

ABIP-QCP

Generated by Doxygen 1.9.5



<b>1 Class Index</b>	<b>1</b>
1.1 Class List	1
<b>2 File Index</b>	<b>3</b>
2.1 File List	3
<b>3 Class Documentation</b>	<b>7</b>
3.1 ABIP_A_DATA_MATRIX Struct Reference	7
3.1.1 Detailed Description	7
3.1.2 Member Data Documentation	7
3.1.2.1 i	7
3.1.2.2 m	8
3.1.2.3 n	8
3.1.2.4 p	8
3.1.2.5 x	8
3.2 ABIP_CONE Struct Reference	8
3.2.1 Detailed Description	9
3.2.2 Member Data Documentation	9
3.2.2.1 f	9
3.2.2.2 l	9
3.2.2.3 q	9
3.2.2.4 qsize	9
3.2.2.5 rq	9
3.2.2.6 rqsize	10
3.2.2.7 z	10
3.3 ABIP_INFO Struct Reference	10
3.3.1 Detailed Description	10
3.3.2 Member Data Documentation	10
3.3.2.1 admm_iter	11
3.3.2.2 avg_cg_iters	11
3.3.2.3 avg_linsys_time	11
3.3.2.4 dobj	11
3.3.2.5 ipm_iter	11
3.3.2.6 pobj	11
3.3.2.7 rel_gap	12
3.3.2.8 res_dual	12
3.3.2.9 res_infeas	12
3.3.2.10 res_pri	12
3.3.2.11 res_unbdd	12
3.3.2.12 setup_time	12
3.3.2.13 solve_time	13
3.3.2.14 status	13
3.3.2.15 status_val	13

3.4 ABIP_LIN_SYS_WORK Struct Reference	13
3.4.1 Detailed Description	14
3.4.2 Member Data Documentation	14
3.4.2.1 bp	14
3.4.2.2 ddum	14
3.4.2.3 Dinv	14
3.4.2.4 error	14
3.4.2.5 handle	14
3.4.2.6 idum	15
3.4.2.7 iparm	15
3.4.2.8 K	15
3.4.2.9 L	15
3.4.2.10 M	15
3.4.2.11 maxfct	15
3.4.2.12 mnum	16
3.4.2.13 msglvl	16
3.4.2.14 mtype	16
3.4.2.15 N	16
3.4.2.16 nnz_LDL	16
3.4.2.17 P	16
3.4.2.18 pt	17
3.4.2.19 S	17
3.4.2.20 total_cg_iters	17
3.4.2.21 total_solve_time	17
3.4.2.22 U	17
3.5 ABIP_PROBLEM_DATA Struct Reference	17
3.5.1 Detailed Description	18
3.5.2 Member Data Documentation	18
3.5.2.1 A	18
3.5.2.2 b	18
3.5.2.3 c	18
3.5.2.4 lambda	18
3.5.2.5 m	19
3.5.2.6 n	19
3.5.2.7 Q	19
3.5.2.8 stgs	19
3.6 ABIP_RESIDUALS Struct Reference	19
3.6.1 Detailed Description	20
3.6.2 Member Data Documentation	20
3.6.2.1 Ax_b_norm	20
3.6.2.2 bt_y_by_tau	20
3.6.2.3 ct_x_by_tau	21

3.6.2.4 dobj	21
3.6.2.5 error_ratio	21
3.6.2.6 kap	21
3.6.2.7 last_admm_iter	21
3.6.2.8 last_ipm_iter	21
3.6.2.9 last_mu	22
3.6.2.10 pobj	22
3.6.2.11 Qx_ATy_c_s_norm	22
3.6.2.12 rel_gap	22
3.6.2.13 res_dif	22
3.6.2.14 res_dual	22
3.6.2.15 res_infeas	23
3.6.2.16 res_pri	23
3.6.2.17 res_unbdd	23
3.6.2.18 tau	23
3.7 ABIP_SETTINGS Struct Reference	23
3.7.1 Detailed Description	24
3.7.2 Member Data Documentation	24
3.7.2.1 alpha	24
3.7.2.2 cg_rate	25
3.7.2.3 eps	25
3.7.2.4 eps_d	25
3.7.2.5 eps_g	25
3.7.2.6 eps_inf	25
3.7.2.7 eps_p	25
3.7.2.8 eps_unb	26
3.7.2.9 err_dif	26
3.7.2.10 inner_check_period	26
3.7.2.11 linsys_solver	26
3.7.2.12 max_admm_iters	26
3.7.2.13 max_ipm_iters	26
3.7.2.14 normalize	27
3.7.2.15 origin_scaling	27
3.7.2.16 outer_check_period	27
3.7.2.17 pc_scaling	27
3.7.2.18 prob_type	27
3.7.2.19 psi	27
3.7.2.20 rho_tau	28
3.7.2.21 rho_x	28
3.7.2.22 rho_y	28
3.7.2.23 ruiz_scaling	28
3.7.2.24 scale	28

3.7.2.25 scale_bc	28
3.7.2.26 scale_E	29
3.7.2.27 time_limit	29
3.7.2.28 use_indirect	29
3.7.2.29 verbose	29
3.8 ABIP_SOL_VARS Struct Reference	29
3.8.1 Detailed Description	29
3.8.2 Member Data Documentation	30
3.8.2.1 s	30
3.8.2.2 x	30
3.8.2.3 y	30
3.9 ABIP_WORK Struct Reference	30
3.9.1 Detailed Description	31
3.9.2 Member Data Documentation	31
3.9.2.1 A	31
3.9.2.2 a	31
3.9.2.3 beta	31
3.9.2.4 gamma	31
3.9.2.5 m	31
3.9.2.6 mu	32
3.9.2.7 n	32
3.9.2.8 nm_inf_b	32
3.9.2.9 nm_inf_c	32
3.9.2.10 r	32
3.9.2.11 rel_ut	32
3.9.2.12 sigma	33
3.9.2.13 u	33
3.9.2.14 u_t	33
3.9.2.15 v	33
3.9.2.16 v_origin	33
3.10 cs_dmperm_results Struct Reference	33
3.10.1 Detailed Description	34
3.10.2 Member Data Documentation	34
3.10.2.1 cc	34
3.10.2.2 nb	34
3.10.2.3 p	34
3.10.2.4 q	34
3.10.2.5 r	35
3.10.2.6 rr	35
3.10.2.7 s	35
3.11 cs_numeric Struct Reference	35
3.11.1 Detailed Description	35

3.11.2 Member Data Documentation	35
3.11.2.1 B	36
3.11.2.2 L	36
3.11.2.3 pinv	36
3.11.2.4 U	36
3.12 cs_sparse Struct Reference	36
3.12.1 Detailed Description	37
3.12.2 Member Data Documentation	37
3.12.2.1 i	37
3.12.2.2 m	37
3.12.2.3 n	37
3.12.2.4 nz	37
3.12.2.5 nzmax	37
3.12.2.6 p	38
3.12.2.7 x	38
3.13 cs_symbolic Struct Reference	38
3.13.1 Detailed Description	38
3.13.2 Member Data Documentation	38
3.13.2.1 cp	38
3.13.2.2 leftmost	39
3.13.2.3 lnz	39
3.13.2.4 m2	39
3.13.2.5 parent	39
3.13.2.6 pinv	39
3.13.2.7 q	39
3.13.2.8 unz	40
3.14 Lasso Struct Reference	40
3.14.1 Detailed Description	41
3.14.2 Member Data Documentation	41
3.14.2.1 A	41
3.14.2.2 b	41
3.14.2.3 c	41
3.14.2.4 calc_residuals	41
3.14.2.5 D	41
3.14.2.6 D_hat	42
3.14.2.7 data	42
3.14.2.8 E	42
3.14.2.9 free_spe_linsys_work	42
3.14.2.10 init_spe_linsys_work	42
3.14.2.11 inner_conv_check	42
3.14.2.12 L	43
3.14.2.13 lambda	43

