
csipl_cmadd_split_f . . . 31
csipl_rcvadd_inter_f . . . 31
csipl_rcvadd_split_f . . . 31
csipl_rcmadd_inter_f . . . 31
csipl_rcmadd_split_f . . . 32
csipl_svadd_P . . . 32
csipl_csvadd_inter_f . . . 32
csipl_csvadd_split_f . . . 32
csipl_smadd_P . . . 32
csipl_csmadd_inter_f . . . 32
csipl_csmadd_split_f . . . 33
csipl_rscvadd_inter_f . . . 33
csipl_rscvadd_split_f . . . 33
csipl_rscmadd_inter_f . . . 33

csipl_rscmadd_split_f . . . 33
csipl_Dvdiv_P . . . 33
csipl_Dvdiv_inter_P . . . 34
csipl_Dvdiv_split_P . . . 34
csipl_mdiv_f . . . 34
csipl_cmdiv_inter_f . . . 34
csipl_cmdiv_split_f . . . 34
csipl_rcvdiv_inter_f . . . 34
csipl_rcvdiv_split_f . . . 35
csipl_rcmdiv_inter_f . . . 35
csipl_rcmdiv_split_f . . . 35
csipl_crmdiv_inter_f . . . 35
csipl_crmdiv_split_f . . . 35
csipl_rscmsub_inter_f . . . 35
csipl_rscmsub_split_f . . . 36
csipl_vsdiv_f . . . 36
csipl_cvrsdiv_inter_f . . . 36
csipl_cvrsdiv_split_f . . . 36
csipl_msdiv_f . . . 36
csipl_cmrsdiv_inter_f . . . 36
csipl_cmrsdiv_split_f . . . 37
csipl_vexpoavg_f . . . 37
csipl_cvexpoavg_inter_f . . . 37
csipl_cvexpoavg_split_f . . . 37
csipl_mexpoavg_f . . . 37
csipl_cmexpoavg_inter_f . . . 37
csipl_cmexpoavg_split_f . . . 38
csipl_vhypot_f . . . 38
csipl_mhypot_f . . . 38
csipl_cvjmul_inter_f . . . 38
csipl_cvjmul_split_f . . . 38
csipl_cmjmul_inter_f . . . 38
csipl_cmjmul_split_f . . . 39
csipl_Dvmul_P . . . 39
csipl_cvmul_inter_f . . . 39
csipl_cvmul_split_f . . . 39
csipl_mmul_P . . . 39
csipl_cmmul_inter_f . . . 40

csipl_cmmul_split_f 40
csipl_rcvmul_inter_f 40
csipl_rcvmul_split_f 40
csipl_rcmmul_inter_f 40
csipl_rcmmul_split_f 41
csipl_rscvmul_inter_f 41
csipl_rscvmul_split_f 41
csipl_csvmul_inter_f 41
csipl_csvmul_split_f 41
csipl_smmul_f 41
csipl_csmmul_inter_f 42
csipl_csmmul_split_f 42
csipl_rscmmul_inter_f 42
csipl_rscmmul_split_f 42
csipl_vmmul_f 42
csipl_cvmmul_inter_f 42
csipl_cvmmul_split_f 43
csipl_rvcmmul_inter_f 43
csipl_rvcmmul_split_f 43
csipl_vsub_P 43
csipl_cvsub_inter_f 43
csipl_cvsub_split_f 44
csipl_msub_P 44
csipl_cmsub_inter_f 44
csipl_cmsub_split_f 44
csipl_crmsub_inter_f 44
csipl_crmsub_split_f 45
csipl_rcvsub_inter_f 45
csipl_rcvsub_split_f 45
csipl_rcmsub_inter_f 45
csipl_rcmsub_split_f 45
csipl_crvsub_inter_f 45
csipl_crvsub_split_f 46
csipl_svsub_P 46
csipl_csvsub_inter_f 46
csipl_csvsub_split_f 46
csipl_smsub_P 46
csipl_csmsub_inter_f 46

csipl_csmsub_split_f . . . 47
csipl_smdiv_f . . . 47
csipl_csmdiv_inter_f . . . 47
csipl_csmdiv_split_f . . . 47
csipl_rscvdiv_inter_f . . . 47
csipl_rscvdiv_split_f . . . 47
csipl_rscvsub_inter_f . . . 48
csipl_rscvsub_split_f . . . 48
csipl_rscmdiv_inter_f . . . 48
csipl_rscmdiv_split_f . . . 48
6.4 Ternary Operations . . . 48
csipl_vam_P . . . 48
csipl_cvam_inter_P . . . 49
csipl_cvam_split_P . . . 49
csipl_vmsa_f . . . 49
csipl_cvmsa_inter_f . . . 49
csipl_cvmsa_split_f . . . 49
csipl_vmsb_f . . . 50
csipl_cvmsb_inter_f . . . 50
csipl_cvmsb_split_f . . . 50
csipl_vsam_f . . . 50
csipl_cvsam_inter_f . . . 50
csipl_cvsam_split_f . . . 51
csipl_vsbm_f . . . 51
csipl_cvsbm_inter_f . . . 51
csipl_cvsbm_split_f . . . 51
csipl_vsma_f . . . 51
csipl_cvsma_inter_f . . . 52
csipl_cvsma_split_f . . . 52
csipl_vsmsa_f . . . 52
csipl_cvsmsa_inter_f . . . 52
csipl_cvsmsa_split_f . . . 52
6.5 Logical Operations . . . 52
csipl_valltrue_bl . . . 53
csipl_malltrue_bl . . . 53
csipl_vanytrue_bl . . . 53
csipl_manytrue_bl . . . 53
csipl_vleq_P . . . 53

csipl_cvleq_inter_P . . . 53
csipl_cvleq_split_P . . . 53
csipl_mleq_P . . . 53
csipl_cmleq_inter_P . . . 54
csipl_cmleq_split_P . . . 54
csipl_vlge_P . . . 54
csipl_mlge_P . . . 54
csipl_vlgt_P . . . 54
csipl_mlgt_P . . . 54
csipl_vlle_P . . . 55
csipl_mlle_P . . . 55
csipl_vllt_P . . . 55
csipl_mllt_P . . . 55
csipl_vlne_P . . . 55
csipl_cvlne_inter_P . . . 55
csipl_cvlne_split_P . . . 56
csipl_mlne_P . . . 56
csipl_cmlne_inter_P . . . 56
csipl_cmlne_split_P . . . 56
6.6 Selection Operations . . . 56
csipl_vclip_P . . . 56
csipl_vinvclip_P . . . 57
csipl_vindexbool . . . 57
csipl_vmax_f . . . 57
csipl_vmaxmg_f . . . 57
csipl_vcmaxmgsq_inter_f . . . 57
csipl_vcmaxmgsq_split_f . . . 57
csipl_vcmaxmgsqval_inter_f . . . 57
csipl_vcmaxmgsqval_split_f . . . 58
csipl_vmaxmgval_f . . . 58
csipl_vmaxval_f . . . 58
csipl_vmin_f . . . 58
csipl_vminmg_f . . . 58
csipl_vcminmgsq_inter_f . . . 58
csipl_vcminmgsq_split_f . . . 58
csipl_vcminmgsqval_inter_f . . . 59
csipl_vcminmgsqval_split_f . . . 59
csipl_vminmgval_f . . . 59

csipl_vminval_f 59
6.7 Bitwise and Boolean Logical Operators 59
csipl_vand_P 59
csipl_mand_P 59
csipl_vnot_P 59
csipl_mnot_P 60
csipl_vor_P 60
csipl_mor_P 60
csipl_mxor_P 60
6.8 Element Generation and Copy 60
csipl_vcopy_P_P 61
csipl_cvcopy_inter_f_f 61
csipl_cvcopy_split_f_f 61
csipl_mcopy_P_P 61
csipl_cmcopy_inter_f_f 62
csipl_cmcopy_split_f_f 62
csipl_vfill_P 62
csipl_cvfill_inter_f 62
csipl_cvfill_split_f 62
csipl_mfill_P 62
csipl_cmfill_inter_f 62
csipl_cmfill_split_f 62
csipl_vramp_P 63
6.9 Manipulation Operations 63
csipl_vcmplx_inter_f 63
csipl_vcmplx_split_f 63
csipl_mcmplx_f 63
csipl_vgather_P 63
csipl_cvgather_inter_f 64
csipl_cvgather_split_f 64
csipl_mgather_P 64
csipl_cmgather_inter_f 64
csipl_cmgather_split_f 64
csipl_vimag_inter_f 64
csipl_vimag_split_f 65
csipl_mimag_f 65
csipl_vpolar_inter_f 65
csipl_vpolar_split_f 65

csipl_mpolar_f . . . 65
csipl_vreal_inter_f . . . 65
csipl_vreal_split_f . . . 65
csipl_mreal_f . . . 66
csipl_vrect_inter_f . . . 66
csipl_vrect_split_f . . . 66
csipl_mrect_f . . . 66
csipl_vscatter_P . . . 66
csipl_cvscatter_inter_f . . . 66
csipl_cvscatter_split_f . . . 67
csipl_mscatter_P . . . 67
csipl_cmscatter_inter_f . . . 67
csipl_cmscatter_split_f . . . 67
csipl_cvswap_inter_f . . . 67
csipl_cvswap_split_f . . . 67
csipl_mswap_P . . . 68
csipl_cmswap_inter_f . . . 68
csipl_cmswap_split_f . . . 68

7 Signal Processing Functions **69**

7.1 FFT Functions . . . 69
csipl_ccfftip_create_f . . . 69
csipl_ccfftop_create_f . . . 69
csipl_crfftop_create_f . . . 69
csipl_rcfftop_create_f . . . 69
csipl_ccfftip_inter_f . . . 69
csipl_ccfftip_split_f . . . 69
csipl_ccfftop_inter_f . . . 69
csipl_ccfftop_split_f . . . 69
csipl_crfftop_inter_f . . . 70
csipl_crfftop_split_f . . . 70
csipl_rcfftop_inter_f . . . 70
csipl_rcfftop_split_f . . . 70
csipl_fft_destroy_f . . . 70
csipl_fft_getattr_f . . . 70
csipl_ccfftmop_create_f . . . 70
csipl_ccfftmip_create_f . . . 70
csipl_crfftmop_create_f . . . 71

csipl_rcfftmop_create_f . . . 71
csipl_fftm_destroy_f . . . 71
csipl_fftm_getattr_f . . . 71
csipl_ccfftmip_inter_f . . . 71
csipl_ccfftmip_split_f . . . 71
csipl_ccfftmop_inter_f . . . 71
csipl_ccfftmop_split_f . . . 71
csipl_crfftmop_inter_f . . . 71
csipl_crfftmop_split_f . . . 72
csipl_rcfftmop_inter_f . . . 72
csipl_rcfftmop_split_f . . . 72
csipl_ccfft2dop_create_f . . . 72
csipl_ccfft2dip_create_f . . . 72
csipl_crfft2dop_create_f . . . 72
csipl_rcfft2dop_create_f . . . 72
csipl_ccfft2dip_inter_f . . . 72
csipl_ccfft2dip_split_f . . . 73
csipl_ccfft2dop_inter_f . . . 73
csipl_ccfft2dop_split_f . . . 73
csipl_crfft2dop_inter_f . . . 73
csipl_crfft2dop_split_f . . . 73
csipl_rcfft2dop_inter_f . . . 73
csipl_rcfft2dop_split_f . . . 73
csipl_fft2d_destroy_f . . . 73
csipl_fft2d_getattr_f . . . 73
7.2 Convolution/Correlation Functions . . . 73
csipl_conv1d_create_f . . . 74
csipl_conv1d_destroy_f . . . 74
csipl_conv1d_getattr_f . . . 74
csipl_convolve1d_f . . . 74
csipl_conv2d_create_f . . . 74
csipl_conv2d_destroy_f . . . 74
csipl_conv2d_getattr_f . . . 74
csipl_convolve2d_f . . . 74
csipl_Dcorr1d_create_P . . . 74
csipl_Dcorr1d_destroy_P . . . 75
csipl_Dcorr1d_getattr_P . . . 75
csipl_correlate1d_f . . . 75

csipl_ccorrelate1d_inter_f . . . 75
csipl_ccorrelate1d_split_f . . . 75
csipl_Dcorr2d_create_P . . . 75
csipl_Dcorr2d_destroy_P . . . 75
csipl_Dcorr2d_getattr_P . . . 76
csipl_correlate2d_f . . . 76
csipl_ccorrelate2d_inter_f . . . 76
csipl_ccorrelate2d_split_f . . . 76
7.3 Window Functions . . . 76
csipl_vcreate_blackman_f . . . 76
csipl_vcreate_cheby_f . . . 76
csipl_vcreate_hanning_f . . . 77
csipl_vcreate_kaiser_f . . . 77
7.4 Filter Functions . . . 77
csipl_fir_create_f . . . 77
csipl_cfir_create_inter_f . . . 77
csipl_cfir_create_split_f . . . 77
csipl_Dfir_destroy_P . . . 77
csipl_firflt_f . . . 77
csipl_cfirflt_inter_f . . . 78
csipl_cfirflt_split_f . . . 78
csipl_Dfir_getattr_P . . . 78
csipl_Dfir_reset_P . . . 78
7.5 Miscellaneous Signal Processing Functions . . . 79
csipl_vhisto_f . . . 79

8 Linear Algebra **80**
8.1 Matrix and Vector Operations . . . 80
csipl_cmherm_inter_f . . . 80
csipl_cmherm_split_f . . . 80
csipl_cvjdot_inter_f . . . 80
csipl_cvjdot_split_f . . . 80
csipl_gemp_f . . . 80
csipl_cgemp_inter_f . . . 81
csipl_cgemp_split_f . . . 81
csipl_gems_f . . . 81
csipl_cgems_inter_f . . . 81
csipl_cgems_split_f . . . 82

csipl_mprod_P . . . 82
csipl_cmprod_inter_P . . . 82
csipl_cmprod_split_P . . . 82
csipl_cmprodh_inter_P . . . 82
csipl_cmprodh_split_P . . . 83
csipl_cmprodj_inter_P . . . 83
csipl_cmprodj_split_P . . . 83
csipl_mprodt_P . . . 83
csipl_cmprodt_inter_P . . . 83
csipl_cmprodt_split_P . . . 84
csipl_mvprod_P . . . 84
csipl_cmvprod_inter_P . . . 84
csipl_cmvprod_split_P . . . 84
csipl_mtrans_P . . . 84
csipl_cmtrans_inter_P . . . 84
csipl_cmtrans_split_P . . . 85
csipl_vdot_f . . . 85
csipl_cvdot_inter_f . . . 85
csipl_cvdot_split_f . . . 85
csipl_vmprod_P . . . 85
csipl_cvmprod_inter_P . . . 85
csipl_cvmprod_split_P . . . 86
csipl_vouter_f . . . 86
csipl_cvouter_inter_f . . . 86
csipl_cvouter_split_f . . . 86
csipl_vcsummgval_inter_f . . . 86
csipl_vcsummgval_split_f . . . 86
csipl_minvlu_f . . . 87
csipl_cminvlu_inter_f . . . 87
csipl_cminvlu_split_f . . . 87
8.2 Special Linear System Solvers . . . 88
csipl_covsol_f . . . 88
csipl_ccovsol_inter_f . . . 88
csipl_ccovsol_split_f . . . 88
csipl_llsqsol_f . . . 88
csipl_cllsqsol_inter_f . . . 88
csipl_cllsqsol_split_f . . . 88
csipl_toepsol_f . . . 89

csipl_ctoepsol_inter_f . . . 89
csipl_ctoepsol_split_f . . . 89
8.3 General Square Linear System Solver . . . 89
csipl_lud_f . . . 89
csipl_clud_inter_f . . . 89
csipl_clud_split_f . . . 89
csipl_Dlud_create_P . . . 90
csipl_Dlud_destroy_P . . . 90
csipl_Dlud_getattr_P . . . 90
csipl_lusol_f . . . 90
csipl_clusol_inter_f . . . 90
csipl_clusol_split_f . . . 90
8.4 Symmetric Positive Definite Linear System Solver . . . 90
csipl_chold_f . . . 90
csipl_cchold_inter_f . . . 90
csipl_cchold_split_f . . . 91
csipl_chold_create_f . . . 91
csipl_cchold_create_f . . . 91
csipl_Dchold_destroy_P . . . 91
csipl_Dchold_getattr_P . . . 91
csipl_cholsol_f . . . 91
csipl_ccholsol_inter_f . . . 91
csipl_ccholsol_split_f . . . 91
8.5 Overdetermined Linear System Solver . . . 91
csipl_qrd_f . . . 91
csipl_cqrd_inter_f . . . 92
csipl_cqrd_split_f . . . 92
csipl_qrd_create_f . . . 92
csipl_cqrd_create_f . . . 92
csipl_Dqrd_destroy_P . . . 92
csipl_Dqrd_getattr_P . . . 92
csipl_qrdprodq_f . . . 92
csipl_cqrdprodq_inter_f . . . 92
csipl_cqrdprodq_split_f . . . 93
csipl_qrdsolr_f . . . 93
csipl_cqrdsolr_inter_f . . . 93
csipl_cqrdsolr_split_f . . . 93
csipl_qrsol_f . . . 93

csipl_cqrsol_inter_f 93
csipl_cqrsol_split_f 94

Chapter 1. Introduction

This document describes the functions available in the CSIPL library of vector, signal processing, and linear algebra routines.

1.1 Types

The following base types are available:

`csipl_bool`	boolean
`csipl_cscalar_f`	complex floating-point scalar
`csipl_index`	index to a vector
`csipl_index_mi`	index (pair of integers) to a matrix
`csipl_length`	dimension of a vector or matrix
`csipl_stride`	stride or jump of a vector or matrix

If you are using a version of the library that has 64-bit pointers, you must define the symbol `POINTER_SIZE_64BIT` at compile time: this will ensure that variables of types `csipl_length` and `csipl_stride` are 64-bit integers.

The library also passes information around in abstract data types. These objects are opaque structures (implemented as incomplete typedefs) and they can only be created, accessed, and destroyed with library functions that reference them via a pointer. Some are used to describe the data layout in memory; others store information on filters, matrix decompositions, and so on. Some objects have a 'get attribute' function that allows the user access to the internal values.

1.2 Symbols and Flags

The following symbolic constants are defined.

```
CSIPL_TRUE
CSIPL_FALSE
```

Other symbols are defined in enumerated types. The valid choices are listed with each function description.

1.3 Vector Variables

A vector is passed into a function with three parameters:

- a pointer to the start of the data
- an integer for the distance between processed elements (stride); stride = 1 (unit stride) means every element of a vector has to be processed, stride = 2 means every other element, and so on

- an integer for the number of elements accessed (length).

Stride and length are measured in number of data elements (not bytes). For a compound type like complex, this means the number of (real,imaginary) pairs and not the number of atomic type elements (floats for example).

When several vectors are passed into a function, each has its own stride, but usually they all share a common length. In the function prototypes, each vector variable has two parameters (a data pointer followed by a stride) and the common processed length is usually the last parameter.

When a stride is negative the processing progresses backwards through the data so a suitable offset must be added to the data pointer. With proper pointer arithmetic the offset is also measured in number of data elements.

1.4 Matrix Variables

Matrices are stored in row-major format. A matrix is passed into a function with four parameters:

- a pointer to the start of the data
- an integer for the distance between adjacent elements in a column of the matrix (leading dimension); it is denoted by `ldM` for matrix `M`; it can be viewed as the number of columns in the 2-dimensional array containing `M`;
- an integer for the number of rows
- an integer for the number of columns (less than or equal to the leading dimension).