3.14.2.14 m	43
3.14.2.15 n	43
3.14.2.16 p	43
3.14.2.17 pro_type	43
3.14.2.18 q	44
3.14.2.19 Q	44
3.14.2.20 rho_dr	44
3.14.2.21 sc	44
3.14.2.22 sc_b	44
3.14.2.23 sc_c	44
3.14.2.24 sc_cone1	45
3.14.2.25 sc_cone2	45
3.14.2.26 scaling_data	45
3.14.2.27 solve_spe_linsys	45
3.14.2.28 sparsity	45
3.14.2.29 spe_A_times	45
3.14.2.30 spe_AT_times	46
3.14.2.31 stgs	46
3.14.2.32 un_scaling_sol	46
3.15 qcp Struct Reference	46
3.15.1 Detailed Description	47
3.15.2 Member Data Documentation	47
3.15.2.1 A	47
3.15.2.2 b	47
3.15.2.3 c	47
3.15.2.4 calc_residuals	47
3.15.2.5 D	47
3.15.2.6 data	48
3.15.2.7 E	48
3.15.2.8 free_spe_linsys_work	48
3.15.2.9 init_spe_linsys_work	48
3.15.2.10 inner_conv_check	48
3.15.2.11 L	48
3.15.2.12 m	49
3.15.2.13 n	49
3.15.2.14 p	49
3.15.2.15 pro_type	49
3.15.2.16 q	49
3.15.2.17 Q	49
3.15.2.18 rho_dr	50
3.15.2.19 sc_b	50
3.15.2.20 sc_c	50



3.15.2.21 scaling_data . . . . .	50
3.15.2.22 solve_spe_linsys . . . . .	50
3.15.2.23 sparsity . . . . .	50
3.15.2.24 spe_A_times . . . . .	51
3.15.2.25 spe_AT_times . . . . .	51
3.15.2.26 stgs . . . . .	51
3.15.2.27 un_scaling_sol . . . . .	51
3.16 solve_specific_problem Struct Reference . . . . .	51
3.16.1 Detailed Description . . . . .	52
3.16.2 Member Data Documentation . . . . .	52
3.16.2.1 A . . . . .	52
3.16.2.2 b . . . . .	52
3.16.2.3 c . . . . .	53
3.16.2.4 calc_residuals . . . . .	53
3.16.2.5 data . . . . .	53
3.16.2.6 free_spe_linsys_work . . . . .	53
3.16.2.7 init_spe_linsys_work . . . . .	53
3.16.2.8 inner_conv_check . . . . .	53
3.16.2.9 L . . . . .	54
3.16.2.10 m . . . . .	54
3.16.2.11 n . . . . .	54
3.16.2.12 p . . . . .	54
3.16.2.13 pro_type . . . . .	54
3.16.2.14 q . . . . .	54
3.16.2.15 Q . . . . .	55
3.16.2.16 rho_dr . . . . .	55
3.16.2.17 scaling_data . . . . .	55
3.16.2.18 solve_spe_linsys . . . . .	55
3.16.2.19 sparsity . . . . .	55
3.16.2.20 spe_A_times . . . . .	55
3.16.2.21 spe_AT_times . . . . .	56
3.16.2.22 stgs . . . . .	56
3.16.2.23 un_scaling_sol . . . . .	56
3.17 SuiteSparse_config_struct Struct Reference . . . . .	56
3.17.1 Detailed Description . . . . .	56
3.17.2 Member Data Documentation . . . . .	57
3.17.2.1 calloc_func . . . . .	57
3.17.2.2 divcomplex_func . . . . .	57
3.17.2.3 free_func . . . . .	57
3.17.2.4 hypot_func . . . . .	57
3.17.2.5 malloc_func . . . . .	57
3.17.2.6 printf_func . . . . .	58

3.17.2.7 realloc_func	58
3.18 Svm Struct Reference	58
3.18.1 Detailed Description	59
3.18.2 Member Data Documentation	59
3.18.2.1 A	59
3.18.2.2 b	59
3.18.2.3 c	59
3.18.2.4 calc_residuals	60
3.18.2.5 data	60
3.18.2.6 free_spe_linsys_work	60
3.18.2.7 init_spe_linsys_work	60
3.18.2.8 inner_conv_check	60
3.18.2.9 L	60
3.18.2.10 lambda	61
3.18.2.11 m	61
3.18.2.12 n	61
3.18.2.13 p	61
3.18.2.14 pro_type	61
3.18.2.15 q	61
3.18.2.16 Q	62
3.18.2.17 rho_dr	62
3.18.2.18 sc	62
3.18.2.19 sc_b	62
3.18.2.20 sc_c	62
3.18.2.21 sc_cone1	62
3.18.2.22 sc_cone2	63
3.18.2.23 sc_D	63
3.18.2.24 sc_E	63
3.18.2.25 sc_F	63
3.18.2.26 scaling_data	63
3.18.2.27 solve_spe_linsys	63
3.18.2.28 sparsity	64
3.18.2.29 spe_A_times	64
3.18.2.30 spe_AT_times	64
3.18.2.31 stgs	64
3.18.2.32 un_scaling_sol	64
3.18.2.33 wA	64
3.18.2.34 wB	65
3.18.2.35 wC	65
3.18.2.36 wD	65
3.18.2.37 wE	65
3.18.2.38 wF	65

3.18.2.39 wG . . . . .	65
3.18.2.40 wH . . . . .	66
3.18.2.41 wX . . . . .	66
3.18.2.42 wy . . . . .	66
3.19 SVMqp Struct Reference . . . . .	66
3.19.1 Detailed Description . . . . .	67
3.19.2 Member Data Documentation . . . . .	67
3.19.2.1 A . . . . .	67
3.19.2.2 b . . . . .	67
3.19.2.3 c . . . . .	67
3.19.2.4 calc_residuals . . . . .	68
3.19.2.5 D . . . . .	68
3.19.2.6 data . . . . .	68
3.19.2.7 E . . . . .	68
3.19.2.8 F . . . . .	68
3.19.2.9 free_spe_linsys_work . . . . .	68
3.19.2.10 H . . . . .	69
3.19.2.11 init_spe_linsys_work . . . . .	69
3.19.2.12 inner_conv_check . . . . .	69
3.19.2.13 L . . . . .	69
3.19.2.14 lambda . . . . .	69
3.19.2.15 m . . . . .	69
3.19.2.16 n . . . . .	70
3.19.2.17 p . . . . .	70
3.19.2.18 pro_type . . . . .	70
3.19.2.19 q . . . . .	70
3.19.2.20 Q . . . . .	70
3.19.2.21 rho_dr . . . . .	70
3.19.2.22 sc_b . . . . .	71
3.19.2.23 sc_c . . . . .	71
3.19.2.24 scaling_data . . . . .	71
3.19.2.25 solve_spe_linsys . . . . .	71
3.19.2.26 sparsity . . . . .	71
3.19.2.27 spe_A_times . . . . .	71
3.19.2.28 spe_AT_times . . . . .	72
3.19.2.29 stgs . . . . .	72
3.19.2.30 un_scaling_sol . . . . .	72
<b>4 File Documentation</b>	<b>73</b>
4.1 amd/amd.h File Reference . . . . .	73
4.1.1 Macro Definition Documentation . . . . .	74
4.1.1.1 AMD_AGGRESSIVE . . . . .	74

4.1.1.2 AMD_CONTROL	74
4.1.1.3 AMD_DATE	75
4.1.1.4 AMD_DEFAULT_AGGRESSIVE	75
4.1.1.5 AMD_DEFAULT_DENSE	75
4.1.1.6 AMD_DENSE	75
4.1.1.7 AMD_DMAX	75
4.1.1.8 AMD_INFO	75
4.1.1.9 AMD_INVALID	76
4.1.1.10 AMD_LNZ	76
4.1.1.11 AMD_MAIN_VERSION	76
4.1.1.12 AMD_MEMORY	76
4.1.1.13 AMD_N	76
4.1.1.14 AMD_NCMPA	76
4.1.1.15 AMD_NDENSE	77
4.1.1.16 AMD_NDIV	77
4.1.1.17 AMD_NMULTSUBS_LDL	77
4.1.1.18 AMD_NMULTSUBS_LU	77
4.1.1.19 AMD_NZ	77
4.1.1.20 AMD_NZ_A_PLUS_AT	77
4.1.1.21 AMD_NZDIAG	78
4.1.1.22 AMD_OK	78
4.1.1.23 AMD_OK_BUT_JUMBLED	78
4.1.1.24 AMD_OUT_OF_MEMORY	78
4.1.1.25 AMD_STATUS	78
4.1.1.26 AMD_SUB_VERSION	78
4.1.1.27 AMD_SUBSUB_VERSION	79
4.1.1.28 AMD_SYMMETRY	79
4.1.1.29 AMD_VERSION	79
4.1.1.30 AMD_VERSION_CODE	79
4.1.1.31 EXTERN	79
4.1.2 Function Documentation	79
4.1.2.1 amd_2()	80
4.1.2.2 amd_control()	80
4.1.2.3 amd_defaults()	80
4.1.2.4 amd_info()	80
4.1.2.5 amd_l2()	81
4.1.2.6 amd_l_control()	81
4.1.2.7 amd_l_defaults()	81
4.1.2.8 amd_l_info()	81
4.1.2.9 amd_l_order()	81
4.1.2.10 amd_l_valid()	82
4.1.2.11 amd_order()	82

4.1.2.12 amd_valid()	82
4.1.3 Variable Documentation	82
4.1.3.1 amd_calloc	82
4.1.3.2 amd_free	82
4.1.3.3 amd_malloc	83
4.1.3.4 amd_printf	83
4.1.3.5 amd_realloc	83
4.2 amd.h	83
4.3 amd/amd_1.c File Reference	88
4.3.1 Function Documentation	88
4.3.1.1 AMD_1()	88
4.4 amd_1.c	89
4.5 amd/amd_2.c File Reference	91
4.5.1 Function Documentation	91
4.5.1.1 AMD_2()	91
4.6 amd_2.c	92
4.7 amd/amd_aat.c File Reference	115
4.7.1 Function Documentation	115
4.7.1.1 AMD_aat()	115
4.8 amd_aat.c	116
4.9 amd/amd_control.c File Reference	118
4.9.1 Function Documentation	118
4.9.1.1 AMD_control()	118
4.10 amd_control.c	119
4.11 amd/amd_defaults.c File Reference	119
4.11.1 Function Documentation	120
4.11.1.1 AMD_defaults()	120
4.12 amd_defaults.c	120
4.13 amd/amd_dump.c File Reference	120
4.13.1 Function Documentation	121
4.13.1.1 AMD_debug_init()	121
4.13.1.2 AMD_dump()	121
4.13.2 Variable Documentation	121
4.13.2.1 AMD_debug	121
4.14 amd_dump.c	122
4.15 amd/amd_global.c File Reference	124
4.15.1 Macro Definition Documentation	124
4.15.1.1 ABIP_NULL	124
4.15.2 Variable Documentation	125
4.15.2.1 amd_calloc	125
4.15.2.2 amd_free	125
4.15.2.3 amd_malloc	125

4.15.2.4 amd_printf . . . . .	125
4.15.2.5 amd_realloc . . . . .	125
4.16 amd_global.c . . . . .	126
4.17 amd/amd_info.c File Reference . . . . .	127
4.17.1 Macro Definition Documentation . . . . .	127
4.17.1.1 PRI . . . . .	127
4.17.2 Function Documentation . . . . .	127
4.17.2.1 AMD_info() . . . . .	127
4.18 amd_info.c . . . . .	128
4.19 amd/amd_internal.h File Reference . . . . .	129
4.19.1 Macro Definition Documentation . . . . .	130
4.19.1.1 ABIP_NULL . . . . .	130
4.19.1.2 AMD_1 . . . . .	130
4.19.1.3 AMD_2 . . . . .	131
4.19.1.4 AMD_aat . . . . .	131
4.19.1.5 AMD_control . . . . .	131
4.19.1.6 AMD_debug . . . . .	131
4.19.1.7 AMD_DEBUG0 . . . . .	131
4.19.1.8 AMD_DEBUG1 . . . . .	131
4.19.1.9 AMD_DEBUG2 . . . . .	132
4.19.1.10 AMD_DEBUG3 . . . . .	132
4.19.1.11 AMD_DEBUG4 . . . . .	132
4.19.1.12 AMD_debug_init . . . . .	132
4.19.1.13 AMD_defaults . . . . .	132
4.19.1.14 AMD_dump . . . . .	132
4.19.1.15 AMD_info . . . . .	133
4.19.1.16 AMD_order . . . . .	133
4.19.1.17 AMD_post_tree . . . . .	133
4.19.1.18 AMD_postorder . . . . .	133
4.19.1.19 AMD_preprocess . . . . .	133
4.19.1.20 AMD_valid . . . . .	133
4.19.1.21 ASSERT . . . . .	134
4.19.1.22 EMPTY [1/2] . . . . .	134
4.19.1.23 EMPTY [2/2] . . . . .	134
4.19.1.24 FALSE . . . . .	134
4.19.1.25 FLIP . . . . .	134
4.19.1.26 GLOBAL . . . . .	134
4.19.1.27 ID . . . . .	135
4.19.1.28 IMPLIES . . . . .	135
4.19.1.29 Int . . . . .	135
4.19.1.30 Int_MAX . . . . .	135
4.19.1.31 MAX . . . . .	135