As with vectors, the leading dimension is measured in number of data elements.

When several matrices are passed into a function, each has its own leading dimension, but often they all have the same dimensions, or have one dimension in common. In the function prototypes, each matrix variable has two parameters (a data pointer followed by a leading dimension); the dimension parameters (numbers of row and columns) are often common for several matrices and are usually the last items in the parameter list. The description of each function explains the dimensions of the matrices.

1.5 Complex Variables

Most functions that use complex data are available in two versions: one accepts data stored in interleaved format and each complex variable is passed as a single parameter; the other takes data stored in split format and each complex variable is passed via two parameters — one each for the real and imaginary parts.

1.6 Function Names

For almost all functions, the suffix indicates the base datatype of the arguments as follows:

`f`	float
`i`	integer
`bl`	boolean
`vi`	vector index
`mi`	matrix index.

1.7 Hints

They following mechanisms are provided for the programmer to indicate preferences for optimisation: currently **almost all are ignored** but they are reserved for future use. The only exception is the algorithm hint in FIR filters.

- Flags of the enumerated type `csipl_memory_hint` specified when allocating or creating some objects.
- Flags of the enumerated type `csipl_alg_hint` used to indicate whether algorithmic optimisation should minimise execution time, memory use, or maximise numerical accuracy.
- An indication of how many times an object will be used (filters and FFTs have such a parameter).

1.8 Notation

The following standard mathematical notation is used in the function descriptions.

$:=$	assignment operator
i	square root of -1
$\lvert x \rvert$	absolute value of the real number x
$\lvert z \rvert$	modulus of the complex number z
$\lfloor x \rfloor$	floor of the real number x (largest integer less than or equal to x)
$\lceil x \rceil$	ceiling of the real number x (smallest integer greater than or equal to x)
z^*	conjugate of the complex number z
M^{T}	transpose of the matrix M
M^{H}	Hermitian (conjugate transpose) of the complex matrix M

Note that in expressions i is always the square root of -1; vectors and matrices are indexed with j and k.

An elementwise operation on vectors will be written $C[j * \text{strideC}] := A[j * \text{strideA}] + B[j * \text{strideB}]$. Often the range of the index variable is not given explicity; in such cases it is clear from the context that it runs over all the elements in the vectors and that the lengths of the vectors must be equal.

An M by N matrix has M rows and N columns.

1.9 Errors and Restrictions

Many functions require that their arguments be **conformant**. This means that the objects passed have compatible attributes: for example, size and shape of matrices, lengths of vectors or filter kernels.

If an argument is required to be **valid**, it means:

- a pointer is not `NULL`
- a flag is a member of the required enumerated type
- an object has been initialised and not destroyed.

Errors can occur for the following reasons:

1. argument is outside the domain for calculation
2. over/underflow during calculation
3. failure to allocate memory
4. algorithm failure because of inappropriate data (as when a matrix does not have full rank)
5. arguments are invalid, out of range, or non-conformant.

Only errors of type 5 are regarded as fatal: in this case, the development version of the library will write a message to `stderr` and call `exit`.

Errors of types 3 and 4 are signalled through the return value of the function. A create function will return `NULL` if the allocation fails; functions with integer return codes use zero to indicate success.

The calling program is not alerted to errors of types 1 and 2.

Chapter 2. Getting the Best Performance

This section is a short guide for programmers using the NAS CSIPL Library. It contains explanations of library behaviour, and tips on selecting the right storage options for your data to increase performance.

2.1 Version Information

Information about the version of the NAS CSIPL library you are using can be found in the comments at the top of the include file `csipl.h`. There is no way that a program can determine the library version at run-time.

2.2 Memory Alignment

The efficiency of many operations is improved if data within memory is correctly aligned on certain word boundaries.

Vectors and matrices can be loaded and stored faster if they are vector aligned.

The following table gives the vector alignment and minimum vector length for float data:

Technology	Vector Aligment
SSE	16
AVX	32
AltiVec	16

Alignment can be controlled using a function such as `memalign`. This is a C function that is not in the ANSI standard but is available on many systems. It is defined in `malloc.h` on Linux systems.

The following macro redefines `malloc` so that all memory allocation is optimally aligned:

```
#include <malloc.h>
#define malloc(SIZE) memalign(16, SIZE)
```

Some operating systems (*e.g.* Apple's OSX) automatically align all memory to a 16-byte boundary so `memalign` is not needed.

2.3 Vector/Matrix Format

When available, vector and matrix calculations are done using single instruction, multiple data (SIMD) instructions to process several elements simultaneously. This imposes a minimum vector length given in the table below:

Technology	minimum Vector Length (floats)
SSE	4
AVX	8
AltiVec	4

For short ints (16 bits) the vector length should be twice that of the float vector length. If the vector unit supports doubles (64 bits), then the vector length should be half that of floats.

For best performance all input and output vectors should:

- have a stride of 1

  Vectors and matrices can be loaded and stored much quicker when they are contiguous in memory. The library includes special optimisations for a stride length of 2 (which was added for interleaved complex numbers), but all other non-unit strides will be significantly slower than a stride of 1 and, in many cases, almost as slow as unvectorised scalar code. Note that, a stride of -1 will also be significantly slower than a stride of $+1$.

- be vector aligned

- have length greater than or equal to the vector length

  The vector unit works on arrays of the vector length so no speed up is gained by using the library on vectors of length less than this.

- have row (row major matrices) or column (column major matrices) length divisible by the vector length

  For a row major matrix: if the row length of a matrix is not divisible by the vector length then the alignment of the first element of each row will change for each row/column. For optimal performance the first element of each row should be vector aligned.

  The same rule applies to columns in column major matrices.

- have a length divisible by the vector length

  Any elements at the end of the vector which cannot be dealt with by the vector unit must be dealt with in normal scalar code, which will decrease the performance. The decrease in performance becomes less important for longer vectors.

2.4 Error Checking and Debugging

Two versions of the NAS CSIPL library are provided: a performance version and a development version. The development version of the library (signified by a

'D' in the library's name) contains full error checking and should always be used when developing and debugging applications.

A few library functions return status information: always check the return code of those that do.

The performance version of the library contains no error checking, and consequently runs faster than the development library. The performance library should only be used with applications that have been run successfully with the development version of the library.

When timing code, the performance version of the library should be used.

Note: the performance version of the library reads in data before it knows how much will be used and as a result often reads more data than is needed. This is not a problem, except when using memory checkers such as Electric Fence which object to this behaviour. The development library only reads in the data it intends to use and so is safe to use with memory checkers.

2.5 Support Functions

Always call `csip_init` and `csip_finalize` at the beginning and end of a program.

Note: For the AltiVec optimized library, calling `csip_init` will put the AltiVec unit into non-Java mode if it is not already. This speeds up most AltiVec instructions.

2.6 Scalar Functions

As the vector unit works on arrays of the vector length, scalar functions in the library are not vectorised.

2.7 Random Number Generation

The random number generation functions have not been vectorised in the current version of the library.

2.8 Vector and Elementwise Operations

All vector and elementwise operations work optimally on vectors which match the conditions given in Section 2.3.

2.9 Signal Processing Functions

All signal processing operations work optimally on vectors which match the conditions given in Section 2.3.

Most of the signal processing routines are split into three stages:

- a create stage

- a compute stage
- a destroy stage.

The library has been optimised to minimise the time taken to do the compute stage, which means as much precomputation as possible is done in the create stage. If you are using the same signal processing routine on many vectors of the same length, it is far quicker to just create the required signal processing object once and reuse it for each computation stage rather than recreating the object each time it is needed.

2.10 FFT Functions

To get the best performance from an FFT, a vector must have length a multiple of the numbers 2, 4, 8, and 3 only. If a vector length is not a multiple of these numbers, a DFT may be done, which is considerably slower than an FFT.

Factors of 3 should be avoided if possible. An FFT will only be done for factors of 3 if the length also has a factor of 16, otherwise a DFT is done.

When doing large FFT's, optimal routines have been developed for the lengths: 4096, 8192, 16384, 32768, and 65536. These lengths should be much quicker than lengths of similar magnitude.

In-place FFT's are normally faster than out-of-place FFT's.

FFT's are fastest with a scale factor of 1. However, if you need to use a different scale factor, it is better to let the FFT routine do the scaling rather than to do it yourself.

The internal FFT routines only work on vector aligned data with a stride of 1. If vectors are used which do not match these restrictions an internal copy of the vector will be made. This is an important consideration when using large vectors. Also, if complex vectors are not stored split an internal copy will be made.

The current version of the NAS CSIPL library does not have any special FFT routines for doing multiple FFT's, so the time to do n single FFT's will be approximately the same as using the multiple FFT routines on a matrix of n rows.

The `ntimes` parameter to the FFT functions is ignored. The algorithmic hint is only used in the FFT create function: if the `CSIP_ALG_NOISE` hint is used, the FFT create function will take significantly longer. By default, the algorithms are optimised to minimise execution time.

2.11 FIR Filter, Convolution and Correlation Functions

These functions call the FFT functions internally and are therefore subject to the same restrictions.

Hints are ignored with the exception of the internal calling of the FFT create function described in the FFT functions section.

2.12 Linear Algebra Functions

For optimal performance the vectors and matrices used with the linear algebra functions should match the conditions given in Section 2.3. (See also the sections below when using complex LU, complex Cholesky, or complex QRD functions).

2.13 Matrix and Vector Operations

Matrix and vector operations should work optimally on row or column major matrices (row major is the default), however, the restriction exists that all matrices passed to a function should be of the same order. For example, using two row major matrices as input to a function and a column major as output will be slower than using all row major or all column major. When matrices are passed to NAS CSIPL functions that are not all of the same order, the library will assume they are all row major and treat the column major matrices as strided matrices. (See also the sections below when using LU, Cholesky, or QRD functions).

2.14 LU Decomposition, Cholesky and QRD Functions

These functions have three separate stages:

- a create stage
- a compute stage
- a destroy stage.

The library has been optimised to minimise the time taken to do the compute stage, which means as much precomputation as possible is done in the create stage. If you are using the linear algebra routine on many matrices of the same size, it is far quicker to just create the required linear algebra object once and reuse it for each computation stage rather than recreating the object each time it is needed.

If matrices of different orders or strided matrices are passed to these functions, an internal copy will be made of the entire matrix before the computation is done.

This is an important consideration when using large matrices. (Note: unaligned matrices do NOT require internal copying provided they have a stride of one and all matrices used with the functions are of the same order).

When using the QRD functions, it is only necessary to save the Q matrix if using the `qrdprodq` function; the `qrsol` and `qrdsolr` do not need the Q matrix.

2.15 Special Linear System Solvers

The `covsol` and `llsqsol` functions internally use the QRD functions and so have the same requirements for optimal performance.

The `toepsol` functions are based on vector operations and so have the same requirements for optimal performance.

2.16 Controlling the Number of Threads

The NAS CSIPL library is multithreaded and will take advantage of multiple cores on the processor invoking it. Utilising multiple threads is automatic:

- The maximum number of threads used is set when `csip_init` is called.
- The maximum number running at any one time is also set at that point. If a threaded routine is called with (say) 4 threads and we have hit this maximum number running then four of them are shut down before the new function is executed.
- The number of threads invoked when a routine is called, is decided by that routine by reference to the data (vector or matrix) size specified in the call, to provide the best performance for that call.

It is possible to change the maximum number of threads used.

1. A threaded, and a non-threaded ("serial") version of the library are provided. If you wish to only ever use one thread in a library call, use the serial version of the library.
2. The maximum number of threads used for a specific function call, and the maximum number kept running at any one time, can be changed by a call to the routine `Thread_SetParams` with arguments `num_threads` and `max_num_running`. This call, if used, *must* be made before the library initialisation routine `csip_init` is called.

   If no call to `Thread_SetParams` is made, the library default values will be utilised.

Chapter 3. Support Functions

3.1 Initilialisation and Finalisation

Prototype	Description
`int csipl_init( void * ptr);`	Provides initialisation, allowing the library to allocate and set a global state, and prepare to support the use of CSIPL functionality by the user.
`int csipl_finalize( void * ptr);`	Provides cleanup and releases resources used by CSIPL (if the last of a nested series of calls), allowing the library to guarantee that any resources allocated by `csipl_init` are no longer in use after the call is complete.

3.2 Sundry Functions

Prototype	Description
`void csipl_complete( void);`	Force all deferred CSIPL execution to complete.
`csipl_cmplx_mem csipl_cstorage( void);`	Returns the preferred complex storage format for the system.

Chapter 4. Scalar Functions

4.1 Real Scalar Functions

Prototype	Description
`float csipl_acos_f( float A);`	Computes the principal radian value in $[0, \pi]$ of the inverse cosine of a scalar.
`float csipl_asin_f( float A);`	Computes the principal radian value in $[0, \pi]$ of the inverse sine of a scalar.
`float csipl_atan_f( float A);`	Computes the principal radian value in $[-\pi/2, \pi/2]$ of the inverse tangent of a scalar.
`float csipl_atan2_f( float A, float B);`	Computes the four-quadrant radian value in $[-\pi, \pi]$ of the inverse tangent of the ratio of two scalars.
`float csipl_ceil_f( float A);`	Computes the ceiling of a scalar.
`float csipl_cos_f( float A);`	Computes the cosine of a scalar angle in radians.
`float csipl_cosh_f( float A);`	Computes the hyperbolic cosine of a scalar.
`float csipl_exp_f( float A);`	Computes the exponential of a scalar.
`float csipl_floor_f( float A);`	Computes the floor of a scalar.
`float csipl_log_f( float A);`	Computes the natural logarithm of a scalar.
`float csipl_log10_f( float A);`	Computes the base 10 logarithm of a scalar.
`float csipl_mag_f( float A);`	Computes the magnitude (absolute value) of a scalar.
`float csipl_pow_f( float A, float B);`	Computes the power function of two scalars.
`float csipl_sin_f( float A);`	Computes the sine of a scalar angle in radians.
`float csipl_sinh_f( float A);`	Computes the hyperbolic sine of a scalar.
`float csipl_sqrt_f( float A);`	Computes the square root of a scalar.
`float csipl_tan_f( float A);`	Computes the tangent of a scalar angle in radians.

Prototype	Description
`float csipl_tanh_f( float A);`	Computes the hyperbolic tangent of a scalar.

4.2 Complex Scalar Functions

Prototype	Description
`float csipl_arg_f( csipl_cscalar_f x);`	Returns the argument in radians $[-\pi, \pi]$ of a complex scalar.
`csipl_cscalar_f csipl_cadd_f( csipl_cscalar_f x, csipl_cscalar_f y);`	Computes the complex sum of two scalars.
`csipl_cscalar_f csipl_rcadd_f( float x, csipl_cscalar_f y);`	Computes the complex sum of two scalars.
`csipl_cscalar_f csipl_cdiv_f( csipl_cscalar_f x, csipl_cscalar_f y);`	Computes the complex quotient of two scalars.
`csipl_cscalar_f csipl_crdiv_f( csipl_cscalar_f x, float y);`	Computes the complex quotient of two scalars.
`csipl_cscalar_f csipl_cexp_f( csipl_cscalar_f A);`	Computes the exponential of a scalar.
`csipl_cscalar_f csipl_cjmul_f( csipl_cscalar_f x, csipl_cscalar_f y);`	Computes the product a complex scalar with the conjugate of a second complex scalar.
`csipl_cscalar_f csipl_cmag_f( csipl_cscalar_f A);`	Computes the magnitude (absolute value) of a scalar.
`float csipl_cmagsq_f( csipl_cscalar_f x);`	Computes the magnitude squared of a complex scalar.
`csipl_cscalar_f csipl_cmplx_f( float re, float im);`	Form a complex scalar from two real scalars.
`csipl_cscalar_f csipl_cmul_f( csipl_cscalar_f x, csipl_cscalar_f y);`	Computes the complex product of two scalars.
`csipl_cscalar_f csipl_rcmul_f( float x, csipl_cscalar_f y);`	Computes the complex product of two scalars.
`csipl_cscalar_f csipl_cneg_f( csipl_cscalar_f x);`	Computes the negation of a complex scalar.
`csipl_cscalar_f csipl_conj_f( csipl_cscalar_f x);`	Computes the complex conjugate of a scalar.
`csipl_cscalar_f csipl_crecip_f( csipl_cscalar_f x);`	Computes the reciprocal of a complex scalar.

Prototype	Description
`csipl_cscalar_f csipl_csqrt_f( csipl_cscalar_f x);`	Computes the square root a complex scalar.
`csipl_cscalar_f csipl_csub_f( csipl_cscalar_f x, csipl_cscalar_f y);`	Computes the complex difference of two scalars.
`csipl_cscalar_f csipl_rcsub_f( float x, csipl_cscalar_f y);`	Computes the complex difference of two scalars.
`csipl_cscalar_f csipl_crsub_f( csipl_cscalar_f x, float y);`	Computes the complex difference of two scalars.
`float csipl_imag_f( csipl_cscalar_f x);`	Extract the imaginary part of a complex scalar.
`void csipl_polar_f( csipl_cscalar_f a, float *r, float *t);`	Convert a complex scalar from rectangular to polar form. The polar data consists of a real scalar containing the radius and a corresponding real scalar containing the argument (angle) of the complex scalar.
`float csipl_real_f( csipl_cscalar_f x);`	Extract the real part of a complex scalar.
`csipl_cscalar_f csipl_rect_f( float r, float t);`	Convert a pair of real scalars from complex polar to complex rectangular form.

4.3 Index Scalar Functions

Prototype	Description
`csipl_scalar_mi csipl_matindex( csipl_index r, csipl_index c);`	Form a matrix index from two vector indices.
`csipl_index csipl_mcolindex( csipl_scalar_mi mi);`	Returns the column vector index from a matrix index.
`csipl_index csipl_mrowindex( csipl_scalar_mi mi);`	Returns the row vector index from a matrix index.

Chapter 5. Random Number Generation

5.1 Random Number Functions

Prototype	Description
csipl_randstate * csipl_randcreate(csipl_index seed, csipl_index numprocs, csipl_index id, csipl_rng portable);	Create a random number generator state object.
int csipl_randdestroy(csipl_randstate * rand);	Destroys (frees the memory used by) a random number generator state object. Returns zero on success, non-zero on failure.
float csipl_randu_f(csipl_randstate * state);	Generate a uniformly distributed (pseudo-)random number. Floating point values are uniformly distributed over the open interval (0,1). Integer deviates are uniformly distributed over the open interval $(0, 2^{31} - 1)$.
csipl_cscalar_f csipl_crandu_f(csipl_randstate * state);	Generate a uniformly distributed (pseudo-)random number. Floating point values are uniformly distributed over the open interval (0,1). Integer deviates are uniformly distributed over the open interval $(0, 2^{31} - 1)$.
void csipl_vrandu_f(csipl_randstate * state, float * R, csipl_stride strideR, csipl_length n);	Generate a uniformly distributed (pseudo-)random number. Floating point values are uniformly distributed over the open interval (0,1). Integer deviates are uniformly distributed over the open interval $(0, 2^{31} - 1)$.
void csipl_cvrandu_inter_f(csipl_randstate * state, void * R, csipl_stride strideR, csipl_length n);	Generate a uniformly distributed (pseudo-)random number. Floating point values are uniformly distributed over the open interval (0,1). Integer deviates are uniformly distributed over the open interval $(0, 2^{31} - 1)$.
void csipl_cvrandu_split_f(csipl_randstate * state, float * R_re, float * R_im, csipl_stride strideR, csipl_length n);	Generate a uniformly distributed (pseudo-)random number. Floating point values are uniformly distributed over the open interval (0,1). Integer deviates are uniformly distributed over the open interval $(0, 2^{31} - 1)$.
float csipl_randn_f(csipl_randstate * state);	Generate an approximately normally distributed (pseudo-)random deviate having mean zero and unit variance: $N(0, 1)$. The random numbers are generated by summing values returned by the uniform random number generator.