4.19.1.32 MIN . . . . .	135
4.19.1.33 PRINTF . . . . .	136
4.19.1.34 PRIVATE . . . . .	136
4.19.1.35 SIZE_T_MAX . . . . .	136
4.19.1.36 TRUE . . . . .	136
4.19.1.37 UNFLIP . . . . .	136
4.19.2 Function Documentation . . . . .	136
4.19.2.1 AMD_1() . . . . .	137
4.19.2.2 AMD_aat() . . . . .	137
4.19.2.3 AMD_post_tree() . . . . .	137
4.19.2.4 AMD_postorder() . . . . .	137
4.19.2.5 AMD_preprocess() . . . . .	138
4.20 amd_internal.h . . . . .	138
4.21 amd/amd_order.c File Reference . . . . .	142
4.21.1 Function Documentation . . . . .	142
4.21.1.1 AMD_order() . . . . .	142
4.22 amd_order.c . . . . .	142
4.23 amd/amd_post_tree.c File Reference . . . . .	145
4.23.1 Function Documentation . . . . .	145
4.23.1.1 AMD_post_tree() . . . . .	145
4.24 amd_post_tree.c . . . . .	146
4.25 amd/amd_postorder.c File Reference . . . . .	147
4.25.1 Function Documentation . . . . .	147
4.25.1.1 AMD_postorder() . . . . .	148
4.26 amd_postorder.c . . . . .	148
4.27 amd/amd_preprocess.c File Reference . . . . .	151
4.27.1 Function Documentation . . . . .	151
4.27.1.1 AMD_preprocess() . . . . .	151
4.28 amd_preprocess.c . . . . .	151
4.29 amd/amd_valid.c File Reference . . . . .	153
4.29.1 Function Documentation . . . . .	153
4.29.1.1 AMD_valid() . . . . .	153
4.30 amd_valid.c . . . . .	153
4.31 amd/SuiteSparse_config.c File Reference . . . . .	155
4.31.1 Function Documentation . . . . .	155
4.31.1.1 SuiteSparse_calloc() . . . . .	155
4.31.1.2 SuiteSparse_divcomplex() . . . . .	156
4.31.1.3 SuiteSparse_finish() . . . . .	156
4.31.1.4 SuiteSparse_free() . . . . .	156
4.31.1.5 SuiteSparse_hypot() . . . . .	156
4.31.1.6 SuiteSparse_malloc() . . . . .	156
4.31.1.7 SuiteSparse_realloc() . . . . .	157

4.31.1.8 SuiteSparse_start()	157
4.31.1.9 SuiteSparse_tic()	157
4.31.1.10 SuiteSparse_time()	157
4.31.1.11 SuiteSparse_toc()	157
4.31.1.12 SuiteSparse_version()	158
4.31.2 Variable Documentation	158
4.31.2.1 SuiteSparse_config	158
4.32 SuiteSparse_config.c	158
4.33 amd/SuiteSparse_config.h File Reference	164
4.33.1 Macro Definition Documentation	165
4.33.1.1 SUITESPARSE_DATE	165
4.33.1.2 SUITESPARSE_HAS_VERSION_FUNCTION	166
4.33.1.3 SuiteSparse_long	166
4.33.1.4 SuiteSparse_long_id	166
4.33.1.5 SuiteSparse_long_idd	166
4.33.1.6 SuiteSparse_long_max	166
4.33.1.7 SUITESPARSE_MAIN_VERSION	166
4.33.1.8 SUITESPARSE_PRINTF	167
4.33.1.9 SUITESPARSE_SUB_VERSION	167
4.33.1.10 SUITESPARSE_SUBSUB_VERSION	167
4.33.1.11 SUITESPARSE_VER_CODE	167
4.33.1.12 SUITESPARSE_VERSION	167
4.33.2 Function Documentation	167
4.33.2.1 SuiteSparse_calloc()	168
4.33.2.2 SuiteSparse_divcomplex()	168
4.33.2.3 SuiteSparse_finish()	168
4.33.2.4 SuiteSparse_free()	168
4.33.2.5 SuiteSparse_hypot()	168
4.33.2.6 SuiteSparse_malloc()	169
4.33.2.7 SuiteSparse_realloc()	169
4.33.2.8 SuiteSparse_start()	169
4.33.2.9 SuiteSparse_tic()	169
4.33.2.10 SuiteSparse_time()	169
4.33.2.11 SuiteSparse_toc()	170
4.33.2.12 SuiteSparse_version()	170
4.33.3 Variable Documentation	170
4.33.3.1 SuiteSparse_config	170
4.34 SuiteSparse_config.h	170
4.35 csparse/Include/cs.h File Reference	173
4.35.1 Macro Definition Documentation	175
4.35.1.1 CS_COPYRIGHT	175
4.35.1.2 CS_CSC	175



4.35.1.3 CS_DATE . . . . .	176
4.35.1.4 CS_FLIP . . . . .	176
4.35.1.5 CS_MARK . . . . .	176
4.35.1.6 CS_MARKED . . . . .	176
4.35.1.7 CS_MAX . . . . .	176
4.35.1.8 CS_MIN . . . . .	177
4.35.1.9 CS_SUBSUB . . . . .	177
4.35.1.10 CS_SUBVER . . . . .	177
4.35.1.11 CS_TRIPLET . . . . .	177
4.35.1.12 CS_UNFLIP . . . . .	177
4.35.1.13 CS_VER . . . . .	177
4.35.1.14 csi . . . . .	178
4.35.2 Typedef Documentation . . . . .	178
4.35.2.1 cs . . . . .	178
4.35.2.2 csd . . . . .	178
4.35.2.3 csn . . . . .	178
4.35.2.4 css . . . . .	178
4.35.3 Function Documentation . . . . .	178
4.35.3.1 cs_add() . . . . .	178
4.35.3.2 cs_amd() . . . . .	179
4.35.3.3 cs_calloc() . . . . .	179
4.35.3.4 cs_chol() . . . . .	179
4.35.3.5 cs_cholsol() . . . . .	179
4.35.3.6 cs_compress() . . . . .	179
4.35.3.7 cs_counts() . . . . .	180
4.35.3.8 cs_cumsum() . . . . .	180
4.35.3.9 cs_dalloc() . . . . .	180
4.35.3.10 cs_ddone() . . . . .	180
4.35.3.11 cs_dfree() . . . . .	180
4.35.3.12 cs_dfs() . . . . .	181
4.35.3.13 cs_dmperm() . . . . .	181
4.35.3.14 cs_done() . . . . .	181
4.35.3.15 cs_droptol() . . . . .	181
4.35.3.16 cs_dropzeros() . . . . .	181
4.35.3.17 cs_dupl() . . . . .	182
4.35.3.18 cs_entry() . . . . .	182
4.35.3.19 cs_ereach() . . . . .	182
4.35.3.20 cs_etree() . . . . .	182
4.35.3.21 cs_fkeep() . . . . .	182
4.35.3.22 cs_free() . . . . .	183
4.35.3.23 cs_gaxpy() . . . . .	183
4.35.3.24 cs_happly() . . . . .	183

4.35.3.25 cs_house()	183
4.35.3.26 cs_idone()	183
4.35.3.27 cs_ipvec()	184
4.35.3.28 cs_leaf()	184
4.35.3.29 cs_load()	184
4.35.3.30 cs_lsolve()	184
4.35.3.31 cs_ltsolve()	184
4.35.3.32 cs_lu()	185
4.35.3.33 cs_lusol()	185
4.35.3.34 cs_malloc()	185
4.35.3.35 cs_maxtrans()	185
4.35.3.36 cs_multiply()	185
4.35.3.37 cs_ndone()	186
4.35.3.38 cs_nfree()	186
4.35.3.39 cs_norm()	186
4.35.3.40 cs_permute()	186
4.35.3.41 cs_pinv()	186
4.35.3.42 cs_post()	187
4.35.3.43 cs_print()	187
4.35.3.44 cs_pvec()	187
4.35.3.45 cs_qr()	187
4.35.3.46 cs_qrsol()	187
4.35.3.47 cs_randperm()	188
4.35.3.48 cs_reach()	188
4.35.3.49 cs_realloc()	188
4.35.3.50 cs_scatter()	188
4.35.3.51 cs_scc()	189
4.35.3.52 cs_schol()	189
4.35.3.53 cs_sfree()	189
4.35.3.54 cs_spalloc()	189
4.35.3.55 cs_spfree()	189
4.35.3.56 cs_sprealloc()	190
4.35.3.57 cs_spsolve()	190
4.35.3.58 cs_sqr()	190
4.35.3.59 cs_symperm()	190
4.35.3.60 cs_tdfs()	191
4.35.3.61 cs_transpose()	191
4.35.3.62 cs_updown()	191
4.35.3.63 cs_usolve()	191
4.35.3.64 cs_utsolve()	191
4.36 cs.h	192
4.37 csparse/Source/cs_add.c File Reference	193

4.37.1 Function Documentation . . . . .	194
4.37.1.1 cs_add() . . . . .	194
4.38 cs_add.c . . . . .	194
4.39 csparse/Source/cs_amd.c File Reference . . . . .	194
4.39.1 Function Documentation . . . . .	195
4.39.1.1 cs_amd() . . . . .	195
4.40 cs_amd.c . . . . .	195
4.41 csparse/Source/cs_chol.c File Reference . . . . .	199
4.41.1 Function Documentation . . . . .	199
4.41.1.1 cs_chol() . . . . .	199
4.42 cs_chol.c . . . . .	200
4.43 csparse/Source/cs_cholsol.c File Reference . . . . .	200
4.43.1 Function Documentation . . . . .	201
4.43.1.1 cs_cholsol() . . . . .	201
4.44 cs_cholsol.c . . . . .	201
4.45 csparse/Source/cs_compress.c File Reference . . . . .	201
4.45.1 Function Documentation . . . . .	201
4.45.1.1 cs_compress() . . . . .	202
4.46 cs_compress.c . . . . .	202
4.47 csparse/Source/cs_counts.c File Reference . . . . .	202
4.47.1 Macro Definition Documentation . . . . .	202
4.47.1.1 HEAD . . . . .	203
4.47.1.2 NEXT . . . . .	203
4.47.2 Function Documentation . . . . .	203
4.47.2.1 cs_counts() . . . . .	203
4.48 cs_counts.c . . . . .	203
4.49 csparse/Source/cs_cumsum.c File Reference . . . . .	204
4.49.1 Function Documentation . . . . .	204
4.49.1.1 cs_cumsum() . . . . .	204
4.50 cs_cumsum.c . . . . .	205
4.51 csparse/Source/cs_dfs.c File Reference . . . . .	205
4.51.1 Function Documentation . . . . .	205
4.51.1.1 cs_dfs() . . . . .	205
4.52 cs_dfs.c . . . . .	206
4.53 csparse/Source/cs_dmperm.c File Reference . . . . .	206
4.53.1 Function Documentation . . . . .	206
4.53.1.1 cs_dmperm() . . . . .	206
4.54 cs_dmperm.c . . . . .	207
4.55 csparse/Source/cs_droptol.c File Reference . . . . .	208
4.55.1 Function Documentation . . . . .	208
4.55.1.1 cs_droptol() . . . . .	209
4.56 cs_droptol.c . . . . .	209

4.57 csparse/Source/cs_dropzeros.c File Reference . . . . .	209
4.57.1 Function Documentation . . . . .	209
4.57.1.1 cs_dropzeros() . . . . .	209
4.58 cs_dropzeros.c . . . . .	209
4.59 csparse/Source/cs_dupl.c File Reference . . . . .	210
4.59.1 Function Documentation . . . . .	210
4.59.1.1 cs_dupl() . . . . .	210
4.60 cs_dupl.c . . . . .	210
4.61 csparse/Source/cs_entry.c File Reference . . . . .	211
4.61.1 Function Documentation . . . . .	211
4.61.1.1 cs_entry() . . . . .	211
4.62 cs_entry.c . . . . .	211
4.63 csparse/Source/cs_ereach.c File Reference . . . . .	211
4.63.1 Function Documentation . . . . .	212
4.63.1.1 cs_ereach() . . . . .	212
4.64 cs_ereach.c . . . . .	212
4.65 csparse/Source/cs_etree.c File Reference . . . . .	212
4.65.1 Function Documentation . . . . .	212
4.65.1.1 cs_etree() . . . . .	213
4.66 cs_etree.c . . . . .	213
4.67 csparse/Source/cs_fkeep.c File Reference . . . . .	213
4.67.1 Function Documentation . . . . .	213
4.67.1.1 cs_fkeep() . . . . .	214
4.68 cs_fkeep.c . . . . .	214
4.69 csparse/Source/cs_gaxpy.c File Reference . . . . .	214
4.69.1 Function Documentation . . . . .	214
4.69.1.1 cs_gaxpy() . . . . .	214
4.70 cs_gaxpy.c . . . . .	215
4.71 csparse/Source/cs_happly.c File Reference . . . . .	215
4.71.1 Function Documentation . . . . .	215
4.71.1.1 cs_happly() . . . . .	215
4.72 cs_happly.c . . . . .	215
4.73 csparse/Source/cs_house.c File Reference . . . . .	216
4.73.1 Function Documentation . . . . .	216
4.73.1.1 cs_house() . . . . .	216
4.74 cs_house.c . . . . .	216
4.75 csparse/Source/cs_ipvec.c File Reference . . . . .	216
4.75.1 Function Documentation . . . . .	217
4.75.1.1 cs_ipvec() . . . . .	217
4.76 cs_ipvec.c . . . . .	217
4.77 csparse/Source/cs_leaf.c File Reference . . . . .	217
4.77.1 Function Documentation . . . . .	217