Prototype	Description
csipl_cscalar_f csipl_crandn_f(csipl_randstate * state);	Generate an approximately normally distributed (pseudo-)random deviate having mean zero and unit variance: $N(0, 1)$. The random numbers are generated by summing values returned by the uniform random number generator.
void csipl_vrandn_f(csipl_randstate * state, float * R, csipl_stride strideR, csipl_length n);	Generate an approximately normally distributed (pseudo-)random deviate having mean zero and unit variance: $N(0, 1)$. The random numbers are generated by summing values returned by the uniform random number generator.
void csipl_cvrandn_inter_f(csipl_randstate * state, void * R, csipl_stride strideR, csipl_length n);	Generate an approximately normally distributed (pseudo-)random deviate having mean zero and unit variance: $N(0, 1)$. The random numbers are generated by summing values returned by the uniform random number generator.
void csipl_cvrandn_split_f(csipl_randstate * state, float * R_re, float * R_im, csipl_stride strideR, csipl_length n);	Generate an approximately normally distributed (pseudo-)random deviate having mean zero and unit variance: $N(0, 1)$. The random numbers are generated by summing values returned by the uniform random number generator.

Chapter 6. Vector And Elementwise Operations

6.1 Elementary Mathematical Functions

Prototype	Description
`void csipl_vacos_f( float * A, csipl_stride strideA, float * R, csipl_stride strideR, csipl_length n);`	Computes the principal radian value in $[0, \pi]$ of the inverse cosine for each element of a vector.
`void csipl_macos_f( float * A, int ldA, float * R, int ldR, csipl_length m, csipl_length n);`	Computes the principal radian value in $[0, \pi]$ of the inverse cosine for each element of a matrix.
`void csipl_vasin_f( float * A, csipl_stride strideA, float * R, csipl_stride strideR, csipl_length n);`	Computes the principal radian value in $[0, \pi]$ of the inverse sine for each element of a vector.
`void csipl_masin_f( float * A, int ldA, float * R, int ldR, csipl_length m, csipl_length n);`	Computes the principal radian value in $[0, \pi]$ of the inverse sine for each element of a matrix.
`void csipl_vatan_f( float * A, csipl_stride strideA, float * R, csipl_stride strideR, csipl_length n);`	Computes the principal radian value in $[-\pi/2, \pi/2]$ of the inverse tangent for each element of a vector.
`void csipl_matan_f( float * A, int ldA, float * R, int ldR, csipl_length m, csipl_length n);`	Computes the principal radian value in $[-\pi/2, \pi/2]$ of the inverse tangent for each element of a matrix.
`void csipl_vatan2_f( float * A, csipl_stride strideA, float * B, csipl_stride strideB, float * R, csipl_stride strideR, csipl_length n);`	Computes the four-quadrant radian value in $[-\pi, \pi]$ of the inverse tangent of the ratio of the elements of two input vectors.

Prototype	Description
void csipl_matan2_f(float * A, int ldA, float * B, int ldB, float * R, int ldR, csipl_length m, csipl_length n);	Computes the four-quadrant radian value in $[-\pi, \pi]$ of the inverse tangent of the ratio of the elements of two input matrices.
void csipl_vcos_f(float * A, csipl_stride strideA, float * R, csipl_stride strideR, csipl_length n);	Computes the cosine for each element of a vector. Element angle values are in radians.
void csipl_mcos_f(float * A, int ldA, float * R, int ldR, csipl_length m, csipl_length n);	Computes the cosine for each element of a matrix. Element angle values are in radians.
void csipl_vcosh_f(float * A, csipl_stride strideA, float * R, csipl_stride strideR, csipl_length n);	Computes the hyperbolic cosine for each element of a vector.
void csipl_mcosh_f(float * A, int ldA, float * R, int ldR, csipl_length m, csipl_length n);	Computes the hyperbolic cosine for each element of a matrix.
void csipl_vexp_f(float * A, csipl_stride strideA, float * R, csipl_stride strideR, csipl_length n);	Computes the exponential function value for each element of a vector.
void csipl_cvexp_inter_f(void * A, csipl_stride strideA, void * R, csipl_stride strideR, csipl_length n);	Computes the exponential function value for each element of a vector.

Prototype	Description
void csipl_cvexp_split_f(float * A_re, float * A_im, csipl_stride strideA, float * R_re, float * R_im, csipl_stride strideR, csipl_length n);	Computes the exponential function value for each element of a vector.
void csipl_mexp_f(float * A, int ldA, float * R, int ldR, csipl_length m, csipl_length n);	Computes the exponential function value for each element of a matrix.
void csipl_cmexp_inter_f(void * A, int ldA, void * R, int ldR, csipl_length m, csipl_length n);	Computes the exponential function value for each element of a matrix.
void csipl_cmexp_split_f(float * A_re, float * A_im, int ldA, float * R_re, float * R_im, int ldR, csipl_length m, csipl_length n);	Computes the exponential function value for each element of a matrix.
void csipl_vexp10_f(float * A, csipl_stride strideA, float * R, csipl_stride strideR, csipl_length n);	Computes the base 10 exponential for each element of a vector.
void csipl_mexp10_f(float * A, int ldA, float * R, int ldR, csipl_length m, csipl_length n);	Computes the base 10 exponential for each element of a matrix.
void csipl_vfloor_f(float * A, csipl_stride strideA, float * R, csipl_stride strideR, csipl_length n);	Computes the floor for each element of a vector.

Prototype	Description
void csipl_vlog_f(float * A, csipl_stride strideA, float * R, csipl_stride strideR, csipl_length n);	Computes the natural logarithm for each element of a vector.
void csipl_cvlog_inter_f(void * A, csipl_stride strideA, void * R, csipl_stride strideR, csipl_length n);	Computes the natural logarithm for each element of a vector.
void csipl_cvlog_split_f(float * A_re, float * A_im, csipl_stride strideA, float * R_re, float * R_im, csipl_stride strideR, csipl_length n);	Computes the natural logarithm for each element of a vector.
void csipl_mlog_f(float * A, int ldA, float * R, int ldR, csipl_length m, csipl_length n);	Computes the natural logarithm for each element of a matrix.
void csipl_cmlog_inter_f(void * A, int ldA, void * R, int ldR, csipl_length m, csipl_length n);	Computes the natural logarithm for each element of a matrix.
void csipl_cmlog_split_f(float * A_re, float * A_im, int ldA, float * R_re, float * R_im, int ldR, csipl_length m, csipl_length n);	Computes the natural logarithm for each element of a matrix.
void csipl_vlog10_f(float * A, csipl_stride strideA, float * R, csipl_stride strideR, csipl_length n);	Compute the base ten logarithm for each element of a vector.

Prototype	Description
void csipl_mlog10_f(float * A, int ldA, float * R, int ldR, csipl_length m, csipl_length n);	Compute the base ten logarithm for each element of a matrix.
void csipl_vsin_f(float * A, csipl_stride strideA, float * R, csipl_stride strideR, csipl_length n);	Compute the sine for each element of a vector. Element angle values are in radians.
void csipl_msin_f(float * A, int ldA, float * R, int ldR, csipl_length m, csipl_length n);	Compute the sine for each element of a matrix. Element angle values are in radians.
void csipl_vsinh_f(float * A, csipl_stride strideA, float * R, csipl_stride strideR, csipl_length n);	Computes the hyperbolic sine for each element of a vector.
void csipl_msinh_f(float * A, int ldA, float * R, int ldR, csipl_length m, csipl_length n);	Computes the hyperbolic sine for each element of a matrix.
void csipl_vsqrt_f(float * A, csipl_stride strideA, float * R, csipl_stride strideR, csipl_length n);	Compute the square root for each element of a vector.
void csipl_cvsqrt_inter_f(void * A, csipl_stride strideA, void * R, csipl_stride strideR, csipl_length n);	Compute the square root for each element of a vector.
void csipl_cvsqrt_split_f(float * A_re, float * A_im, csipl_stride strideA, float * R_re, float * R_im, csipl_stride strideR, csipl_length n);	Compute the square root for each element of a vector.

Prototype	Description
void csipl_msqrt_f(float * A, int ldA, float * R, int ldR, csipl_length m, csipl_length n);	Compute the square root for each element of a matrix.
void csipl_cmsqrt_inter_f(void * A, int ldA, void * R, int ldR, csipl_length m, csipl_length n);	Compute the square root for each element of a matrix.
void csipl_cmsqrt_split_f(float * A_re, float * A_im, int ldA, float * R_re, float * R_im, int ldR, csipl_length m, csipl_length n);	Compute the square root for each element of a matrix.
void csipl_mtan_f(float * A, int ldA, float * R, int ldR, csipl_length m, csipl_length n);	Compute the tangent for each element of a matrix. Element angle values are in radians.
void csipl_mtanh_f(float * A, int ldA, float * R, int ldR, csipl_length m, csipl_length n);	Computes the hyperbolic tangent for each element of a matrix.

6.2 Unary Operations

Prototype	Description
void csipl_varg_f(void * A, csipl_stride strideA, float * R, csipl_stride strideR, csipl_length n);	Computes the argument in radians $[-\pi, \pi]$ for each element of a complex vector.
void csipl_marg_f(void * A, int ldA, float * R, int ldR, csipl_length m, csipl_length n);	Computes the argument in radians $[-\pi, \pi]$ for each element of a complex matrix.

Prototype	Description
void csipl_vceil_f(float * A, csipl_stride strideA, float * R, csipl_stride strideR, csipl_length n);	Computes the ceiling for each element of a vector.
void csipl_cvconj_inter_f(void * A, csipl_stride strideA, void * R, csipl_stride strideR, csipl_length n);	Compute the conjugate for each element of a complex vector.
void csipl_cvconj_split_f(float * A_re, float * A_im, csipl_stride strideA, float * R_re, float * R_im, csipl_stride strideR, csipl_length n);	Compute the conjugate for each element of a complex vector.
void csipl_cmconj_inter_f(void * A, int ldA, void * R, int ldR, csipl_length m, csipl_length n);	Compute the conjugate for each element of a complex matrix.
void csipl_cmconj_split_f(float * A_re, float * A_im, int ldA, float * R_re, float * R_im, int ldR, csipl_length m, csipl_length n);	Compute the conjugate for each element of a complex matrix.
void csipl_vcumsum_P(scalar_P * A, csipl_stride strideA, scalar_P * R, csipl_stride strideR, csipl_length n);	Compute the cumulative sum of the elements of a vector. The following instances are supported: csipl_vcumsum_f csipl_vcumsum_i
void csipl_cvcumsum_inter_P(void * A, csipl_stride strideA, void * R, csipl_stride strideR, csipl_length n);	Compute the cumulative sum of the elements of a vector. The following instances are supported: csipl_cvcumsum_inter_f csipl_cvcumsum_inter_i

Prototype	Description
void csipl_cvcumsum_split_P(scalar_P * A_re, scalar_P * A_im, csipl_stride strideA, scalar_P * R_re, scalar_P * R_im, csipl_stride strideR, csipl_length n);	Compute the cumulative sum of the elements of a vector. The following instances are supported: csipl_cvcumsum_split_f csipl_cvcumsum_split_i
void csipl_mcumsum_P(csipl_major dir, scalar_P * R, int ldR, csipl_length m, csipl_length n);	Compute the cumulative sums of the elements in the rows or columns of a matrix. The following instances are supported: csipl_mcumsum_f csipl_mcumsum_i
void csipl_cmcumsum_inter_P(void * A, csipl_stride strideA, void * R, csipl_stride strideR, csipl_length n);	Compute the cumulative sum of the elements of a vector. The following instances are supported: csipl_cmcumsum_inter_f csipl_cmcumsum_inter_i
void csipl_cmcumsum_split_P(scalar_P * A_re, scalar_P * A_im, csipl_stride strideA, scalar_P * R_re, scalar_P * R_im, csipl_stride strideR, csipl_length n);	Compute the cumulative sum of the elements of a vector. The following instances are supported: csipl_cmcumsum_split_f csipl_cmcumsum_split_i
void csipl_veuler_f(float * A, csipl_stride strideA, void * R, csipl_stride strideR, csipl_length n);	Computes the complex numbers corresponding to the angle of a unit vector in the complex plane for each element of a vector.
void csipl_meuler_f(float * A, int ldA, void * R, int ldR, csipl_length m, csipl_length n);	Computes the complex numbers corresponding to the angle of a unit vector in the complex plane for each element of a matrix.
void csipl_vmag_P(scalar_P * A, csipl_stride strideA, scalar_P * R, csipl_stride strideR, csipl_length n);	Compute the magnitude for each element of a vector. The following instances are supported: csipl_vmag_f csipl_vmag_i csipl_vmag_si
void csipl_cvmag_inter_f(void * A, csipl_stride strideA, float * R, csipl_stride strideR, csipl_length n);	Compute the magnitude for each element of a vector.

Prototype	Description
void csipl_cvmag_split_f(float * A_re, float * A_im, csipl_stride strideA, float * R, csipl_stride strideR, csipl_length n);	Compute the magnitude for each element of a vector.
void csipl_mmag_f(float * A, int ldA, float * R, int ldR, csipl_length m, csipl_length n);	Compute the magnitude for each element of a matrix.
void csipl_cmmag_inter_f(void * A, int ldA, float * R, int ldR, csipl_length m, csipl_length n);	Compute the magnitude for each element of a matrix.
void csipl_cmmag_split_f(float * A_re, float * A_im, int ldA, float * R, int ldR, csipl_length m, csipl_length n);	Compute the magnitude for each element of a matrix.
void csipl_vcmagsq_inter_f(void * A, csipl_stride strideA, float * R, csipl_stride strideR, csipl_length n);	Computes the square of the magnitudes for each element of a vector.
void csipl_vcmagsq_split_f(float * A_re, float * A_im, csipl_stride strideA, float * R, csipl_stride strideR, csipl_length n);	Computes the square of the magnitudes for each element of a vector.
void csipl_mcmagsq_f(void * A, int ldA, float * R, int ldR, csipl_length m, csipl_length n);	Computes the square of the magnitudes for each element of a matrix.
float csipl_vmeanval_f(float * A, csipl_stride strideA, csipl_length n);	Returns the mean value of the elements of a vector.

Prototype	Description
csipl_cscalar_f csipl_cvmeanval_inter_f(void * A, csipl_stride strideA, csipl_length n);	Returns the mean value of the elements of a vector.
csipl_cscalar_f csipl_cvmeanval_split_f(float * A_re, float * A_im, csipl_stride strideA, csipl_length n);	Returns the mean value of the elements of a vector.
float csipl_mmeanval_f(float * A, int ldA, csipl_length m, csipl_length n);	Returns the mean value of the elements of a matrix.
csipl_cscalar_f csipl_cmmeanval_inter_f(void * A, int ldA, csipl_length m, csipl_length n);	Returns the mean value of the elements of a matrix.
csipl_cscalar_f csipl_cmmeanval_split_f(float * A_re, float * A_im, int ldA, csipl_length m, csipl_length n);	Returns the mean value of the elements of a matrix.
float csipl_vmeansqval_f(float * A, csipl_stride strideA, csipl_length n);	Returns the mean magnitude squared value of the elements of a vector.
float csipl_cvmeansqval_inter_f(void * A, csipl_stride strideA, csipl_length n);	Returns the mean magnitude squared value of the elements of a vector.
float csipl_cvmeansqval_split_f(float * A_re, float * A_im, csipl_stride strideA, csipl_length n);	Returns the mean magnitude squared value of the elements of a vector.
float csipl_mmeansqval_f(float * A, int ldA, csipl_length m, csipl_length n);	Returns the mean magnitude squared value of the elements of a matrix.
float csipl_cmmeansqval_inter_f(void * A, int ldA, csipl_length m, csipl_length n);	Returns the mean magnitude squared value of the elements of a matrix.

Prototype	Description
float csipl_cmmeansqval_split_f(float * A_re, float * A_im, int ldA, csipl_length m, csipl_length n);	Returns the mean magnitude squared value of the elements of a matrix.
float csipl_vmodulate_f(float * A, csipl_stride strideA, float nu, float phi, float * R, csipl_stride strideR, csipl_length n);	Computes the modulation of a real vector by a specified complex frequency.
float csipl_cvmodulate_inter_f(void * A, csipl_stride strideA, float nu, float phi, void * R, csipl_stride strideR, csipl_length n);	Computes the modulation of a real vector by a specified complex frequency.
float csipl_cvmodulate_split_f(float * A_re, float * A_im, csipl_stride strideA, float nu, float phi, float * R_re, float * R_im, csipl_stride strideR, csipl_length n);	Computes the modulation of a real vector by a specified complex frequency.
void csipl_vneg_P(scalar_P * A, csipl_stride strideA, scalar_P * R, csipl_stride strideR, csipl_length n);	Computes the negation for each element of a vector. The following instances are supported: csipl_vneg_f csipl_vneg_i csipl_vneg_si
void csipl_cvneg_inter_f(void * A, csipl_stride strideA, void * R, csipl_stride strideR, csipl_length n);	Computes the negation for each element of a vector.
void csipl_cvneg_split_f(float * A_re, float * A_im, csipl_stride strideA, float * R_re, float * R_im, csipl_stride strideR, csipl_length n);	Computes the negation for each element of a vector.

Prototype	Description
void csipl_mneg_P(scalar_P * A, int ldA, scalar_P * R, int ldR, csipl_length m, csipl_length n);	Computes the negation for each element of a matrix. The following instances are supported: csipl_mneg_f csipl_mneg_i
void csipl_cmneg_inter_f(void * A, int ldA, void * R, int ldR, csipl_length m, csipl_length n);	Computes the negation for each element of a matrix.
void csipl_cmneg_split_f(float * A_re, float * A_im, int ldA, float * R_re, float * R_im, int ldR, csipl_length m, csipl_length n);	Computes the negation for each element of a matrix.
void csipl_vrecip_f(float * A, csipl_stride strideA, float * R, csipl_stride strideR, csipl_length n);	Computes the reciprocal for each element of a vector.
void csipl_cvrecip_inter_f(void * A, csipl_stride strideA, void * R, csipl_stride strideR, csipl_length n);	Computes the reciprocal for each element of a vector.
void csipl_cvrecip_split_f(float * A_re, float * A_im, csipl_stride strideA, float * R_re, float * R_im, csipl_stride strideR, csipl_length n);	Computes the reciprocal for each element of a vector.
void csipl_mrecip_f(float * A, int ldA, float * R, int ldR, csipl_length m, csipl_length n);	Computes the reciprocal for each element of a matrix.