4.77.1.1 <a href="#">cs_leaf()</a> . . . . .	218
4.78 <a href="#">cs_leaf.c</a> . . . . .	218
4.79 <a href="#">csparse/Source/cs_load.c</a> File Reference . . . . .	218
4.79.1 Function Documentation . . . . .	218
4.79.1.1 <a href="#">cs_load()</a> . . . . .	218
4.80 <a href="#">cs_load.c</a> . . . . .	219
4.81 <a href="#">csparse/Source/cs_ksolve.c</a> File Reference . . . . .	219
4.81.1 Function Documentation . . . . .	219
4.81.1.1 <a href="#">cs_ksolve()</a> . . . . .	219
4.82 <a href="#">cs_ksolve.c</a> . . . . .	219
4.83 <a href="#">csparse/Source/cs_ksolve.c</a> File Reference . . . . .	220
4.83.1 Function Documentation . . . . .	220
4.83.1.1 <a href="#">cs_ksolve()</a> . . . . .	220
4.84 <a href="#">cs_ksolve.c</a> . . . . .	220
4.85 <a href="#">csparse/Source/cs_lu.c</a> File Reference . . . . .	220
4.85.1 Function Documentation . . . . .	221
4.85.1.1 <a href="#">cs_lu()</a> . . . . .	221
4.86 <a href="#">cs_lu.c</a> . . . . .	221
4.87 <a href="#">csparse/Source/cs_lusol.c</a> File Reference . . . . .	222
4.87.1 Function Documentation . . . . .	222
4.87.1.1 <a href="#">cs_lusol()</a> . . . . .	222
4.88 <a href="#">cs_lusol.c</a> . . . . .	223
4.89 <a href="#">csparse/Source/cs_malloc.c</a> File Reference . . . . .	223
4.89.1 Function Documentation . . . . .	223
4.89.1.1 <a href="#">cs_calloc()</a> . . . . .	223
4.89.1.2 <a href="#">cs_free()</a> . . . . .	224
4.89.1.3 <a href="#">cs_malloc()</a> . . . . .	224
4.89.1.4 <a href="#">cs_realloc()</a> . . . . .	224
4.90 <a href="#">cs_malloc.c</a> . . . . .	224
4.91 <a href="#">csparse/Source/cs_maxtrans.c</a> File Reference . . . . .	225
4.91.1 Function Documentation . . . . .	225
4.91.1.1 <a href="#">cs_maxtrans()</a> . . . . .	225
4.92 <a href="#">cs_maxtrans.c</a> . . . . .	225
4.93 <a href="#">csparse/Source/cs_multiply.c</a> File Reference . . . . .	226
4.93.1 Function Documentation . . . . .	226
4.93.1.1 <a href="#">cs_multiply()</a> . . . . .	227
4.94 <a href="#">cs_multiply.c</a> . . . . .	227
4.95 <a href="#">csparse/Source/cs_norm.c</a> File Reference . . . . .	227
4.95.1 Function Documentation . . . . .	227
4.95.1.1 <a href="#">cs_norm()</a> . . . . .	228
4.96 <a href="#">cs_norm.c</a> . . . . .	228
4.97 <a href="#">csparse/Source/cs_permute.c</a> File Reference . . . . .	228

4.97.1 Function Documentation	228
4.97.1.1 cs_permute()	228
4.98 cs_permute.c	229
4.99 csparse/Source/cs_pinv.c File Reference	229
4.99.1 Function Documentation	229
4.99.1.1 cs_pinv()	229
4.100 cs_pinv.c	229
4.101 csparse/Source/cs_post.c File Reference	230
4.101.1 Function Documentation	230
4.101.1.1 cs_post()	230
4.102 cs_post.c	230
4.103 csparse/Source/cs_print.c File Reference	230
4.103.1 Function Documentation	231
4.103.1.1 cs_print()	231
4.104 cs_print.c	231
4.105 csparse/Source/cs_pvec.c File Reference	231
4.105.1 Function Documentation	232
4.105.1.1 cs_pvec()	232
4.106 cs_pvec.c	232
4.107 csparse/Source/cs_qr.c File Reference	232
4.107.1 Function Documentation	232
4.107.1.1 cs_qr()	233
4.108 cs_qr.c	233
4.109 csparse/Source/cs_qrsol.c File Reference	234
4.109.1 Function Documentation	234
4.109.1.1 cs_qrsol()	234
4.110 cs_qrsol.c	234
4.111 csparse/Source/cs_randperm.c File Reference	235
4.111.1 Function Documentation	235
4.111.1.1 cs_randperm()	235
4.112 cs_randperm.c	236
4.113 csparse/Source/cs_reach.c File Reference	236
4.113.1 Function Documentation	236
4.113.1.1 cs_reach()	236
4.114 cs_reach.c	237
4.115 csparse/Source/cs_scatter.c File Reference	237
4.115.1 Function Documentation	237
4.115.1.1 cs_scatter()	237
4.116 cs_scatter.c	238
4.117 csparse/Source/cs_scc.c File Reference	238
4.117.1 Function Documentation	238
4.117.1.1 cs_scc()	238

4.118 cs_scc.c . . . . .	239
4.119 csparse/Source/cs_schol.c File Reference . . . . .	239
4.119.1 Function Documentation . . . . .	239
4.119.1.1 cs_schol() . . . . .	239
4.120 cs_schol.c . . . . .	240
4.121 csparse/Source/cs_spsolve.c File Reference . . . . .	240
4.121.1 Function Documentation . . . . .	240
4.121.1.1 cs_spsolve() . . . . .	240
4.122 cs_spsolve.c . . . . .	241
4.123 csparse/Source/cs_sqr.c File Reference . . . . .	241
4.123.1 Function Documentation . . . . .	241
4.123.1.1 cs_sqr() . . . . .	241
4.124 cs_sqr.c . . . . .	242
4.125 csparse/Source/cs_symperm.c File Reference . . . . .	243
4.125.1 Function Documentation . . . . .	243
4.125.1.1 cs_symperm() . . . . .	243
4.126 cs_symperm.c . . . . .	243
4.127 csparse/Source/cs_tdfs.c File Reference . . . . .	244
4.127.1 Function Documentation . . . . .	244
4.127.1.1 cs_tdfs() . . . . .	244
4.128 cs_tdfs.c . . . . .	244
4.129 csparse/Source/cs_transpose.c File Reference . . . . .	245
4.129.1 Function Documentation . . . . .	245
4.129.1.1 cs_transpose() . . . . .	245
4.130 cs_transpose.c . . . . .	245
4.131 csparse/Source/cs_updown.c File Reference . . . . .	245
4.131.1 Function Documentation . . . . .	246
4.131.1.1 cs_updown() . . . . .	246
4.132 cs_updown.c . . . . .	246
4.133 csparse/Source/cs_usolve.c File Reference . . . . .	246
4.133.1 Function Documentation . . . . .	247
4.133.1.1 cs_usolve() . . . . .	247
4.134 cs_usolve.c . . . . .	247
4.135 csparse/Source/cs_util.c File Reference . . . . .	247
4.135.1 Function Documentation . . . . .	248
4.135.1.1 cs_dalloc() . . . . .	248
4.135.1.2 cs_ddone() . . . . .	248
4.135.1.3 cs_dfree() . . . . .	248
4.135.1.4 cs_done() . . . . .	248
4.135.1.5 cs_idone() . . . . .	249
4.135.1.6 cs_ndone() . . . . .	249
4.135.1.7 cs_nfree() . . . . .	249

4.135.1.8 cs_sfree()	249
4.135.1.9 cs_salloc()	249
4.135.1.10 cs_spfree()	250
4.135.1.11 cs_sprealloc()	250
4.136 cs_util.c	250
4.137 csparse/Source/cs_utsolve.c File Reference	251
4.137.1 Function Documentation	252
4.137.1.1 cs_utsolve()	252
4.138 cs_utsolve.c	252
4.139 include/abip.h File Reference	252
4.139.1 Typedef Documentation	253
4.139.1.1 ABIPAdaptWork	253
4.139.1.2 ABIPCone	254
4.139.1.3 ABIPData	254
4.139.1.4 ABIPInfo	254
4.139.1.5 ABIPLinSysWork	254
4.139.1.6 ABIPMatrix	254
4.139.1.7 ABIPResiduals	254
4.139.1.8 ABIPSettings	255
4.139.1.9 ABIPSolution	255
4.139.1.10 ABIPWork	255
4.139.1.11 MKLlinsys	255
4.139.1.12 spe_problem	255
4.139.2 Enumeration Type Documentation	255
4.139.2.1 problem_type	255
4.139.3 Function Documentation	256
4.139.3.1 abip()	256
4.139.3.2 finish()	256
4.139.3.3 init()	256
4.139.3.4 main()	257
4.139.3.5 solve()	257
4.139.3.6 version()	257
4.140 abip.h	257
4.141 include/amatrix.h File Reference	260
4.142 amatrix.h	260
4.143 include/cones.h File Reference	261
4.143.1 Function Documentation	261
4.143.1.1 free_barrier_subproblem()	262
4.143.1.2 get_cone_dims()	262
4.143.1.3 get_cone_header()	262
4.143.1.4 positive_orthant_barrier_subproblem()	262
4.143.1.5 rsoc_barrier_subproblem()	263



4.143.1.6 soc_barrier_subproblem()	263
4.143.1.7 validate_cones()	263
4.143.1.8 zero_barrier_subproblem()	263
4.144 cones.h	264
4.145 include/ctrlc.h File Reference	264
4.145.1 Macro Definition Documentation	265
4.145.1.1 abip_end_interrupt_listener	265
4.145.1.2 abip_is_interrupted	265
4.145.1.3 abip_start_interrupt_listener	265
4.145.2 Typedef Documentation	265
4.145.2.1 abip_make_iso_compilers_happy	265
4.146 ctrlc.h	266
4.147 include/glbopts.h File Reference	266
4.147.1 Macro Definition Documentation	268
4.147.1.1 _abip_calloc	268
4.147.1.2 _abip_free	268
4.147.1.3 _abip_malloc	268
4.147.1.4 _abip_realloc	268
4.147.1.5 ABIP	268
4.147.1.6 abip_calloc	269
4.147.1.7 ABIP_FAILED	269
4.147.1.8 abip_free	269
4.147.1.9 ABIP_INDETERMINATE	269
4.147.1.10 ABIP_INFEASIBLE	269
4.147.1.11 ABIP_INFEASIBLE_INACCURATE	270
4.147.1.12 abip_malloc	270
4.147.1.13 ABIP_NULL	270
4.147.1.14 abip_printf	270
4.147.1.15 abip_realloc	270
4.147.1.16 ABIP_SIGINT	270
4.147.1.17 ABIP_SOLVED	271
4.147.1.18 ABIP_SOLVED_INACCURATE	271
4.147.1.19 ABIP_UNBOUNDED	271
4.147.1.20 ABIP_UNBOUNDED_INACCURATE	271
4.147.1.21 ABIP_UNFINISHED	271
4.147.1.22 ABIP_UNSOLVED	271
4.147.1.23 ABIP_VERSION	272
4.147.1.24 ABS	272
4.147.1.25 ADAPTIVE	272
4.147.1.26 ADAPTIVE_LOOKBACK	272
4.147.1.27 ALPHA	272
4.147.1.28 CG_BEST_TOL	272

4.147.1.29 CG_MIN_TOL . . . . .	273
4.147.1.30 CG_RATE . . . . .	273
4.147.1.31 CONE_TOL . . . . .	273
4.147.1.32 CONVERGED_INTERVAL . . . . .	273
4.147.1.33 DEBUG_FUNC . . . . .	273
4.147.1.34 EPS . . . . .	273
4.147.1.35 EPS_COR . . . . .	274
4.147.1.36 EPS_PEN . . . . .	274
4.147.1.37 EPS_TOL . . . . .	274
4.147.1.38 GAMMA . . . . .	274
4.147.1.39 INDETERMINATE_TOL . . . . .	274
4.147.1.40 INFINITY . . . . .	274
4.147.1.41 MAX . . . . .	275
4.147.1.42 MAX_ADMM_ITERS . . . . .	275
4.147.1.43 MAX_IPM_ITERS . . . . .	275
4.147.1.44 MIN . . . . .	275
4.147.1.45 NAN . . . . .	275
4.147.1.46 NORMALIZE . . . . .	275
4.147.1.47 POWF . . . . .	276
4.147.1.48 RETURN . . . . .	276
4.147.1.49 RHO_Y . . . . .	276
4.147.1.50 SAFEDIV_POS . . . . .	276
4.147.1.51 SCALE . . . . .	276
4.147.1.52 SIGMA . . . . .	276
4.147.1.53 SPARSITY_RATIO . . . . .	277
4.147.1.54 SQRTF . . . . .	277
4.147.1.55 VERBOSE . . . . .	277
4.147.1.56 WARM_START . . . . .	277
4.147.2 Typedef Documentation . . . . .	277
4.147.2.1 abip_float . . . . .	277
4.147.2.2 abip_int . . . . .	277
4.148 glbopts.h . . . . .	278
4.149 include/lasso_config.h File Reference . . . . .	280
4.149.1 Typedef Documentation . . . . .	281
4.149.1.1 lasso . . . . .	281
4.149.2 Function Documentation . . . . .	281
4.149.2.1 calc_lasso_residuals() . . . . .	281
4.149.2.2 free_lasso_linsys_work() . . . . .	281
4.149.2.3 init_lasso() . . . . .	281
4.149.2.4 init_lasso_linsys_work() . . . . .	282
4.149.2.5 lasso_A_times() . . . . .	282
4.149.2.6 lasso_AT_times() . . . . .	282

4.149.2.7	<a href="#">lasso_inner_conv_check()</a>	282
4.149.2.8	<a href="#">scaling_lasso_data()</a>	283
4.149.2.9	<a href="#">solve_lasso_linsys()</a>	283
4.149.2.10	<a href="#">un_scaling_lasso_sol()</a>	283
4.150	<a href="#">lasso_config.h</a>	283
4.151	<a href="#">include/linalg.h</a> File Reference	284
4.151.1	Macro Definition Documentation	285
4.151.1.1	<a href="#">ColMajor</a>	285
4.151.1.2	<a href="#">RowMajor</a>	286
4.151.2	Function Documentation	286
4.151.2.1	<a href="#">add_array()</a>	286
4.151.2.2	<a href="#">add_scaled_array()</a>	286
4.151.2.3	<a href="#">c_dot()</a>	286
4.151.2.4	<a href="#">cone_norm_1()</a>	287
4.151.2.5	<a href="#">csc_to_dense()</a>	287
4.151.2.6	<a href="#">dot()</a>	287
4.151.2.7	<a href="#">norm()</a>	287
4.151.2.8	<a href="#">norm_1()</a>	288
4.151.2.9	<a href="#">norm_diff()</a>	288
4.151.2.10	<a href="#">norm_inf()</a>	288
4.151.2.11	<a href="#">norm_inf_diff()</a>	288
4.151.2.12	<a href="#">norm_sq()</a>	289
4.151.2.13	<a href="#">scale_array()</a>	289
4.151.2.14	<a href="#">set_as_scaled_array()</a>	289
4.151.2.15	<a href="#">set_as_sq()</a>	289
4.151.2.16	<a href="#">set_as_sqrt()</a>	290
4.151.2.17	<a href="#">vec_mean()</a>	290
4.152	<a href="#">linalg.h</a>	290
4.153	<a href="#">include/linsys.h</a> File Reference	292
4.153.1	Function Documentation	292
4.153.1.1	<a href="#">accum_by_A()</a>	293
4.153.1.2	<a href="#">accum_by_Atrans()</a>	293
4.153.1.3	<a href="#">copy_A_matrix()</a>	293
4.153.1.4	<a href="#">free_A_matrix()</a>	293
4.153.1.5	<a href="#">free_linsys()</a>	294
4.153.1.6	<a href="#">get_lin_sys_method()</a>	294
4.153.1.7	<a href="#">get_lin_sys_summary()</a>	294
4.153.1.8	<a href="#">init_linsys_work()</a>	294
4.153.1.9	<a href="#">LDL_factor()</a>	294
4.153.1.10	<a href="#">solve_linsys()</a>	295
4.153.1.11	<a href="#">validate_lin_sys()</a>	295
4.154	<a href="#">linsys.h</a>	295