Prototype	Description
void csipl_cmrecip_inter_f(void * A, int ldA, void * R, int ldR, csipl_length m, csipl_length n);	Computes the reciprocal for each element of a matrix.
void csipl_cmrecip_split_f(float * A_re, float * A_im, int ldA, float * R_re, float * R_im, int ldR, csipl_length m, csipl_length n);	Computes the reciprocal for each element of a matrix.
void csipl_vrsqrt_f(float * A, csipl_stride strideA, float * R, csipl_stride strideR, csipl_length n);	Computes the reciprocal of the square root for each element of a vector.
void csipl_mrsqrt_f(float * A, int ldA, float * R, int ldR, csipl_length m, csipl_length n);	Computes the reciprocal of the square root for each element of a matrix.
void csipl_vsq_f(float * A, csipl_stride strideA, float * R, csipl_stride strideR, csipl_length n);	Computes the square for each element of a vector.
void csipl_msq_f(float * A, int ldA, float * R, int ldR, csipl_length m, csipl_length n);	Computes the square for each element of a matrix.
scalar_P csipl_cvsumval_inter_P(scalar_P * A, csipl_stride strideA, csipl_length n);	Returns the sum of the elements of a vector. The following instances are supported: csipl_cvsumval_inter_f csipl_cvsumval_inter_i
scalar_P csipl_cvsumval_split_P(scalar_P * A_re, scalar_P * A_im, csipl_stride strideA, csipl_length n);	Returns the sum of the elements of a vector. The following instances are supported: csipl_cvsumval_split_f csipl_cvsumval_split_i

Prototype	Description
scalar_P csipl_msumval_P(scalar_P * A, csipl_stride strideA, csipl_length n);	Returns the sum of the elements of a vector. The following instances are supported: csipl_msumval_f csipl_msumval_i
scalar_P csipl_cmsumval_inter_P(scalar_P * A, csipl_stride strideA, csipl_length n);	Returns the sum of the elements of a vector. The following instances are supported: csipl_cmsumval_inter_f csipl_cmsumval_inter_i
scalar_P csipl_cmsumval_split_P(scalar_P * A_re, scalar_P * A_im, csipl_stride strideA, csipl_length n);	Returns the sum of the elements of a vector. The following instances are supported: csipl_cmsumval_split_f csipl_cmsumval_split_i
float csipl_msumsqval_f(float * A, int ldA, csipl_length m, csipl_length n);	Returns the sum of the squares of the elements of a matrix.

6.3 Binary Operations

Prototype	Description
void csipl_vadd_P(scalar_P * A, csipl_stride strideA, scalar_P * B, csipl_stride strideB, scalar_P * R, csipl_stride strideR, csipl_length n);	Computes the sum, by element, of two vectors. The following instances are supported: csipl_vadd_f csipl_vadd_i csipl_vadd_si
void csipl_cvadd_inter_P(void * A, csipl_stride strideA, void * B, csipl_stride strideB, void * R, csipl_stride strideR, csipl_length n);	Computes the sum, by element, of two vectors. The following instances are supported: csipl_cvadd_inter_f csipl_cvadd_inter_i
void csipl_cvadd_split_P(scalar_P * A_re, scalar_P * A_im, csipl_stride strideA, scalar_P * B_re, scalar_P * B_im, csipl_stride strideB, scalar_P * R_re, scalar_P * R_im, csipl_stride strideR, csipl_length n);	Computes the sum, by element, of two vectors. The following instances are supported: csipl_cvadd_split_f csipl_cvadd_split_i

Prototype	Description
void csipl_madd_P(scalar_P * A, int ldA, scalar_P * B, int ldB, scalar_P * R, int ldR, csipl_length m, csipl_length n);	Computes the sum, by element, of two matrices. The following instances are supported: csipl_madd_f csipl_madd_i
void csipl_cmadd_inter_f(void * A, int ldA, void * B, int ldB, void * R, int ldR, csipl_length m, csipl_length n);	Computes the sum, by element, of two matrices.
void csipl_cmadd_split_f(float * A_re, float * A_im, int ldA, float * B_re, float * B_im, int ldB, float * R_re, float * R_im, int ldR, csipl_length m, csipl_length n);	Computes the sum, by element, of two matrices.
void csipl_rcvadd_inter_f(float * A, csipl_stride strideA, void * B, csipl_stride strideB, void * R, csipl_stride strideR, csipl_length n);	Computes the sum, by element, of two vectors.
void csipl_rcvadd_split_f(float * A, csipl_stride strideA, float * B_re, float * B_im, csipl_stride strideB, float * R_re, float * R_im, csipl_stride strideR, csipl_length n);	Computes the sum, by element, of two vectors.
void csipl_rcmadd_inter_f(float * A, int ldA, void * B, int ldB, void * R, int ldR, csipl_length m, csipl_length n);	Computes the sum, by element, of two matrices.

Prototype	Description
void csipl_rcmadd_split_f(float * A, int ldA, float * B_re, float * B_im, int ldB, float * R_re, float * R_im, int ldR, csipl_length m, csipl_length n);	Computes the sum, by element, of two matrices.
void csipl_svadd_P(scalar_P a, scalar_P * B, csipl_stride strideB, scalar_P * R, csipl_stride strideR, csipl_length n);	Computes the sum, by element, of a scalar and a vector. The following instances are supported: csipl_svadd_f csipl_svadd_i csipl_svadd_si
void csipl_csvadd_inter_f(csipl_cscalar_f a, void * B, csipl_stride strideB, void * R, csipl_stride strideR, csipl_length n);	Computes the sum, by element, of a scalar and a vector.
void csipl_csvadd_split_f(csipl_cscalar_f a_re, csipl_cscalar_f a_im, float * B_re, float * B_im, csipl_stride strideB, float * R_re, float * R_im, csipl_stride strideR, csipl_length n);	Computes the sum, by element, of a scalar and a vector.
void csipl_smadd_P(scalar_P a, scalar_P * B, int ldB, scalar_P * R, int ldR, csipl_length m, csipl_length n);	Computes the sum, by element, of a scalar and a matrix. The following instances are supported: csipl_smadd_f csipl_smadd_i
void csipl_csmadd_inter_f(csipl_cscalar_f a, void * B, int ldB, void * R, int ldR, csipl_length m, csipl_length n);	Computes the sum, by element, of a scalar and a matrix.

Prototype	Description
void csipl_csmadd_split_f(csipl_cscalar_f a_re, csipl_cscalar_f a_im, float * B_re, float * B_im, int ldB, float * R_re, float * R_im, int ldR, csipl_length m, csipl_length n);	Computes the sum, by element, of a scalar and a matrix.
void csipl_rscvadd_inter_f(float a, void * B, csipl_stride strideB, void * R, csipl_stride strideR, csipl_length n);	Computes the sum, by element, of a real scalar and a complex vector.
void csipl_rscvadd_split_f(float a, float * B_re, float * B_im, csipl_stride strideB, float * R_re, float * R_im, csipl_stride strideR, csipl_length n);	Computes the sum, by element, of a real scalar and a complex vector.
void csipl_rscmadd_inter_f(float a, void * B, int ldB, void * R, int ldR, csipl_length m, csipl_length n);	Computes the sum, by element, of a real scalar and a complex matrix.
void csipl_rscmadd_split_f(float a, float * B_re, float * B_im, int ldB, float * R_re, float * R_im, int ldR, csipl_length m, csipl_length n);	Computes the sum, by element, of a real scalar and a complex matrix.
void csipl_Dvdiv_P(csipl_Dvview_P * A, csipl_stride strideA, csipl_Dvview_P * B, csipl_stride strideB, csipl_Dvview_P * R, csipl_stride strideR, csipl_length n);	Computes the quotient, by element, of two vectors. The following instances are supported: csipl_vdiv_f csipl_svdiv_f

Prototype	Description
void csipl_Dvdiv_inter_P(csipl_Dvview_P * A, csipl_stride strideA, csipl_Dvview_P * B, csipl_stride strideB, csipl_Dvview_P * R, csipl_stride strideR, csipl_length n);	Computes the quotient, by element, of two vectors. The following instances are supported: csipl_crvdiv_inter_f csipl_cvdiv_inter_f
void csipl_Dvdiv_split_P(csipl_Dvview_P * A_re, csipl_Dvview_P * A_im, csipl_stride strideA, csipl_Dvview_P * B_re, csipl_Dvview_P * B_im, csipl_stride strideB, csipl_Dvview_P * R_re, csipl_Dvview_P * R_im, csipl_stride strideR, csipl_length n);	Computes the quotient, by element, of two vectors. The following instances are supported: csipl_crvdiv_split_f csipl_cvdiv_split_f
void csipl_mdiv_f(float * A, int ldA, float * B, int ldB, float * R, int ldR, csipl_length m, csipl_length n);	Computes the quotient, by element, of two matrices.
void csipl_cmdiv_inter_f(void * A, int ldA, void * B, int ldB, void * R, int ldR, csipl_length m, csipl_length n);	Computes the quotient, by element, of two matrices.
void csipl_cmdiv_split_f(float * A_re, float * A_im, int ldA, float * B_re, float * B_im, int ldB, float * R_re, float * R_im, int ldR, csipl_length m, csipl_length n);	Computes the quotient, by element, of two matrices.
void csipl_rcvdiv_inter_f(float a, void * B, csipl_stride strideB, void * R, csipl_stride strideR, csipl_length n);	Computes the quotient, by element, of two vectors.

Prototype	Description
void csipl_rcvdiv_split_f(float a, float * B_re, float * B_im, csipl_stride strideB, float * R_re, float * R_im, csipl_stride strideR, csipl_length n);	Computes the quotient, by element, of two vectors.
void csipl_rcmdiv_inter_f(float a, void * B, int ldB, void * R, int ldR, csipl_length m, csipl_length n);	Computes the quotient, by element, of two matrices.
void csipl_rcmdiv_split_f(float a, float * B_re, float * B_im, int ldB, float * R_re, float * R_im, int ldR, csipl_length m, csipl_length n);	Computes the quotient, by element, of two matrices.
void csipl_crmdiv_inter_f(void * A, int ldA, float * B, int ldB, void * R, int ldR, csipl_length m, csipl_length n);	Computes the quotient, by element, of two matrices.
void csipl_crmdiv_split_f(float * A_re, float * A_im, int ldA, float * B, int ldB, float * R_re, float * R_im, int ldR, csipl_length m, csipl_length n);	Computes the quotient, by element, of two matrices.
void csipl_rscmsub_inter_f(float a, void * B, int ldB, void * R, int ldR, csipl_length m, csipl_length n);	Computes the difference, by element, of a real scalar and a complex matrix.

Prototype	Description
void csipl_rscmsub_split_f(float a, float * B_re, float * B_im, int ldB, float * R_re, float * R_im, int ldR, csipl_length m, csipl_length n);	Computes the difference, by element, of a real scalar and a complex matrix.
void csipl_vsdiv_f(float * A, csipl_stride strideA, float b, float * R, csipl_stride strideR, csipl_length n);	Computes the quotient, by element, of a vector and a scalar.
void csipl_cvrsdiv_inter_f(float * A, csipl_stride strideA, float b, float * R, csipl_stride strideR, csipl_length n);	Computes the quotient, by element, of a vector and a scalar.
void csipl_cvrsdiv_split_f(float * A_re, float * A_im, csipl_stride strideA, float b_re, float b_im, float * R_re, float * R_im, csipl_stride strideR, csipl_length n);	Computes the quotient, by element, of a vector and a scalar.
void csipl_msdiv_f(float * A, int ldA, float b, float * R, int ldR, csipl_length m, csipl_length n);	Computes the quotient, by element, of a matrix and a scalar.
void csipl_cmrsdiv_inter_f(void * A, int ldA, csipl_cscalar_f b, void * R, int ldR, csipl_length m, csipl_length n);	Computes the quotient, by element, of a matrix and a scalar.

Prototype	Description
void csipl_cmrsdiv_split_f(float * A_re, float * A_im, int ldA, csipl_cscalar_f b_re, csipl_cscalar_f b_im, float * R_re, float * R_im, int ldR, csipl_length m, csipl_length n);	Computes the quotient, by element, of a matrix and a scalar.
void csipl_vexpoavg_f(float a, float * B, csipl_stride strideB, float * C, csipl_stride strideC, csipl_length n);	Computes an exponential weighted average, by element, of two vectors.
void csipl_cvexpoavg_inter_f(float a, void * B, csipl_stride strideB, void * C, csipl_stride strideC, csipl_length n);	Computes an exponential weighted average, by element, of two vectors.
void csipl_cvexpoavg_split_f(float a, float * B_re, float * B_im, csipl_stride strideB, float * C_re, float * C_im, csipl_stride strideC, csipl_length n);	Computes an exponential weighted average, by element, of two vectors.
void csipl_mexpoavg_f(float a, float * B, int ldB, float * C, int ldC, csipl_length m, csipl_length n);	Computes an exponential weighted average, by element, of two matrices.
void csipl_cmexpoavg_inter_f(float a, void * B, int ldB, void * C, int ldC, csipl_length m, csipl_length n);	Computes an exponential weighted average, by element, of two matrices.

Prototype	Description
void csipl_cmexpoavg_split_f(float a, float * B_re, float * B_im, int ldB, float * C_re, float * C_im, int ldC, csipl_length m, csipl_length n);	Computes an exponential weighted average, by element, of two matrices.
void csipl_vhypot_f(float * A, csipl_stride strideA, float * B, csipl_stride strideB, float * R, csipl_stride strideR, csipl_length n);	Computes the square root of the sum of squares, by element, of two input vectors.
void csipl_mhypot_f(float * A, int ldA, float * B, int ldB, float * R, int ldR, csipl_length m, csipl_length n);	Computes the square root of the sum of squares, by element, of two input matrices.
void csipl_cvjmul_inter_f(void * A, csipl_stride strideA, void * B, csipl_stride strideB, void * R, csipl_stride strideR, csipl_length n);	Computes the product of a complex vector with the conjugate of a second complex vector, by element.
void csipl_cvjmul_split_f(float * A_re, float * A_im, csipl_stride strideA, float * B_re, float * B_im, csipl_stride strideB, float * R_re, float * R_im, csipl_stride strideR, csipl_length n);	Computes the product of a complex vector with the conjugate of a second complex vector, by element.
void csipl_cmjmul_inter_f(void * A, int ldA, void * B, int ldB, void * R, int ldR, csipl_length m, csipl_length n);	Computes the product of a complex matrix with the conjugate of a second complex matrix, by element.

Prototype	Description
void csipl_cmjmul_split_f(float * A_re, float * A_im, int ldA, float * B_re, float * B_im, int ldB, float * R_re, float * R_im, int ldR, csipl_length m, csipl_length n);	Computes the product of a complex matrix with the conjugate of a second complex matrix, by element.
void csipl_Dvmul_P(csipl_Dvview_P * A, csipl_stride strideA, csipl_Dvview_P * B, csipl_stride strideB, csipl_Dvview_P * R, csipl_stride strideR, csipl_length n);	Computes the product, by element, of two vectors. The following instances are supported: csipl_vmul_f csipl_vmul_i csipl_vmul_si csipl_svmul_f csipl_svmul_i csipl_svmul_si
void csipl_cvmul_inter_f(void * A, csipl_stride strideA, void * B, csipl_stride strideB, void * R, csipl_stride strideR, csipl_length n);	Computes the product, by element, of two vectors.
void csipl_cvmul_split_f(float * A_re, float * A_im, csipl_stride strideA, float * B_re, float * B_im, csipl_stride strideB, float * R_re, float * R_im, csipl_stride strideR, csipl_length n);	Computes the product, by element, of two vectors.
void csipl_mmul_P(scalar_P * A, int ldA, scalar_P * B, int ldB, scalar_P * R, int ldR, csipl_length m, csipl_length n);	Computes the product, by element, of two matrices. The following instances are supported: csipl_mmul_f csipl_mmul_i

Prototype	Description
void csipl_cmmul_inter_f(void * A, int ldA, void * B, int ldB, void * R, int ldR, csipl_length m, csipl_length n);	Computes the product, by element, of two matrices.
void csipl_cmmul_split_f(float * A_re, float * A_im, int ldA, float * B_re, float * B_im, int ldB, float * R_re, float * R_im, int ldR, csipl_length m, csipl_length n);	Computes the product, by element, of two matrices.
void csipl_rcvmul_inter_f(float * A, csipl_stride strideA, void * B, csipl_stride strideB, void * R, csipl_stride strideR, csipl_length n);	Computes the product, by element, of two vectors.
void csipl_rcvmul_split_f(float * A, csipl_stride strideA, float * B_re, float * B_im, csipl_stride strideB, float * R_re, float * R_im, csipl_stride strideR, csipl_length n);	Computes the product, by element, of two vectors.
void csipl_rcmmul_inter_f(float * A, int ldA, void * B, int ldB, void * R, int ldR, csipl_length m, csipl_length n);	Computes the product, by element, of two matrices.

Prototype	Description
void csipl_rcmmul_split_f(float * A, int ldA, float * B_re, float * B_im, int ldB, float * R_re, float * R_im, int ldR, csipl_length m, csipl_length n);	Computes the product, by element, of two matrices.
void csipl_rscvmul_inter_f(float a, void * B, csipl_stride strideB, void * R, csipl_stride strideR, csipl_length n);	Computes the product, by element, of a real scalar and a complex vector.
void csipl_rscvmul_split_f(float a, float * B_re, float * B_im, csipl_stride strideB, float * R_re, float * R_im, csipl_stride strideR, csipl_length n);	Computes the product, by element, of a real scalar and a complex vector.
void csipl_csvmul_inter_f(csipl_cscalar_f a, void * B, csipl_stride strideB, void * R, csipl_stride strideR, csipl_length n);	Computes the product, by element, of a scalar and a vector.
void csipl_csvmul_split_f(csipl_cscalar_f a_re, csipl_cscalar_f a_im, float * B_re, float * B_im, csipl_stride strideB, float * R_re, float * R_im, csipl_stride strideR, csipl_length n);	Computes the product, by element, of a scalar and a vector.
void csipl_smmul_f(float a, float * B, int ldB, float * R, int ldR, csipl_length m, csipl_length n);	Computes the product, by element, of a scalar and a matrix.

Prototype	Description
void csipl_csmmul_inter_f(csipl_cscalar_f a, void * B, int ldB, void * R, int ldR, csipl_length m, csipl_length n);	Computes the product, by element, of a scalar and a matrix.
void csipl_csmmul_split_f(csipl_cscalar_f a_re, csipl_cscalar_f a_im, float * B_re, float * B_im, int ldB, float * R_re, float * R_im, int ldR, csipl_length m, csipl_length n);	Computes the product, by element, of a scalar and a matrix.
void csipl_rscmmul_inter_f(float a, void * B, int ldB, void * R, int ldR, csipl_length m, csipl_length n);	Computes the product, by element, of a real scalar and a complex matrix.
void csipl_rscmmul_split_f(float a, float * B_re, float * B_im, int ldB, float * R_re, float * R_im, int ldR, csipl_length m, csipl_length n);	Computes the product, by element, of a real scalar and a complex matrix.
void csipl_vmmul_f(float * A, csipl_stride strideA, float * B, int ldB, csipl_major major, float * R, int ldR, csipl_length m, csipl_length n);	Computes the product, by element, of a vector and the rows or columns of a matrix.
void csipl_cvmmul_inter_f(void * A, csipl_stride strideA, void * B, int ldB, csipl_major major, void * R, int ldR, csipl_length m, csipl_length n);	Computes the product, by element, of a vector and the rows or columns of a matrix.

Prototype	Description
void csipl_cvmmul_split_f(csipl_Dvview_f * A_re, csipl_Dvview_f * A_im, csipl_stride strideA, csipl_Dmview_f * B_re, csipl_Dmview_f * B_im, int ldB, csipl_major major, csipl_Dmview_f * R_re, csipl_Dmview_f * R_im, int ldR, csipl_length m, csipl_length n);	Computes the product, by element, of a vector and the rows or columns of a matrix.
void csipl_rvcmmul_inter_f(float * A, csipl_stride strideA, void * B, int ldB, csipl_major major, void * R, int ldR, csipl_length m, csipl_length n);	Computes the product, by element, of a vector and the rows or columns of a matrix.
void csipl_rvcmmul_split_f(float * A, csipl_stride strideA, float * B_re, float * B_im, int ldB, csipl_major major, float * R_re, float * R_im, int ldR, csipl_length m, csipl_length n);	Computes the product, by element, of a vector and the rows or columns of a matrix.
void csipl_vsub_P(scalar_P * A, csipl_stride strideA, scalar_P * B, csipl_stride strideB, scalar_P * R, csipl_stride strideR, csipl_length n);	Computes the difference, by element, of two vectors. The following instances are supported: csipl_vsub_f csipl_vsub_i csipl_vsub_si
void csipl_cvsub_inter_f(void * A, csipl_stride strideA, void * B, csipl_stride strideB, void * R, csipl_stride strideR, csipl_length n);	Computes the difference, by element, of two vectors.

Prototype	Description
void csipl_cvsub_split_f(float * A_re, float * A_im, csipl_stride strideA, float * B_re, float * B_im, csipl_stride strideB, float * R_re, float * R_im, csipl_stride strideR, csipl_length n);	Computes the difference, by element, of two vectors.
void csipl_msub_P(scalar_P * A, int ldA, scalar_P * B, int ldB, scalar_P * R, int ldR, csipl_length m, csipl_length n);	Computes the difference, by element, of two matrices. The following instances are supported: csipl_msub_f csipl_msub_i
void csipl_cmsub_inter_f(void * A, int ldA, void * B, int ldB, void * R, int ldR, csipl_length m, csipl_length n);	Computes the difference, by element, of two matrices.
void csipl_cmsub_split_f(float * A_re, float * A_im, int ldA, float * B_re, float * B_im, int ldB, float * R_re, float * R_im, int ldR, csipl_length m, csipl_length n);	Computes the difference, by element, of two matrices.
void csipl_crmsub_inter_f(void * A, int ldA, float * B, int ldB, void * R, int ldR, csipl_length m, csipl_length n);	Computes the difference, by element, of two matrices.

Prototype	Description
void csipl_crmsub_split_f(float * A_re, float * A_im, int ldA, float * B, int ldB, float * R_re, float * R_im, int ldR, csipl_length m, csipl_length n);	Computes the difference, by element, of two matrices.
void csipl_rcvsub_inter_f(float * A, csipl_stride strideA, void * B, csipl_stride strideB, void * R, csipl_stride strideR, csipl_length n);	Computes the difference, by element, of two vectors.
void csipl_rcvsub_split_f(float * A, csipl_stride strideA, float * B_re, float * B_im, csipl_stride strideB, float * R_re, float * R_im, csipl_stride strideR, csipl_length n);	Computes the difference, by element, of two vectors.
void csipl_rcmsub_inter_f(float * A, int ldA, void * B, int ldB, void * R, int ldR, csipl_length m, csipl_length n);	Computes the difference, by element, of two matrices.
void csipl_rcmsub_split_f(float * A, int ldA, float * B_re, float * B_im, int ldB, float * R_re, float * R_im, int ldR, csipl_length m, csipl_length n);	Computes the difference, by element, of two matrices.
void csipl_crvsub_inter_f(void * A, csipl_stride strideA, float * B, csipl_stride strideB, void * R, csipl_stride strideR, csipl_length n);	Computes the difference, by element, of two vectors.

Prototype	Description
void csipl_crvsub_split_f(float * A_re, float * A_im, csipl_stride strideA, float * B, csipl_stride strideB, float * R_re, float * R_im, csipl_stride strideR, csipl_length n);	Computes the difference, by element, of two vectors.
void csipl_svsub_P(scalar_P a, scalar_P * B, csipl_stride strideB, scalar_P * R, csipl_stride strideR, csipl_length n);	Computes the difference, by element, of a scalar and a vector. The following instances are supported: csipl_svsub_f csipl_svsub_i csipl_svsub_si
void csipl_csvsub_inter_f(csipl_cscalar_f a, void * B, csipl_stride strideB, void * R, csipl_stride strideR, csipl_length n);	Computes the difference, by element, of a scalar and a vector.
void csipl_csvsub_split_f(csipl_cscalar_f a_re, csipl_cscalar_f a_im, float * B_re, float * B_im, csipl_stride strideB, float * R_re, float * R_im, csipl_stride strideR, csipl_length n);	Computes the difference, by element, of a scalar and a vector.
void csipl_smsub_P(scalar_P a, scalar_P * B, int ldB, scalar_P * R, int ldR, csipl_length m, csipl_length n);	Computes the difference, by element, of a scalar and a matrix. The following instances are supported: csipl_smsub_f csipl_smsub_i
void csipl_csmsub_inter_f(csipl_cscalar_f a, void * B, int ldB, void * R, int ldR, csipl_length m, csipl_length n);	Computes the difference, by element, of a scalar and a matrix.

Prototype	Description
void csipl_csmsub_split_f(csipl_cscalar_f a_re, csipl_cscalar_f a_im, float * B_re, float * B_im, int ldB, float * R_re, float * R_im, int ldR, csipl_length m, csipl_length n);	Computes the difference, by element, of a scalar and a matrix.
void csipl_smdiv_f(float a, float * B, int ldB, float * R, int ldR, csipl_length m, csipl_length n);	Computes the quotient, by element, of a scalar and a matrix.
void csipl_csmdiv_inter_f(csipl_cscalar_f a, void * B, int ldB, void * R, int ldR, csipl_length m, csipl_length n);	Computes the quotient, by element, of a scalar and a matrix.
void csipl_csmdiv_split_f(csipl_cscalar_f a_re, csipl_cscalar_f a_im, float * B_re, float * B_im, int ldB, float * R_re, float * R_im, int ldR, csipl_length m, csipl_length n);	Computes the quotient, by element, of a scalar and a matrix.
void csipl_rscvdiv_inter_f(float a, void * B, csipl_stride strideB, void * R, csipl_stride strideR, csipl_length n);	Computes the quotient, by element, of a real scalar and a complex vector.
void csipl_rscvdiv_split_f(float a, float * B_re, float * B_im, csipl_stride strideB, float * R_re, float * R_im, csipl_stride strideR, csipl_length n);	Computes the quotient, by element, of a real scalar and a complex vector.

Prototype	Description
void csipl_rscvsub_inter_f(float a, void * B, csipl_stride strideB, void * R, csipl_stride strideR, csipl_length n);	Computes the difference, by element, of a real scalar and a complex vector.
void csipl_rscvsub_split_f(float a, float * B_re, float * B_im, csipl_stride strideB, float * R_re, float * R_im, csipl_stride strideR, csipl_length n);	Computes the difference, by element, of a real scalar and a complex vector.
void csipl_rscmdiv_inter_f(float a, void * B, int ldB, void * R, int ldR, csipl_length m, csipl_length n);	Computes the quotient, by element, of a real scalar and a complex matrix.
void csipl_rscmdiv_split_f(float a, float * B_re, float * B_im, int ldB, float * R_re, float * R_im, int ldR, csipl_length m, csipl_length n);	Computes the quotient, by element, of a real scalar and a complex matrix.

6.4 Ternary Operations

Prototype	Description
void csipl_vam_P(scalar_P * A, csipl_stride strideA, scalar_P * B, csipl_stride strideB, scalar_P * C, csipl_stride strideC, scalar_P * R, csipl_stride strideR, csipl_length n);	Computes the sum of two vectors and product of a third vector, by element. The following instances are supported: csipl_vam_f csipl_vma_f

Prototype	Description
void csipl_cvam_inter_P(void * A, csipl_stride strideA, void * B, csipl_stride strideB, void * C, csipl_stride strideC, void * R, csipl_stride strideR, csipl_length n);	Computes the sum of two vectors and product of a third vector, by element. The following instances are supported: csipl_cvam_inter_f csipl_cvma_inter_f
void csipl_cvam_split_P(scalar_P * A_re, scalar_P * A_im, csipl_stride strideA, scalar_P * B_re, scalar_P * B_im, csipl_stride strideB, scalar_P * C_re, scalar_P * C_im, csipl_stride strideC, scalar_P * R_re, scalar_P * R_im, csipl_stride strideR, csipl_length n);	Computes the sum of two vectors and product of a third vector, by element. The following instances are supported: csipl_cvam_split_f csipl_cvma_split_f
void csipl_vmsa_f(float * A, csipl_stride strideA, float * B, csipl_stride strideB, float c, float * R, csipl_stride strideR, csipl_length n);	Computes the product of two vectors and sum of a scalar, by element.
void csipl_cvmsa_inter_f(void * A, csipl_stride strideA, void * B, csipl_stride strideB, csipl_cscalar_f c, void * R, csipl_stride strideR, csipl_length n);	Computes the product of two vectors and sum of a scalar, by element.
void csipl_cvmsa_split_f(float * A_re, float * A_im, csipl_stride strideA, float * B_re, float * B_im, csipl_stride strideB, csipl_cscalar_f c_re, csipl_cscalar_f c_im, float * R_re, float * R_im, csipl_stride strideR, csipl_length n);	Computes the product of two vectors and sum of a scalar, by element.

Prototype	Description
void csipl_vmsb_f(float * A, csipl_stride strideA, float * B, csipl_stride strideB, float * C, csipl_stride strideC, float * R, csipl_stride strideR, csipl_length n);	Computes the product of two vectors and difference of a third vector, by element.
void csipl_cvmsb_inter_f(void * A, csipl_stride strideA, void * B, csipl_stride strideB, void * C, csipl_stride strideC, void * R, csipl_stride strideR, csipl_length n);	Computes the product of two vectors and difference of a third vector, by element.
void csipl_cvmsb_split_f(float * A_re, float * A_im, csipl_stride strideA, float * B_re, float * B_im, csipl_stride strideB, float * C_re, float * C_im, csipl_stride strideC, float * R_re, float * R_im, csipl_stride strideR, csipl_length n);	Computes the product of two vectors and difference of a third vector, by element.
void csipl_vsam_f(float * A, csipl_stride strideA, float b, float * C, csipl_stride strideC, float * R, csipl_stride strideR, csipl_length n);	Computes the sum of a vector and a scalar, and product with a second vector, by element.
void csipl_cvsam_inter_f(void * A, csipl_stride strideA, csipl_cscalar_f b, void * C, csipl_stride strideC, void * R, csipl_stride strideR, csipl_length n);	Computes the sum of a vector and a scalar, and product with a second vector, by element.

Prototype	Description
void csipl_cvsam_split_f(float * A_re, float * A_im, csipl_stride strideA, csipl_cscalar_f b_re, csipl_cscalar_f b_im, float * C_re, float * C_im, csipl_stride strideC, float * R_re, float * R_im, csipl_stride strideR, csipl_length n);	Computes the sum of a vector and a scalar, and product with a second vector, by element.
void csipl_vsbm_f(float * A, csipl_stride strideA, float * B, csipl_stride strideB, float * C, csipl_stride strideC, float * R, csipl_stride strideR, csipl_length n);	Computes the difference of two vectors, and product with a third vector, by element.
void csipl_cvsbm_inter_f(void * A, csipl_stride strideA, void * B, csipl_stride strideB, void * C, csipl_stride strideC, void * R, csipl_stride strideR, csipl_length n);	Computes the difference of two vectors, and product with a third vector, by element.
void csipl_cvsbm_split_f(float * A_re, float * A_im, csipl_stride strideA, float * B_re, float * B_im, csipl_stride strideB, float * C_re, float * C_im, csipl_stride strideC, float * R_re, float * R_im, csipl_stride strideR, csipl_length n);	Computes the difference of two vectors, and product with a third vector, by element.
void csipl_vsma_f(float * A, csipl_stride strideA, float b, float * C, csipl_stride strideC, float * R, csipl_stride strideR, csipl_length n);	Computes the product of a vector and a scalar, and sum with a second vector, by element.

Prototype	Description
void csipl_cvsma_inter_f(void * A, csipl_stride strideA, csipl_cscalar_f b, void * C, csipl_stride strideC, void * R, csipl_stride strideR, csipl_length n);	Computes the product of a vector and a scalar, and sum with a second vector, by element.
void csipl_cvsma_split_f(float * A_re, float * A_im, csipl_stride strideA, csipl_cscalar_f b_re, csipl_cscalar_f b_im, float * C_re, float * C_im, csipl_stride strideC, float * R_re, float * R_im, csipl_stride strideR, csipl_length n);	Computes the product of a vector and a scalar, and sum with a second vector, by element.
void csipl_vsmsa_f(float * A, csipl_stride strideA, float b, float c, float * R, csipl_stride strideR, csipl_length n);	Computes the product of a vector and a scalar, and sum with a second scalar, by element.
void csipl_cvsmsa_inter_f(void * A, csipl_stride strideA, csipl_cscalar_f b, csipl_cscalar_f c, void * R, csipl_stride strideR, csipl_length n);	Computes the product of a vector and a scalar, and sum with a second scalar, by element.
void csipl_cvsmsa_split_f(float * A_re, float * A_im, csipl_stride strideA, csipl_cscalar_f b_re, csipl_cscalar_f b_im, csipl_cscalar_f c_re, csipl_cscalar_f c_im, float * R_re, float * R_im, csipl_stride strideR, csipl_length n);	Computes the product of a vector and a scalar, and sum with a second scalar, by element.

6.5 Logical Operations

Prototype	Description
csipl_scalar_bl csipl_valltrue_bl(signed int * A, csipl_stride strideA, csipl_length n);	Returns true if all the elements of a vector are true.
csipl_scalar_bl csipl_malltrue_bl(signed int * A, csipl_stride strideA, csipl_length n);	Returns true if all the elements of a vector are true.
csipl_scalar_bl csipl_vanytrue_bl(signed int * A, csipl_stride strideA, csipl_length n);	Returns true if one or more elements of a vector are true.
csipl_scalar_bl csipl_manytrue_bl(signed int * A, csipl_stride strideA, csipl_length n);	Returns true if one or more elements of a vector are true.
void csipl_vleq_P(scalar_P * A, csipl_stride strideA, scalar_P * B, csipl_stride strideB, signed int * R, csipl_stride strideR, csipl_length n);	Computes the boolean comparison of 'equal', by element, of two vectors. The following instances are supported: csipl_vleq_f csipl_vleq_i
void csipl_cvleq_inter_P(void * A, csipl_stride strideA, void * B, csipl_stride strideB, signed int * R, csipl_stride strideR, csipl_length n);	Computes the boolean comparison of 'equal', by element, of two vectors. The following instances are supported: csipl_cvleq_inter_f csipl_cvleq_inter_i
void csipl_cvleq_split_P(scalar_P * A_re, scalar_P * A_im, csipl_stride strideA, scalar_P * B_re, scalar_P * B_im, csipl_stride strideB, signed int * R_re, signed int * R_im, csipl_stride strideR, csipl_length n);	Computes the boolean comparison of 'equal', by element, of two vectors. The following instances are supported: csipl_cvleq_split_f csipl_cvleq_split_i
void csipl_mleq_P(scalar_P * A, int ldA, scalar_P * B, int ldB, signed int * R, csipl_stride strideR, csipl_length m, csipl_length n);	Computes the boolean comparison of 'equal', by element, of two vectors/matrices. The following instances are supported: csipl_mleq_f csipl_mleq_i

Prototype	Description
void csipl_cmleq_inter_P(void * A, int ldA, void * B, int ldB, signed int * R, csipl_stride strideR, csipl_length m, csipl_length n);	Computes the boolean comparison of 'equal', by element, of two vectors/matrices. The following instances are supported: csipl_cmleq_inter_f csipl_cmleq_inter_i
void csipl_cmleq_split_P(scalar_P * A_re, scalar_P * A_im, int ldA, scalar_P * B_re, scalar_P * B_im, int ldB, signed int * R_re, signed int * R_im, csipl_stride strideR, csipl_length m, csipl_length n);	Computes the boolean comparison of 'equal', by element, of two vectors/matrices. The following instances are supported: csipl_cmleq_split_f csipl_cmleq_split_i
void csipl_vlge_P(scalar_P * A, csipl_stride strideA, scalar_P * B, csipl_stride strideB, signed int * R, csipl_stride strideR, csipl_length n);	Computes the boolean comparison of 'greater than or equal', by element, of two vectors. The following instances are supported: csipl_vlge_f csipl_vlge_i
void csipl_mlge_P(scalar_P * A, int ldA, scalar_P * B, int ldB, signed int * R, csipl_stride strideR, csipl_length m, csipl_length n);	Computes the boolean comparison of 'greater than or equal', by element, of two vectors/matrices. The following instances are supported: csipl_mlge_f csipl_mlge_i
void csipl_vlgt_P(scalar_P * A, csipl_stride strideA, scalar_P * B, csipl_stride strideB, signed int * R, csipl_stride strideR, csipl_length n);	Computes the boolean comparison of 'greater than', by element, of two vectors. The following instances are supported: csipl_vlgt_f csipl_vlgt_i
void csipl_mlgt_P(scalar_P * A, int ldA, scalar_P * B, int ldB, signed int * R, csipl_stride strideR, csipl_length m, csipl_length n);	Computes the boolean comparison of 'greater than', by element, of two vectors/matrices. The following instances are supported: csipl_mlgt_f csipl_mlgt_i

Prototype	Description
void csipl_vlle_P(scalar_P * A, csipl_stride strideA, scalar_P * B, csipl_stride strideB, signed int * R, csipl_stride strideR, csipl_length n);	Computes the boolean comparison of 'less than or equal', by element, of two vectors. The following instances are supported: csipl_vlle_f csipl_vlle_i
void csipl_mlle_P(scalar_P * A, int ldA, scalar_P * B, int ldB, signed int * R, csipl_stride strideR, csipl_length m, csipl_length n);	Computes the boolean comparison of 'less than or equal', by element, of two vectors/matrices. The following instances are supported: csipl_mlle_f csipl_mlle_i
void csipl_vllt_P(scalar_P * A, csipl_stride strideA, scalar_P * B, csipl_stride strideB, signed int * R, csipl_stride strideR, csipl_length n);	Computes the boolean comparison of 'less than', by element, of two vectors. The following instances are supported: csipl_vllt_f csipl_vllt_i
void csipl_mllt_P(scalar_P * A, int ldA, scalar_P * B, int ldB, signed int * R, csipl_stride strideR, csipl_length m, csipl_length n);	Computes the boolean comparison of 'less than', by element, of two vectors/matrices. The following instances are supported: csipl_mllt_f csipl_mllt_i
void csipl_vlne_P(scalar_P * A, csipl_stride strideA, scalar_P * B, csipl_stride strideB, signed int * R, csipl_stride strideR, csipl_length n);	Computes the boolean comparison of 'not equal', by element, of two vectors. The following instances are supported: csipl_vlne_f csipl_vlne_i
void csipl_cvlne_inter_P(void * A, csipl_stride strideA, void * B, csipl_stride strideB, signed int * R, csipl_stride strideR, csipl_length n);	Computes the boolean comparison of 'not equal', by element, of two vectors. The following instances are supported: csipl_cvlne_inter_f csipl_cvlne_inter_i

Prototype	Description
void csipl_cvlne_split_P(scalar_P * A_re, scalar_P * A_im, csipl_stride strideA, scalar_P * B_re, scalar_P * B_im, csipl_stride strideB, signed int * R_re, signed int * R_im, csipl_stride strideR, csipl_length n);	Computes the boolean comparison of 'not equal', by element, of two vectors. The following instances are supported: csipl_cvlne_split_f csipl_cvlne_split_i
void csipl_mlne_P(scalar_P * A, int ldA, scalar_P * B, int ldB, signed int * R, csipl_stride strideR, csipl_length m, csipl_length n);	Computes the boolean comparison of 'not equal', by element, of two vectors/matrices. The following instances are supported: csipl_mlne_f csipl_mlne_i
void csipl_cmlne_inter_P(void * A, int ldA, void * B, int ldB, signed int * R, csipl_stride strideR, csipl_length m, csipl_length n);	Computes the boolean comparison of 'not equal', by element, of two vectors/matrices. The following instances are supported: csipl_cmlne_inter_f csipl_cmlne_inter_i
void csipl_cmlne_split_P(scalar_P * A_re, scalar_P * A_im, int ldA, scalar_P * B_re, scalar_P * B_im, int ldB, signed int * R_re, signed int * R_im, csipl_stride strideR, csipl_length m, csipl_length n);	Computes the boolean comparison of 'not equal', by element, of two vectors/matrices. The following instances are supported: csipl_cmlne_split_f csipl_cmlne_split_i

6.6 Selection Operations

Prototype	Description
void csipl_vclip_P(scalar_P * A, csipl_stride strideA, scalar_P t1, scalar_P t2, scalar_P c1, scalar_P c2, scalar_P * R, csipl_stride strideR, csipl_length n);	Computes the generalised double clip, by element, of two vectors. The following instances are supported: csipl_vclip_f csipl_vclip_i csipl_vclip_si

Prototype	Description
void csipl_vinvclip_P(scalar_P * A, csipl_stride strideA, scalar_P t1, scalar_P t2, scalar_P t3, scalar_P c1, scalar_P c2, scalar_P * R, csipl_stride strideR, csipl_length n);	Computes the generalised inverted double clip, by element, of two vectors. The following instances are supported: csipl_vinvclip_f csipl_vinvclip_i csipl_vinvclip_si
unsigned int csipl_vindexbool(signed int * X, csipl_stride strideX, unsigned int * Y, csipl_stride strideY, csipl_length n);	Computes an index vector of the indices of the non-false elements of the boolean vector, and returns the number of non-false elements.
void csipl_vmax_f(float * A, csipl_stride strideA, float * B, csipl_stride strideB, float * R, csipl_stride strideR, csipl_length n);	Computes the maximum, by element, of two vectors.
void csipl_vmaxmg_f(float * A, csipl_stride strideA, float * B, csipl_stride strideB, float * R, csipl_stride strideR, csipl_length n);	Computes the maximum magnitude (absolute value), by element, of two vectors.
void csipl_vcmaxmgsq_inter_f(void * A, csipl_stride strideA, void * B, csipl_stride strideB, float * R, csipl_stride strideR, csipl_length n);	Computes the maximum magnitude squared, by element, of two complex vectors.
void csipl_vcmaxmgsq_split_f(float * A_re, float * A_im, csipl_stride strideA, float * B_re, float * B_im, csipl_stride strideB, float * R, csipl_stride strideR, csipl_length n);	Computes the maximum magnitude squared, by element, of two complex vectors.
float csipl_vcmaxmgsqval_inter_f(void * A, csipl_stride strideA, csipl_index * index, csipl_length n);	Returns the index and value of the maximum magnitude squared of the elements of a complex vector. The index is returned by reference as one of the arguments.

Prototype	Description
float csipl_vcmaxmgsqval_split_f(float * A_re, float * A_im, csipl_stride strideA, csipl_index * index, csipl_length n);	Returns the index and value of the maximum magnitude squared of the elements of a complex vector. The index is returned by reference as one of the arguments.
float csipl_vmaxmgval_f(float * A, csipl_stride strideA, csipl_index * index, csipl_length n);	Returns the index and value of the maximum absolute value of the elements of a vector. The index is returned by reference as one of the arguments.
float csipl_vmaxval_f(float * A, csipl_stride strideA, csipl_index * index, csipl_length n);	Returns the index and value of the maximum value of the elements of a vector. The index is returned by reference as one of the arguments.
void csipl_vmin_f(float * A, csipl_stride strideA, float * B, csipl_stride strideB, float * R, csipl_stride strideR, csipl_length n);	Computes the minimum, by element, of two vectors.
void csipl_vminmg_f(float * A, csipl_stride strideA, float * B, csipl_stride strideB, float * R, csipl_stride strideR, csipl_length n);	Computes the minimum magnitude (absolute value), by element, of two vectors.
void csipl_vcminmgsq_inter_f(void * A, csipl_stride strideA, void * B, csipl_stride strideB, float * R, csipl_stride strideR, csipl_length n);	Computes the minimum magnitude squared, by element, of two complex vectors.
void csipl_vcminmgsq_split_f(float * A_re, float * A_im, csipl_stride strideA, float * B_re, float * B_im, csipl_stride strideB, float * R, csipl_stride strideR, csipl_length n);	Computes the minimum magnitude squared, by element, of two complex vectors.

Prototype	Description
float csipl_vcminmgsqval_inter_f(void * A, csipl_stride strideA, csipl_index * index, csipl_length n);	Returns the index and value of the minimum magnitude squared of the elements of a complex vector. The index is returned by reference as one of the arguments.
float csipl_vcminmgsqval_split_f(float * A_re, float * A_im, csipl_stride strideA, csipl_index * index, csipl_length n);	Returns the index and value of the minimum magnitude squared of the elements of a complex vector. The index is returned by reference as one of the arguments.
float csipl_vminmgval_f(float * A, csipl_stride strideA, csipl_index * index, csipl_length n);	Returns the index and value of the minimum absolute value of the elements of a vector. The index is returned by reference as one of the arguments.
float csipl_vminval_f(float * A, csipl_stride strideA, csipl_index * index, csipl_length n);	Returns the index and value of the minimum value of the elements of a vector. The index is returned by reference as one of the arguments.

6.7 Bitwise and Boolean Logical Operators

Prototype	Description
void csipl_vand_P(scalar_P * A, csipl_stride strideA, scalar_P * B, csipl_stride strideB, scalar_P * R, csipl_stride strideR, csipl_length n);	Computes the bitwise and, by element, of two vectors. The following instances are supported: csipl_vand_i csipl_vand_si csipl_vand_bl
void csipl_mand_P(scalar_P * A, int ldA, scalar_P * B, int ldB, scalar_P * R, int ldR, csipl_length m, csipl_length n);	Computes the bitwise and, by element, of two matrices. The following instances are supported: csipl_mand_i csipl_mand_si csipl_mand_bl
void csipl_vnot_P(scalar_P * A, csipl_stride strideA, scalar_P * R, csipl_stride strideR, csipl_length n);	Computes the bitwise not (one's complement), by element, of two vectors. The following instances are supported: csipl_vnot_i csipl_vnot_si csipl_vnot_bl

Prototype	Description
void csipl_mnot_P(scalar_P * A, int ldA, scalar_P * R, int ldR, csipl_length m, csipl_length n);	Computes the bitwise not (one's complement), by element, of two matrices. The following instances are supported: csipl_mnot_i csipl_mnot_si csipl_mnot_bl
void csipl_vor_P(scalar_P * A, csipl_stride strideA, scalar_P * B, csipl_stride strideB, scalar_P * R, csipl_stride strideR, csipl_length n);	Computes the bitwise inclusive or, by element, of two vectors. The following instances are supported: csipl_vor_i csipl_vor_si csipl_vor_bl
void csipl_mor_P(scalar_P * A, int ldA, scalar_P * B, int ldB, scalar_P * R, int ldR, csipl_length m, csipl_length n);	Computes the bitwise inclusive or, by element, of two matrices. The following instances are supported: csipl_mor_i csipl_mor_si csipl_mor_bl
void csipl_mxor_P(scalar_P * A, int ldA, scalar_P * B, int ldB, scalar_P * R, int ldR, csipl_length m, csipl_length n);	Computes the bitwise exclusive or, by element, of two matrices. The following instances are supported: csipl_mxor_i csipl_mxor_si csipl_mxor_bl

6.8 Element Generation and Copy

Prototype	Description
void csipl_vcopy_P_P(scalar_P * A, csipl_stride strideA, scalar_P * R, csipl_stride strideR, csipl_length n);	Copy the source vector to the destination vector performing any necessary type conversion of the standard ANSI C scalar types. The following instances are supported: csipl_vcopy_f_f csipl_vcopy_f_i csipl_vcopy_f_si csipl_vcopy_f_bl csipl_vcopy_i_f csipl_vcopy_i_i csipl_vcopy_i_si csipl_vcopy_i_vi csipl_vcopy_si_f csipl_vcopy_si_i csipl_vcopy_si_si csipl_vcopy_bl_f csipl_vcopy_bl_bl csipl_vcopy_vi_i csipl_vcopy_vi_vi csipl_vcopy_mi_mi
void csipl_cvcopy_inter_f_f(void * A, csipl_stride strideA, void * R, csipl_stride strideR, csipl_length n);	Copy the source vector to the destination vector performing any necessary type conversion of the standard ANSI C scalar types.
void csipl_cvcopy_split_f_f(float * A_re, float * A_im, csipl_stride strideA, float * R_re, float * R_im, csipl_stride strideR, csipl_length n);	Copy the source vector to the destination vector performing any necessary type conversion of the standard ANSI C scalar types.
void csipl_mcopy_P_P(scalar_P * A, int ldA, scalar_P * R, int ldR, csipl_length m, csipl_length n);	Copy the source matrix to the destination matrix performing any necessary type conversion of the standard ANSI C scalar types. The following instances are supported: csipl_mcopy_f_f csipl_mcopy_f_i csipl_mcopy_f_si csipl_mcopy_f_bl csipl_mcopy_i_f csipl_mcopy_i_i csipl_mcopy_i_si csipl_mcopy_si_f csipl_mcopy_si_i csipl_mcopy_si_si csipl_mcopy_bl_f csipl_mcopy_bl_bl

Prototype	Description
void csipl_cmcopy_inter_f_f(void * A, int ldA, void * R, int ldR, csipl_length m, csipl_length n);	Copy the source matrix to the destination matrix performing any necessary type conversion of the standard ANSI C scalar types.
void csipl_cmcopy_split_f_f(scalar_P * A_re, scalar_P * A_im, int ldA, scalar_P * R_re, scalar_P * R_im, int ldR, csipl_length m, csipl_length n);	Copy the source matrix to the destination matrix performing any necessary type conversion of the standard ANSI C scalar types.
void csipl_vfill_P(scalar_P a, scalar_P * R, csipl_stride strideR, csipl_length n);	Fill a vector with a constant value. The following instances are supported: csipl_vfill_f csipl_vfill_i csipl_vfill_si
void csipl_cvfill_inter_f(csipl_cscalar_f a, void * R, csipl_stride strideR, csipl_length n);	Fill a vector with a constant value.
void csipl_cvfill_split_f(csipl_cscalar_f a_re, csipl_cscalar_f a_im, float * R_re, float * R_im, csipl_stride strideR, csipl_length n);	Fill a vector with a constant value.
void csipl_mfill_P(scalar_P a, scalar_P * R, int ldR, csipl_length m, csipl_length n);	Fill a matrix with a constant value. The following instances are supported: csipl_mfill_f csipl_mfill_i csipl_mfill_si
void csipl_cmfill_inter_f(csipl_cscalar_f a, void * R, int ldR, csipl_length m, csipl_length n);	Fill a matrix with a constant value.
void csipl_cmfill_split_f(csipl_cscalar_f a_re, csipl_cscalar_f a_im, float * R_re, float * R_im, int ldR, csipl_length m, csipl_length n);	Fill a matrix with a constant value.

Prototype	Description
void csipl_vramp_P(scalar_P alpha, scalar_P beta, scalar_P * R, csipl_stride strideR, csipl_length n);	Computes a vector ramp by starting at an initial value and incrementing each successive element by the ramp step size. The following instances are supported: csipl_vramp_f csipl_vramp_i csipl_vramp_si

6.9 Manipulation Operations

Prototype	Description
void csipl_vcmplx_inter_f(float * A, csipl_stride strideA, float * B, csipl_stride strideB, void * R, csipl_stride strideR, csipl_length n);	Form a complex vector from two real vectors.
void csipl_vcmplx_split_f(float * A, csipl_stride strideA, float * B, csipl_stride strideB, float * R_re, float * R_im, csipl_stride strideR, csipl_length n);	Form a complex vector from two real vectors.
void csipl_mcmplx_f(float * A, int ldA, float * B, int ldB, void * R, int ldR, csipl_length m, csipl_length n);	Form a complex matrix from two real matrices.
void csipl_vgather_P(scalar_P * X, csipl_stride strideX, unsigned int * I, csipl_stride strideI, scalar_P * Y, csipl_stride strideY, csipl_length n);	The gather operation selects elements of a source vector using indices supplied by an index vector. The selected elements are placed sequentially in an output vector so that the output vector and the index vector are indexed the same. The following instances are supported: csipl_vgather_f csipl_vgather_i csipl_vgather_si

Prototype	Description
void csipl_cvgather_inter_f(void * X, csipl_stride strideX, unsigned int * I, csipl_stride strideI, void * Y, csipl_stride strideY, csipl_length n);	The gather operation selects elements of a source vector using indices supplied by an index vector. The selected elements are placed sequentially in an output vector so that the output vector and the index vector are indexed the same.
void csipl_cvgather_split_f(float * X_re, float * X_im, csipl_stride strideX, unsigned int * I_re, unsigned int * I_im, csipl_stride strideI, float * Y_re, float * Y_im, csipl_stride strideY, csipl_length n);	The gather operation selects elements of a source vector using indices supplied by an index vector. The selected elements are placed sequentially in an output vector so that the output vector and the index vector are indexed the same.
void csipl_mgather_P(scalar_P * X, int ldX, csipl_scalar_mi * I, csipl_stride strideI, scalar_P * Y, csipl_stride strideY, csipl_length n);	The gather operation selects elements of a source vector/matrix using indices supplied by an index vector. The selected elements are placed sequentially in an output vector so that the output vector and the index vector are indexed the same. The following instances are supported: csipl_mgather_f csipl_mgather_i csipl_mgather_si
void csipl_cmgather_inter_f(void * X, int ldX, csipl_scalar_mi * I, csipl_stride strideI, void * Y, csipl_stride strideY, csipl_length n);	The gather operation selects elements of a source vector/matrix using indices supplied by an index vector. The selected elements are placed sequentially in an output vector so that the output vector and the index vector are indexed the same.
void csipl_cmgather_split_f(float * X_re, float * X_im, int ldX, csipl_scalar_mi * I_re, csipl_scalar_mi * I_im, csipl_stride strideI, float * Y_re, float * Y_im, csipl_stride strideY, csipl_length n);	The gather operation selects elements of a source vector/matrix using indices supplied by an index vector. The selected elements are placed sequentially in an output vector so that the output vector and the index vector are indexed the same.
void csipl_vimag_inter_f(void * A, csipl_stride strideA, float * R, csipl_stride strideR, csipl_length n);	Extract the imaginary part of a complex vector.

Prototype	Description
void csipl_vimag_split_f(float * A_re, float * A_im, csipl_stride strideA, float * R, csipl_stride strideR, csipl_length n);	Extract the imaginary part of a complex vector.
void csipl_mimag_f(void * A, int ldA, float * R, int ldR, csipl_length m, csipl_length n);	Extract the imaginary part of a complex matrix.
void csipl_vpolar_inter_f(void * A, csipl_stride strideA, float * R, csipl_stride strideR, float * P, csipl_stride strideP, csipl_length n);	Convert a complex vector from rectangular to polar form. The polar data consists of a real vector containing the radius and a corresponding real vector containing the argument (angle) of the complex input data.
void csipl_vpolar_split_f(float * A_re, float * A_im, csipl_stride strideA, float * R, csipl_stride strideR, float * P, csipl_stride strideP, csipl_length n);	Convert a complex vector from rectangular to polar form. The polar data consists of a real vector containing the radius and a corresponding real vector containing the argument (angle) of the complex input data.
void csipl_mpolar_f(void * A, int ldA, float * R, int ldR, float * P, int ldP, csipl_length m, csipl_length n);	Convert a complex matrix from rectangular to polar form. The polar data consists of a real matrix containing the radius and a corresponding real matrix containing the argument (angle) of the complex input data.
void csipl_vreal_inter_f(void * A, csipl_stride strideA, float * R, csipl_stride strideR, csipl_length n);	Extract the real part of a complex vector.
void csipl_vreal_split_f(float * A_re, float * A_im, csipl_stride strideA, float * R, csipl_stride strideR, csipl_length n);	Extract the real part of a complex vector.

Prototype	Description
void csipl_mreal_f(void * A, int ldA, float * R, int ldR, csipl_length m, csipl_length n);	Extract the real part of a complex matrix.
void csipl_vrect_inter_f(float * R, csipl_stride strideR, float * P, csipl_stride strideP, void * A, csipl_stride strideA, csipl_length n);	Convert a pair of real vectors from complex polar to complex rectangular form.
void csipl_vrect_split_f(float * R, csipl_stride strideR, float * P, csipl_stride strideP, float * A_re, float * A_im, csipl_stride strideA, csipl_length n);	Convert a pair of real vectors from complex polar to complex rectangular form.
void csipl_mrect_f(float * R, int ldR, float * P, int ldP, void * A, int ldA, csipl_length m, csipl_length n);	Convert a pair of real matrices from complex polar to complex rectangular form.
void csipl_vscatter_P(scalar_P * X, csipl_stride strideX, scalar_P * Y, csipl_stride strideY, unsigned int * I, csipl_stride strideI, csipl_length n);	The scatter operation sequentially uses elements of a source vector and an index vector. The element of the vector index is used to select a storage location in the output vector to store the element from the source vector. The following instances are supported: csipl_vscatter_f csipl_vscatter_i csipl_vscatter_si
void csipl_cvscatter_inter_f(void * X, csipl_stride strideX, void * Y, csipl_stride strideY, unsigned int * I, csipl_stride strideI, csipl_length n);	The scatter operation sequentially uses elements of a source vector and an index vector. The element of the vector index is used to select a storage location in the output vector to store the element from the source vector.

Prototype	Description
void csipl_cvscatter_split_f(float * X_re, float * X_im, csipl_stride strideX, float * Y_re, float * Y_im, csipl_stride strideY, unsigned int * I_re, unsigned int * I_im, csipl_stride strideI, csipl_length n);	The scatter operation sequentially uses elements of a source vector and an index vector. The element of the vector index is used to select a storage location in the output vector to store the element from the source vector.
void csipl_mscatter_P(scalar_P * X, csipl_stride strideX, scalar_P * Y, int ldY, csipl_scalar_mi * I, csipl_stride strideI, csipl_length n);	The scatter operation sequentially uses elements of a source vector and an index vector. The element of the vector/matrix index is used to select a storage location in the output vector/matrix to store the element from the source vector. The following instances are supported: csipl_mscatter_f csipl_mscatter_i csipl_mscatter_si
void csipl_cmscatter_inter_f(void * X, csipl_stride strideX, void * Y, int ldY, csipl_scalar_mi * I, csipl_stride strideI, csipl_length n);	The scatter operation sequentially uses elements of a source vector and an index vector. The element of the vector/matrix index is used to select a storage location in the output vector/matrix to store the element from the source vector.
void csipl_cmscatter_split_f(float * X_re, float * X_im, csipl_stride strideX, float * Y_re, float * Y_im, int ldY, csipl_scalar_mi * I_re, csipl_scalar_mi * I_im, csipl_stride strideI, csipl_length n);	The scatter operation sequentially uses elements of a source vector and an index vector. The element of the vector/matrix index is used to select a storage location in the output vector/matrix to store the element from the source vector.
void csipl_cvswap_inter_f(void * A, csipl_stride strideA, void * B, csipl_stride strideB, csipl_length n);	Swap elements between two vectors.
void csipl_cvswap_split_f(float * A_re, float * A_im, csipl_stride strideA, float * B_re, float * B_im, csipl_stride strideB, csipl_length n);	Swap elements between two vectors.

Prototype	Description
void csipl_mswap_P(scalar_P * A, int ldA, scalar_P * B, int ldB, csipl_length m, csipl_length n);	Swap elements between two matrices. The following instances are supported: csipl_mswap_f csipl_mswap_i csipl_mswap_si
void csipl_cmswap_inter_f(void * A, int ldA, void * B, int ldB, csipl_length m, csipl_length n);	Swap elements between two matrices.
void csipl_cmswap_split_f(float * A_re, float * A_im, int ldA, float * B_re, float * B_im, int ldB, csipl_length m, csipl_length n);	Swap elements between two matrices.

Chapter 7. Signal Processing Functions

7.1 FFT Functions

Prototype	Description

`csipl_fft_f *`
`csipl_ccfftip_create_f(`
`csipl_index length,`
`float scale,`
`csipl_fft_dir dir,`
`unsigned int ntimes,`
`csipl_alg_hint hint);`
	Create a 1D FFT object.

`csipl_fft_f *`
`csipl_ccfftop_create_f(`
`csipl_index length,`
`float scale,`
`csipl_fft_dir dir,`
`unsigned int ntimes,`
`csipl_alg_hint hint);`
	Create a 1D FFT object.

`csipl_fft_f *`
`csipl_crfftop_create_f(`
`csipl_index length,`
`float scale,`
`unsigned int ntimes,`
`csipl_alg_hint hint);`
	Create a 1D FFT object.

`csipl_fft_f *`
`csipl_rcfftop_create_f(`
`csipl_index length,`
`float scale,`
`unsigned int ntimes,`
`csipl_alg_hint hint);`
	Create a 1D FFT object.

`void`
`csipl_ccfftip_inter_f(`
`csipl_fft_f * plan,`
`void * xy,`
`csipl_stride stridexy);`
	Apply a complex-to-complex Fast Fourier Transform (FFT).

`void`
`csipl_ccfftip_split_f(`
`csipl_fft_f * plan,`
`float * xy_re,`
`float * xy_im,`
`csipl_stride stridexy);`
	Apply a complex-to-complex Fast Fourier Transform (FFT).

`void`
`csipl_ccfftop_inter_f(`
`csipl_fft_f * plan,`
`void * x,`
`csipl_stride stridex,`
`void * y,`
`csipl_stride stridey);`
	Apply a complex-to-complex Fast Fourier Transform (FFT).

`void`
`csipl_ccfftop_split_f(`
`csipl_fft_f * plan,`
`float * x_re,`
`float * x_im,`
`csipl_stride stridex,`
`float * y_re,`
`float * y_im,`
`csipl_stride stridey);`
	Apply a complex-to-complex Fast Fourier Transform (FFT).

Prototype	Description
void csipl_crfftop_inter_f(csipl_fft_f * plan, void * x, csipl_stride stridex, float * y, csipl_stride stridey);	Apply a complex-to-real Fast Fourier Transform (FFT).
void csipl_crfftop_split_f(csipl_fft_f * plan, float * x_re, float * x_im, csipl_stride stridex, float * y, csipl_stride stridey);	Apply a complex-to-real Fast Fourier Transform (FFT).
void csipl_rcfftop_inter_f(csipl_fft_f * plan, float * x, csipl_stride stridex, void * y, csipl_stride stridey);	Apply a real-to-complex Fast Fourier Transform (FFT).
void csipl_rcfftop_split_f(csipl_fft_f * plan, float * x, csipl_stride stridex, float * y_re, float * y_im, csipl_stride stridey);	Apply a real-to-complex Fast Fourier Transform (FFT).
int csipl_fft_destroy_f(csipl_fft_f * plan);	Destroy an FFT object.
void csipl_fft_getattr_f(csipl_fft_f * plan, csipl_fft_attr_f * attr);	Return the attributes of an FFT object.
*csipl_fftm_f ** csipl_ccfftmop_create_f(csipl_index rows, csipl_index cols, float scale, csipl_fft_dir dir, csipl_major major, unsigned int ntimes, csipl_alg_hint hint);	Create a 1D multiple FFT object.
*csipl_fftm_f ** csipl_ccfftmip_create_f(csipl_index rows, csipl_index cols, float scale, csipl_fft_dir dir, csipl_major major, unsigned int ntimes, csipl_alg_hint hint);	Create a 1D multiple FFT object.

Prototype	Description
`csipl_fftm_f * csipl_crfftmop_create_f( csipl_index rows, csipl_index cols, float scale, csipl_major major, unsigned int ntimes, csipl_alg_hint hint);`	Create a 1D multiple FFT object.
`csipl_fftm_f * csipl_rcfftmop_create_f( csipl_index rows, csipl_index cols, float scale, csipl_major major, unsigned int ntimes, csipl_alg_hint hint);`	Create a 1D multiple FFT object.
`int csipl_fftm_destroy_f( csipl_fftm_f * plan);`	Destroy an FFT object.
`void csipl_fftm_getattr_f( csipl_fftm_f * plan, csipl_fftm_attr_f * attr);`	Return the attributes of an FFT object.
`void csipl_ccfftmip_inter_f( csipl_fftm_f * plan, void * XY, int ldXY);`	Apply a multiple complex-to-complex Fast Fourier Transform (FFT).
`void csipl_ccfftmip_split_f( csipl_fftm_f * plan, float * XY_re, float * XY_im, int ldXY);`	Apply a multiple complex-to-complex Fast Fourier Transform (FFT).
`void csipl_ccfftmop_inter_f( csipl_fftm_f * plan, void * X, int ldX, void * Y, int ldY);`	Apply a multiple complex-to-complex Fast Fourier Transform (FFT).
`void csipl_ccfftmop_split_f( csipl_fftm_f * plan, float * X_re, float * X_im, int ldX, float * Y_re, float * Y_im, int ldY);`	Apply a multiple complex-to-complex Fast Fourier Transform (FFT).
`void csipl_crfftmop_inter_f( csipl_fftm_f * plan, void * X, int ldX, float * Y, int ldY);`	Apply a multiple complex-to-real Fast Fourier Transform (FFT).

Prototype	Description
void csipl_crfftmop_split_f(csipl_fftm_f * plan, float * X_re, float * X_im, int ldX, float * Y, int ldY);	Apply a multiple complex-to-real Fast Fourier Transform (FFT).
void csipl_rcfftmop_inter_f(csipl_fftm_f * plan, float * X, int ldX, void * Y, int ldY);	Apply a multiple real-to-complex out of place Fast Fourier Transform (FFT).
void csipl_rcfftmop_split_f(csipl_fftm_f * plan, float * X, int ldX, float * Y_re, float * Y_im, int ldY);	Apply a multiple real-to-complex out of place Fast Fourier Transform (FFT).
csipl_fft2d_f * csipl_ccfft2dop_create_f(csipl_index rows, csipl_index cols, float scale, csipl_fft_dir dir, unsigned int ntimes, csipl_alg_hint hint);	Create a 2D FFT object.
csipl_fft2d_f * csipl_ccfft2dip_create_f(csipl_index rows, csipl_index cols, float scale, csipl_fft_dir dir, unsigned int ntimes, csipl_alg_hint hint);	Create a 2D FFT object.
csipl_fft2d_f * csipl_crfft2dop_create_f(csipl_index rows, csipl_index cols, float scale, unsigned int ntimes, csipl_alg_hint hint);	Create a 2D FFT object.
csipl_fft2d_f * csipl_rcfft2dop_create_f(csipl_index rows, csipl_index cols, float scale, unsigned int ntimes, csipl_alg_hint hint);	Create a 2D FFT object.
void csipl_ccfft2dip_inter_f(csipl_fft2d_f * plan, void * XY, int ldXY);	Apply a complex-to-complex 2D Fast Fourier Transform (FFT).

Prototype	Description
void csipl_ccfft2dip_split_f(csipl_fft2d_f * plan, float * XY_re, float * XY_im, int ldXY);	Apply a complex-to-complex 2D Fast Fourier Transform (FFT).
void csipl_ccfft2dop_inter_f(csipl_fft2d_f * plan, void * X, int ldX, void * Y, int ldY);	Apply a complex-to-complex 2D Fast Fourier Transform (FFT).
void csipl_ccfft2dop_split_f(csipl_fft2d_f * plan, float * X_re, float * X_im, int ldX, float * Y_re, float * Y_im, int ldY);	Apply a complex-to-complex 2D Fast Fourier Transform (FFT).
void csipl_crfft2dop_inter_f(csipl_fft2d_f * plan, void * X, int ldX, float * Y, int ldY);	Apply a complex-to-real 2D Fast Fourier Transform (FFT).
void csipl_crfft2dop_split_f(csipl_fft2d_f * plan, float * X_re, float * X_im, int ldX, float * Y, int ldY);	Apply a complex-to-real 2D Fast Fourier Transform (FFT).
void csipl_rcfft2dop_inter_f(csipl_fft2d_f * plan, float * X, int ldX, void * Y, int ldY);	Apply a real-to-complex 2D Fast Fourier Transform (FFT).
void csipl_rcfft2dop_split_f(csipl_fft2d_f * plan, float * X, int ldX, float * Y_re, float * Y_im, int ldY);	Apply a real-to-complex 2D Fast Fourier Transform (FFT).
int csipl_fft2d_destroy_f(csipl_fft2d_f * plan);	Destroy an FFT object.
void csipl_fft2d_getattr_f(csipl_fft2d_f * plan, csipl_fft2d_attr_f * attr);	Return the attributes of an FFT object.

7.2 Convolution/Correlation Functions

Prototype	Description
csipl_conv1d_f * csipl_conv1d_create_f(float * kernel, csipl_stride stridekernel, csipl_symmetry symm, unsigned int N, unsigned int D, csipl_support_region support, unsigned int ntimes, csipl_alg_hint hint, csipl_length n);	Create a decimated 1D convolution filter object.
int csipl_conv1d_destroy_f(csipl_conv1d_f * plan);	Destroy a 1D convolution object.
void csipl_conv1d_getattr_f(csipl_conv1d_f * plan, csipl_conv1d_attr_f * attr);	Returns the attributes for a 1D convolution object.
void csipl_convolve1d_f(csipl_conv1d_f * plan, float * x, csipl_stride stridex, float * y, csipl_stride stridey, csipl_length n);	Compute a decimated real one-dimensional (1D) convolution of two vectors.
csipl_conv2d_f * csipl_conv2d_create_f(float * H, int ldH, csipl_symmetry symm, unsigned int P, unsigned int Q, unsigned int D, csipl_support_region support, unsigned int ntimes, csipl_alg_hint hint, csipl_length m, csipl_length n);	Create a decimated 2D convolution filter object.
int csipl_conv2d_destroy_f(csipl_conv2d_f * plan);	Destroy a 2D convolution object.
void csipl_conv2d_getattr_f(csipl_conv2d_f * plan, csipl_conv2d_attr_f * attr);	Returns the attributes for a 2D convolution object.
void csipl_convolve2d_f(csipl_conv2d_f * plan, float * x, int ldx, float * y, int ldy, csipl_length m, csipl_length n);	Compute a decimated real two-dimensional (2D) convolution of two matrices.
csipl_Dcorr1d_P * csipl_Dcorr1d_create_P(unsigned int M, unsigned int N, csipl_support_region support, unsigned int ntimes, csipl_alg_hint hint);	Create a 1D correlation object. The following instances are supported: csipl_corr1d_create_f csipl_ccorr1d_create_f

Prototype	Description
int csipl_Dcorr1d_destroy_P(csipl_Dcorr1d_P * plan);	Destroy a 1D correlation object. The following instances are supported: csipl_corr1d_destroy_f csipl_ccorr1d_destroy_f
void csipl_Dcorr1d_getattr_P(csipl_Dcorr1d_P * plan, csipl_Dcorr1d_attr_P * attr);	Return the attributes for a 1D correlation object. The following instances are supported: csipl_corr1d_getattr_f csipl_ccorr1d_getattr_f
void csipl_correlate1d_f(csipl_corr1d_f * plan, csipl_bias bias, float * ref, csipl_stride strideref, float * x, csipl_stride stridex, float * y, csipl_stride stridey, csipl_length n);	Compute a real one-dimensional (1D) correlation of two vectors.
void csipl_ccorrelate1d_inter_f(csipl_ccorr1d_f * plan, csipl_bias bias, void * ref, csipl_stride strideref, void * x, csipl_stride stridex, void * y, csipl_stride stridey, csipl_length n);	Compute a real one-dimensional (1D) correlation of two vectors.
void csipl_ccorrelate1d_split_f(csipl_ccorr1d_f * plan, csipl_bias bias, float * ref_re, float * ref_im, csipl_stride strideref, float * x_re, float * x_im, csipl_stride stridex, float * y_re, float * y_im, csipl_stride stridey, csipl_length n);	Compute a real one-dimensional (1D) correlation of two vectors.
csipl_Dcorr2d_P * csipl_Dcorr2d_create_P(unsigned int M, unsigned int N, unsigned int P, unsigned int Q, csipl_support_region support, unsigned int ntimes, csipl_alg_hint hint);	Create a 2D correlation object. The following instances are supported: csipl_corr2d_create_f csipl_ccorr2d_create_f
int csipl_Dcorr2d_destroy_P(csipl_Dcorr2d_P * plan);	Destroy a 2D correlation object. The following instances are supported: csipl_corr2d_destroy_f csipl_ccorr2d_destroy_f

Prototype	Description
void csipl_Dcorr2d_getattr_P(csipl_Dcorr2d_P * plan, csipl_Dcorr2d_attr_P * attr);	Return the attributes for a 2D correlation object. The following instances are supported: csipl_corr2d_getattr_f csipl_ccorr2d_getattr_f
void csipl_correlate2d_f(csipl_corr2d_f * plan, csipl_bias bias, float * ref, int ldref, float * x, int ldx, float * y, int ldy, csipl_length m, csipl_length n);	Compute a two-dimensional (2D) correlation of two matrices.
void csipl_ccorrelate2d_inter_f(csipl_ccorr2d_f * plan, csipl_bias bias, void * ref, int ldref, void * x, int ldx, void * y, int ldy, csipl_length m, csipl_length n);	Compute a two-dimensional (2D) correlation of two matrices.
void csipl_ccorrelate2d_split_f(csipl_ccorr2d_f * plan, csipl_bias bias, float * ref_re, float * ref_im, int ldref, float * x_re, float * x_im, int ldx, float * y_re, float * y_im, int ldy, csipl_length m, csipl_length n);	Compute a two-dimensional (2D) correlation of two matrices.

7.3 Window Functions

Prototype	Description
float * csipl_vcreate_blackman_f(unsigned int N, csipl_memory_hint hint);	Create a vector with Blackman window weights.
float * csipl_vcreate_cheby_f(unsigned int N, float ripple, csipl_memory_hint hint);	Create a vector with Dolph-Chebyshev window weights.

Prototype	Description
`float * csipl_vcreate_hanning_f( unsigned int N, csipl_memory_hint hint);`	Create a vector with Hanning window weights.
`float * csipl_vcreate_kaiser_f( unsigned int N, float beta, csipl_memory_hint hint);`	Create a vector with Kaiser window weights.

7.4 Filter Functions

Prototype	Description
`csipl_fir_f * csipl_fir_create_f( float * kernel, csipl_stride stridekernel, csipl_symmetry symm, unsigned int N, unsigned int D, csipl_obj_state state, unsigned int ntimes, csipl_alg_hint hint, csipl_length n);`	Create a decimated FIR filter object.
`csipl_cfir_f * csipl_cfir_create_inter_f( void * kernel, csipl_stride stridekernel, csipl_symmetry symm, unsigned int N, unsigned int D, csipl_obj_state state, unsigned int ntimes, csipl_alg_hint hint, csipl_length n);`	Create a decimated FIR filter object.
`csipl_cfir_f * csipl_cfir_create_split_f( float * kernel_re, float * kernel_im, csipl_stride stridekernel, csipl_symmetry symm, unsigned int N, unsigned int D, csipl_obj_state state, unsigned int ntimes, csipl_alg_hint hint, csipl_length n);`	Create a decimated FIR filter object.

`int csipl_Dfir_destroy_P( csipl_Dfir_P * plan);`	Destroy a FIR filter object.
	The following instances are supported:
	`csipl_fir_destroy_f`
	`csipl_cfir_destroy_f`

`int csipl_firflt_f( csipl_fir_f * plan, float * x, csipl_stride stridex, float * y, csipl_stride stridey, csipl_length n);`	FIR filter an input sequence and decimate the output.

Prototype	Description
int csipl_cfirflt_inter_f(csipl_cfir_f * plan, void * x, csipl_stride stridex, void * y, csipl_stride stridey, csipl_length n);	FIR filter an input sequence and decimate the output.
int csipl_cfirflt_split_f(csipl_cfir_f * plan, float * x_re, float * x_im, csipl_stride stridex, float * y_re, float * y_im, csipl_stride stridey, csipl_length n);	FIR filter an input sequence and decimate the output.
void csipl_Dfir_getattr_P(csipl_Dfir_P * plan, csipl_Dfir_attr_P * attr);	Return the attributes of a FIR filter object. The following instances are supported: csipl_fir_getattr_f csipl_cfir_getattr_f
void csipl_Dfir_reset_P(csipl_Dfir_P * fir);	Reset the state of a decimated FIR filter object. The following instances are supported: csipl_fir_reset_f csipl_cfir_reset_f

7.5 Miscellaneous Signal Processing Functions

Prototype	Description
`void csipl_vhisto_f( float * A, csipl_stride strideA, float min, float max, csipl_hist_opt opt, float * R, csipl_stride strideR, csipl_length n);`	Compute the histogram of a vector.

Chapter 8. Linear Algebra

8.1 Matrix and Vector Operations

Prototype	Description
void csipl_cmherm_inter_f(void * A, int ldA, void * R, int ldR, csipl_length m, csipl_length n);	Complex Hermitian (conjugate transpose) of a matrix.
void csipl_cmherm_split_f(float * A_re, float * A_im, int ldA, float * R_re, float * R_im, int ldR, csipl_length m, csipl_length n);	Complex Hermitian (conjugate transpose) of a matrix.
csipl_cscalar_f csipl_cvjdot_inter_f(void * A, csipl_stride strideA, void * B, csipl_stride strideB, csipl_length n);	Compute the conjugate inner (dot) product of two complex vectors.
csipl_cscalar_f csipl_cvjdot_split_f(float * A_re, float * A_im, csipl_stride strideA, float * B_re, float * B_im, csipl_stride strideB, csipl_length n);	Compute the conjugate inner (dot) product of two complex vectors.
void csipl_gemp_f(float alpha, float * A, int ldA, csipl_mat_op Aop, float * B, int ldB, csipl_mat_op Bop, float beta, float * R, int ldR, csipl_length m, csipl_length n, csipl_length p);	Calculate the general product of two matrices and accumulate.

Prototype	Description
void csipl_cgemp_inter_f(csipl_cscalar_f alpha, void * A, int ldA, csipl_mat_op Aop, void * B, int ldB, csipl_mat_op Bop, csipl_cscalar_f beta, void * R, int ldR, csipl_length m, csipl_length n, csipl_length p);	Calculate the general product of two matrices and accumulate.
void csipl_cgemp_split_f(csipl_cscalar_f alpha_re, csipl_cscalar_f alpha_im, float * A_re, float * A_im, int ldA, csipl_mat_op Aop, float * B_re, float * B_im, int ldB, csipl_mat_op Bop, csipl_cscalar_f beta_re, csipl_cscalar_f beta_im, float * R_re, float * R_im, int ldR, csipl_length m, csipl_length n, csipl_length p);	Calculate the general product of two matrices and accumulate.
void csipl_gems_f(float alpha, float * A, int ldA, csipl_mat_op Aop, float beta, float * C, int ldC, csipl_length m, csipl_length n);	Calculate a general matrix sum.
void csipl_cgems_inter_f(csipl_cscalar_f alpha, void * A, int ldA, csipl_mat_op Aop, csipl_cscalar_f beta, void * C, int ldC, csipl_length m, csipl_length n);	Calculate a general matrix sum.

Prototype	Description
void csipl_cgems_split_f(csipl_cscalar_f alpha_re, csipl_cscalar_f alpha_im, float * A_re, float * A_im, int ldA, csipl_mat_op Aop, csipl_cscalar_f beta_re, csipl_cscalar_f beta_im, float * C_re, float * C_im, int ldC, csipl_length m, csipl_length n);	Calculate a general matrix sum.
void csipl_mprod_P(scalar_P * A, int ldA, scalar_P * B, int ldB, scalar_P * R, int ldR, csipl_length m, csipl_length n, csipl_length p);	Calculate the product of two matrices. The following instances are supported: csipl_mprod_f csipl_mprod_i csipl_mprod_si
void csipl_cmprod_inter_P(void * A, int ldA, void * B, int ldB, void * R, int ldR, csipl_length m, csipl_length n, csipl_length p);	Calculate the product of two matrices. The following instances are supported: csipl_cmprod_inter_f csipl_cmprod_inter_i csipl_cmprod_inter_si
void csipl_cmprod_split_P(scalar_P * A_re, scalar_P * A_im, int ldA, scalar_P * B_re, scalar_P * B_im, int ldB, scalar_P * R_re, scalar_P * R_im, int ldR, csipl_length m, csipl_length n, csipl_length p);	Calculate the product of two matrices. The following instances are supported: csipl_cmprod_split_f csipl_cmprod_split_i csipl_cmprod_split_si
void csipl_cmprodh_inter_P(void * A, int ldA, void * B, int ldB, void * R, int ldR, csipl_length m, csipl_length n, csipl_length p);	Calculate the product a complex matrix and the Hermitian of a complex matrix. The following instances are supported: csipl_cmprodh_inter_f csipl_cmprodh_inter_i csipl_cmprodh_inter_si

Prototype	Description
void csipl_cmprodh_split_P(scalar_P * A_re, scalar_P * A_im, int ldA, scalar_P * B_re, scalar_P * B_im, int ldB, scalar_P * R_re, scalar_P * R_im, int ldR, csipl_length m, csipl_length n, csipl_length p);	Calculate the product a complex matrix and the Hermitian of a complex matrix. The following instances are supported: csipl_cmprodh_split_f csipl_cmprodh_split_i csipl_cmprodh_split_si
void csipl_cmprodj_inter_P(void * A, int ldA, void * B, int ldB, void * R, int ldR, csipl_length m, csipl_length n, csipl_length p);	Calculate the product a complex matrix and the conjugate of a complex matrix. The following instances are supported: csipl_cmprodj_inter_f csipl_cmprodj_inter_i csipl_cmprodj_inter_si
void csipl_cmprodj_split_P(scalar_P * A_re, scalar_P * A_im, int ldA, scalar_P * B_re, scalar_P * B_im, int ldB, scalar_P * R_re, scalar_P * R_im, int ldR, csipl_length m, csipl_length n, csipl_length p);	Calculate the product a complex matrix and the conjugate of a complex matrix. The following instances are supported: csipl_cmprodj_split_f csipl_cmprodj_split_i csipl_cmprodj_split_si
void csipl_mprodt_P(scalar_P * A, int ldA, scalar_P * B, int ldB, scalar_P * R, int ldR, csipl_length m, csipl_length n, csipl_length p);	Calculate the product of a matrix and the transpose of a matrix. The following instances are supported: csipl_mprodt_f csipl_mprodt_i csipl_mprodt_si
void csipl_cmprodt_inter_P(void * A, int ldA, void * B, int ldB, void * R, int ldR, csipl_length m, csipl_length n, csipl_length p);	Calculate the product of a matrix and the transpose of a matrix. The following instances are supported: csipl_cmprodt_inter_f csipl_cmprodt_inter_i csipl_cmprodt_inter_si

Prototype	Description
void csipl_cmprodt_split_P(scalar_P * A_re, scalar_P * A_im, int ldA, scalar_P * B_re, scalar_P * B_im, int ldB, scalar_P * R_re, scalar_P * R_im, int ldR, csipl_length m, csipl_length n, csipl_length p);	Calculate the product of a matrix and the transpose of a matrix. The following instances are supported: csipl_cmprodt_split_f csipl_cmprodt_split_i csipl_cmprodt_split_si
void csipl_mvprod_P(scalar_P * A, int ldA, scalar_P * X, csipl_stride strideX, scalar_P * Y, csipl_stride strideY, csipl_length m, csipl_length n);	Calculate a matrix–vector product. The following instances are supported: csipl_mvprod_f csipl_mvprod_i csipl_mvprod_si
void csipl_cmvprod_inter_P(void * A, int ldA, void * X, csipl_stride strideX, void * Y, csipl_stride strideY, csipl_length m, csipl_length n);	Calculate a matrix–vector product. The following instances are supported: csipl_cmvprod_inter_f csipl_cmvprod_inter_i csipl_cmvprod_inter_si
void csipl_cmvprod_split_P(scalar_P * A_re, scalar_P * A_im, int ldA, scalar_P * X_re, scalar_P * X_im, csipl_stride strideX, scalar_P * Y_re, scalar_P * Y_im, csipl_stride strideY, csipl_length m, csipl_length n);	Calculate a matrix–vector product. The following instances are supported: csipl_cmvprod_split_f csipl_cmvprod_split_i csipl_cmvprod_split_si
void csipl_mtrans_P(scalar_P * A, int ldA, scalar_P * R, int ldR, csipl_length m, csipl_length n);	Transpose a matrix. The following instances are supported: csipl_mtrans_bl csipl_mtrans_f csipl_mtrans_i csipl_mtrans_si
void csipl_cmtrans_inter_P(void * A, int ldA, void * R, int ldR, csipl_length m, csipl_length n);	Transpose a matrix. The following instances are supported: csipl_cmtrans_inter_f csipl_cmtrans_inter_i csipl_cmtrans_inter_si

Prototype	Description
void csipl_cmtrans_split_P(scalar_P * A_re, scalar_P * A_im, int ldA, scalar_P * R_re, scalar_P * R_im, int ldR, csipl_length m, csipl_length n);	Transpose a matrix. The following instances are supported: csipl_cmtrans_split_f csipl_cmtrans_split_i csipl_cmtrans_split_si
float csipl_vdot_f(float * A, csipl_stride strideA, float * B, csipl_stride strideB, csipl_length n);	Compute the inner (dot) product of two vectors.
csipl_cscalar_f csipl_cvdot_inter_f(void * A, csipl_stride strideA, void * B, csipl_stride strideB, csipl_length n);	Compute the inner (dot) product of two vectors.
csipl_cscalar_f csipl_cvdot_split_f(float * A_re, float * A_im, csipl_stride strideA, float * B_re, float * B_im, csipl_stride strideB, csipl_length n);	Compute the inner (dot) product of two vectors.
void csipl_vmprod_P(scalar_P * X, csipl_stride strideX, scalar_P * A, int ldA, scalar_P * Y, csipl_stride strideY, csipl_length m, csipl_length n);	Calculate a vector–matrix product. The following instances are supported: csipl_vmprod_f csipl_vmprod_i csipl_vmprod_si
void csipl_cvmprod_inter_P(void * X, csipl_stride strideX, void * A, int ldA, void * Y, csipl_stride strideY, csipl_length m, csipl_length n);	Calculate a vector–matrix product. The following instances are supported: csipl_cvmprod_inter_f csipl_cvmprod_inter_i csipl_cvmprod_inter_si

Prototype	Description
void csipl_cvmprod_split_P(scalar_P * X_re, scalar_P * X_im, csipl_stride strideX, scalar_P * A_re, scalar_P * A_im, int ldA, scalar_P * Y_re, scalar_P * Y_im, csipl_stride strideY, csipl_length m, csipl_length n);	Calculate a vector–matrix product. The following instances are supported: csipl_cvmprod_split_f csipl_cvmprod_split_i csipl_cvmprod_split_si
void csipl_vouter_f(float alpha, float * X, csipl_stride strideX, float * Y, csipl_stride strideY, float * R, int ldR, csipl_length m, csipl_length n);	Calculate the outer product of two vectors.
void csipl_cvouter_inter_f(csipl_cscalar_f alpha, void * X, csipl_stride strideX, void * Y, csipl_stride strideY, void * R, int ldR, csipl_length m, csipl_length n);	Calculate the outer product of two vectors.
void csipl_cvouter_split_f(csipl_cscalar_f alpha_re, csipl_cscalar_f alpha_im, float * X_re, float * X_im, csipl_stride strideX, float * Y_re, float * Y_im, csipl_stride strideY, float * R_re, float * R_im, int ldR, csipl_length m, csipl_length n);	Calculate the outer product of two vectors.
float csipl_vcsummgval_inter_f(void * A, csipl_stride strideA, csipl_length n);	Returns the sum of the magnitudes of the elements of a complex vector.
float csipl_vcsummgval_split_f(float * A_re, float * A_im, csipl_stride strideA, csipl_length n);	Returns the sum of the magnitudes of the elements of a complex vector.

Prototype	Description
void csipl_minvlu_f(float * A, int ldA, signed int * V, csipl_stride strideV, float * R, int ldR, csipl_length n);	Invert a square matrix using LU decomposition.
void csipl_cminvlu_inter_f(void * A, int ldA, signed int * V, csipl_stride strideV, void * R, int ldR, csipl_length n);	Invert a square matrix using LU decomposition.
void csipl_cminvlu_split_f(float * A_re, float * A_im, int ldA, signed int * V, csipl_stride strideV, float * R_re, float * R_im, int ldR, csipl_length n);	Invert a square matrix using LU decomposition.

8.2 Special Linear System Solvers

Prototype	Description
`int csipl_covsol_f( float * A, int ldA, float * XB, int ldXB, csipl_length m, csipl_length n, csipl_length p);`	Solve a covariance linear system problem.
`int csipl_ccovsol_inter_f( void * A, int ldA, void * XB, int ldXB, csipl_length m, csipl_length n, csipl_length p);`	Solve a covariance linear system problem.
`int csipl_ccovsol_split_f( float * A_re, float * A_im, int ldA, float * XB_re, float * XB_im, int ldXB, csipl_length m, csipl_length n, csipl_length p);`	Solve a covariance linear system problem.
`int csipl_llsqsol_f( float * A, int ldA, float * XB, int ldXB, csipl_length m, csipl_length n, csipl_length p);`	Solve a linear least squares problem.
`int csipl_cllsqsol_inter_f( void * A, int ldA, void * XB, int ldXB, csipl_length m, csipl_length n, csipl_length p);`	Solve a linear least squares problem.
`int csipl_cllsqsol_split_f( float * A_re, float * A_im, int ldA, float * XB_re, float * XB_im, int ldXB, csipl_length m, csipl_length n, csipl_length p);`	Solve a linear least squares problem.

Prototype	Description
int csipl_toepsol_f(float * T, csipl_stride strideT, float * B, csipl_stride strideB, float * W, csipl_stride strideW, float * X, csipl_stride strideX, csipl_length n);	Solve a real symmetric positive definite Toeplitz linear system.
int csipl_ctoepsol_inter_f(void * T, csipl_stride strideT, void * B, csipl_stride strideB, void * W, csipl_stride strideW, void * X, csipl_stride strideX, csipl_length n);	Solve a real symmetric positive definite Toeplitz linear system.
int csipl_ctoepsol_split_f(float * T_re, float * T_im, csipl_stride strideT, float * B_re, float * B_im, csipl_stride strideB, float * W_re, float * W_im, csipl_stride strideW, float * X_re, float * X_im, csipl_stride strideX, csipl_length n);	Solve a real symmetric positive definite Toeplitz linear system.

8.3 General Square Linear System Solver

Prototype	Description
int csipl_lud_f(csipl_clu_f * lud, float * A, int ldA, csipl_length n);	Compute an LU decomposition of a square matrix using partial pivoting.
int csipl_clud_inter_f(csipl_clu_f * lud, void * A, int ldA, csipl_length n);	Compute an LU decomposition of a square matrix using partial pivoting.
int csipl_clud_split_f(csipl_clu_f * lud, float * A_re, float * A_im, int ldA, csipl_length n);	Compute an LU decomposition of a square matrix using partial pivoting.

Prototype	Description
csipl_Dlu_P * csipl_Dlud_create_P(unsigned int N);	Create an LU decomposition object. The following instances are supported: csipl_lud_create_f csipl_clud_create_f
int csipl_Dlud_destroy_P(csipl_Dlu_P * lud);	Destroy an LU decomposition object. The following instances are supported: csipl_lud_destroy_f csipl_clud_destroy_f
void csipl_Dlud_getattr_P(csipl_Dlu_P * lud, csipl_Dlu_attr_P * attr);	Returns the attributes of an LU decomposition object. The following instances are supported: csipl_lud_getattr_f csipl_clud_getattr_f
int csipl_lusol_f(csipl_clu_f * clud, csipl_mat_op opA, float * XB, int ldXB, csipl_length m, csipl_length n);	Solve a square linear system.
int csipl_clusol_inter_f(csipl_clu_f * clud, csipl_mat_op opA, void * XB, int ldXB, csipl_length m, csipl_length n);	Solve a square linear system.
int csipl_clusol_split_f(csipl_clu_f * clud_re, csipl_clu_f * clud_im, csipl_mat_op opA, float * XB_re, float * XB_im, int ldXB, csipl_length m, csipl_length n);	Solve a square linear system.

8.4 Symmetric Positive Definite Linear System Solver

Prototype	Description
int csipl_chold_f(csipl_cchol_f * chold, float * A, int ldA, csipl_length n);	Compute a Cholesky decomposition of a symmetric positive definite matrix.
int csipl_cchold_inter_f(csipl_cchol_f * chold, void * A, int ldA, csipl_length n);	Compute a Cholesky decomposition of a symmetric positive definite matrix.

Prototype	Description
int csipl_cchold_split_f(csipl_cchol_f * chold, float * A_re, float * A_im, int ldA, csipl_length n);	Compute a Cholesky decomposition of a symmetric positive definite matrix.
csipl_chol_f * csipl_chold_create_f(csipl_mat_uplo uplo, unsigned int n);	Creates a Cholesky decomposition object.
csipl_cchol_f * csipl_cchold_create_f(csipl_mat_uplo uplo, unsigned int n);	Creates a Cholesky decomposition object.
int csipl_Dchold_destroy_P(csipl_Dchol_P * chold);	Destroy a Cholesky decomposition object. The following instances are supported: csipl_chold_destroy_f csipl_cchold_destroy_f
void csipl_Dchold_getattr_P(csipl_Dchol_P * chold, csipl_Dchol_attr_P * attr);	Returns the attributes of a Cholesky decomposition object. The following instances are supported: csipl_chold_getattr_f csipl_cchold_getattr_f
int csipl_cholsol_f(csipl_cchol_f * chold, float * XB, int ldXB, csipl_length m, csipl_length n);	Solve a symmetric positive definite linear system.
int csipl_ccholsol_inter_f(csipl_cchol_f * chold, void * XB, int ldXB, csipl_length m, csipl_length n);	Solve a symmetric positive definite linear system.
int csipl_ccholsol_split_f(csipl_cchol_f * chold, float * XB_re, float * XB_im, int ldXB, csipl_length m, csipl_length n);	Solve a symmetric positive definite linear system.

8.5 Overdetermined Linear System Solver

Prototype	Description
int csipl_qrd_f(csipl_cqr_f * qrd, float * A, int ldA, csipl_length m, csipl_length n);	Compute a QR decomposition of a matrix .

Prototype	Description
int csipl_cqrd_inter_f(csipl_cqr_f * qrd, void * A, int ldA, csipl_length m, csipl_length n);	Compute a QR decomposition of a matrix .
int csipl_cqrd_split_f(csipl_cqr_f * qrd, float * A_re, float * A_im, int ldA, csipl_length m, csipl_length n);	Compute a QR decomposition of a matrix .
csipl_qr_f * csipl_qrd_create_f(unsigned int m, unsigned int n, csipl_qrd_qopt qopt);	Create a QR decomposition object.
csipl_cqr_f * csipl_cqrd_create_f(unsigned int m, unsigned int n, csipl_qrd_qopt qopt);	Create a QR decomposition object.
int csipl_Dqrd_destroy_P(csipl_Dqr_P * qrd);	Destroy a QR decomposition object. The following instances are supported: csipl_qrd_destroy_f csipl_cqrd_destroy_f
void csipl_Dqrd_getattr_P(csipl_Dqr_P * qrd, csipl_Dqr_attr_P * attr);	Returns the attributes of a QR decomposition object. The following instances are supported: csipl_qrd_getattr_f csipl_cqrd_getattr_f
int csipl_qrdprodq_f(csipl_qr_f * qrd, csipl_mat_op opQ, csipl_mat_side apQ, float * C, int ldC, csipl_length m, csipl_length n);	Multiply a matrix by the matrix Q from a QR decomposition.
int csipl_cqrdprodq_inter_f(csipl_cqr_f * qrd, csipl_mat_op opQ, csipl_mat_side apQ, void * C, int ldC, csipl_length m, csipl_length n);	Multiply a matrix by the matrix Q from a QR decomposition.

Prototype	Description
int csipl_cqrdprodq_split_f(csipl_cqr_f * qrd_re, csipl_cqr_f * qrd_im, csipl_mat_op opQ, csipl_mat_side apQ, float * C_re, float * C_im, int ldC, csipl_length m, csipl_length n);	Multiply a matrix by the matrix Q from a QR decomposition.
int csipl_qrdsolr_f(csipl_qr_f * qrd, csipl_mat_op OpR, float alpha, float * XB, int ldXB, csipl_length m, csipl_length n);	Solve linear system based on the matrix R, from QR decomposition of the matrix A.
int csipl_cqrdsolr_inter_f(csipl_cqr_f * qrd, csipl_mat_op OpR, csipl_cscalar_f alpha, void * XB, int ldXB, csipl_length m, csipl_length n);	Solve linear system based on the matrix R, from QR decomposition of the matrix A.
int csipl_cqrdsolr_split_f(csipl_cqr_f * qrd_re, csipl_cqr_f * qrd_im, csipl_mat_op OpR, csipl_cscalar_f alpha_re, csipl_cscalar_f alpha_im, float * XB_re, float * XB_im, int ldXB, csipl_length m, csipl_length n);	Solve linear system based on the matrix R, from QR decomposition of the matrix A.
int csipl_qrsol_f(csipl_qr_f * qrd, csipl_qrd_prob prob, float * XB, int ldXB, csipl_length m, csipl_length n);	Solve either a linear covariance or linear least squares problem.
int csipl_cqrsol_inter_f(csipl_cqr_f * qrd, csipl_qrd_prob prob, void * XB, int ldXB, csipl_length m, csipl_length n);	Solve either a linear covariance or linear least squares problem.

Prototype	Description

`int`
`csipl_cqrsol_split_f(`
`csipl_cqr_f * qrd_re,`
`csipl_cqr_f * qrd_im,`
`csipl_qrd_prob prob,`
`void  * XB,`
`int ldXB,`
`csipl_length m,`
`csipl_length n);`
	Solve either a linear covariance or linear least squares problem.