4.155 include/qcp_config.h File Reference . . . . .	297
4.155.1 Typedef Documentation . . . . .	297
4.155.1.1 qcp . . . . .	298
4.155.2 Function Documentation . . . . .	298
4.155.2.1 calc_qcp_residuals() . . . . .	298
4.155.2.2 free_qcp_linsys_work() . . . . .	298
4.155.2.3 init_qcp() . . . . .	298
4.155.2.4 init_qcp_linsys_work() . . . . .	299
4.155.2.5 qcp_A_times() . . . . .	299
4.155.2.6 qcp_AT_times() . . . . .	299
4.155.2.7 qcp_inner_conv_check() . . . . .	299
4.155.2.8 scaling_qcp_data() . . . . .	300
4.155.2.9 solve_qcp_linsys() . . . . .	300
4.155.2.10 un_scaling_qcp_sol() . . . . .	300
4.156 qcp_config.h . . . . .	300
4.157 include/svm_config.h File Reference . . . . .	301
4.157.1 Typedef Documentation . . . . .	302
4.157.1.1 svm . . . . .	302
4.157.2 Function Documentation . . . . .	302
4.157.2.1 calc_svm_residuals() . . . . .	303
4.157.2.2 free_svm_linsys_work() . . . . .	303
4.157.2.3 init_svm() . . . . .	303
4.157.2.4 init_svm_linsys_work() . . . . .	303
4.157.2.5 scaling_svm_data() . . . . .	304
4.157.2.6 solve_svm_linsys() . . . . .	304
4.157.2.7 svm_A_times() . . . . .	304
4.157.2.8 svm_AT_times() . . . . .	304
4.157.2.9 svm_inner_conv_check() . . . . .	305
4.157.2.10 un_scaling_svm_sol() . . . . .	305
4.158 svm_config.h . . . . .	305
4.159 include/svm_qp_config.h File Reference . . . . .	306
4.159.1 Typedef Documentation . . . . .	307
4.159.1.1 svmqp . . . . .	307
4.159.2 Function Documentation . . . . .	307
4.159.2.1 calc_svmqp_residuals() . . . . .	307
4.159.2.2 free_svmqp_linsys_work() . . . . .	308
4.159.2.3 init_svmqp() . . . . .	308
4.159.2.4 init_svmqp_linsys_work() . . . . .	308
4.159.2.5 scaling_svmqp_data() . . . . .	308
4.159.2.6 solve_svmqp_linsys() . . . . .	309
4.159.2.7 svmqp_A_times() . . . . .	309
4.159.2.8 svmqp_AT_times() . . . . .	309

4.159.2.9 svmqp_inner_conv_check()	309
4.159.2.10 un_scaling_svmqp_sol()	310
4.160 svm_qp_config.h	310
4.161 include/util.h File Reference	311
4.161.1 Function Documentation	311
4.161.1.1 ABIP()	311
4.161.1.2 free_cone()	312
4.161.1.3 free_data()	312
4.161.1.4 free_info()	312
4.161.1.5 free_sol()	312
4.161.1.6 print_array()	312
4.161.1.7 print_data()	313
4.161.1.8 print_work()	313
4.161.1.9 set_default_settings()	313
4.161.1.10 str_toc()	313
4.161.1.11 tic()	313
4.161.1.12 toc()	314
4.161.1.13 tocq()	314
4.162 util.h	314
4.163 make_abip_qcp.m File Reference	315
4.163.1 Function Documentation	316
4.163.1.1 addpath()	316
4.163.1.2 error()	316
4.163.1.3 eval()	316
4.163.1.4 fprintf() [1/6]	317
4.163.1.5 fprintf() [2/6]	317
4.163.1.6 fprintf() [3/6]	317
4.163.1.7 fprintf() [4/6]	317
4.163.1.8 fprintf() [5/6]	317
4.163.1.9 fprintf() [6/6]	318
4.163.1.10 if() [1/2]	318
4.163.1.11 if() [2/2]	318
4.163.2 Variable Documentation	318
4.163.2.1 alternatively	318
4.163.2.2 amd	318
4.163.2.3 amd_files	318
4.163.2.4 amd_include	319
4.163.2.5 amdlist	319
4.163.2.6 cs	319
4.163.2.7 cs_files	319
4.163.2.8 cs_include	319
4.163.2.9 cslist	319

4.163.2.10 debug	320
4.163.2.11 debugcommand	320
4.163.2.12 example	320
4.163.2.13 i	320
4.163.2.14 inc	320
4.163.2.15 intel64	320
4.163.2.16 ldI	321
4.163.2.17 ldI_files	321
4.163.2.18 ldI_include	321
4.163.2.19 ldIlist	321
4.163.2.20 lib_path	321
4.163.2.21 link	321
4.163.2.22 linux	322
4.163.2.23 mex_file	322
4.163.2.24 mex_type	322
4.163.2.25 mexcommand	322
4.163.2.26 mexfname	322
4.163.2.27 MKL	322
4.163.2.28 mkl_include	323
4.163.2.29 mkl_lib_path	323
4.163.2.30 MKLROOT	323
4.163.2.31 oneapi	323
4.163.2.32 pamd	323
4.163.2.33 pcs	323
4.163.2.34 pinc	324
4.163.2.35 platform	324
4.163.2.36 pldl	324
4.163.2.37 pmex	324
4.163.2.38 psrc	324
4.163.2.39 self	324
4.163.2.40 src	325
4.163.2.41 src_files	325
4.163.2.42 srclist	325
4.164 make_abip_qcp.m	325
4.165 mex/abip_ml_mex.c File Reference	327
4.165.1 Function Documentation	327
4.165.1.1 mexFunction()	327
4.166 abip_ml_mex.c	327
4.167 mex/abip_qcp_mex.c File Reference	332
4.167.1 Function Documentation	333
4.167.1.1 mexFunction()	333
4.168 abip_qcp_mex.c	333

4.169 qdldl/include/qdldl.h File Reference . . . . .	339
4.169.1 Function Documentation . . . . .	340
4.169.1.1 QDLDL_etree() . . . . .	340
4.169.1.2 QDLDL_factor() . . . . .	341
4.169.1.3 QDLDL_Lsolve() . . . . .	342
4.169.1.4 QDLDL_Ltsolve() . . . . .	342
4.169.1.5 QDLDL_solve() . . . . .	343
4.170 qdldl.h . . . . .	343
4.171 qdldl/include/qdldl_types.h File Reference . . . . .	344
4.171.1 Macro Definition Documentation . . . . .	344
4.171.1.1 QDLDL_INT_MAX . . . . .	344
4.171.2 Typedef Documentation . . . . .	344
4.171.2.1 QDLDL_bool . . . . .	345
4.171.2.2 QDLDL_float . . . . .	345
4.171.2.3 QDLDL_int . . . . .	345
4.172 qdldl_types.h . . . . .	345
4.173 qdldl/src/qdldl.c File Reference . . . . .	345
4.173.1 Macro Definition Documentation . . . . .	346
4.173.1.1 QDLDL_UNKNOWN . . . . .	346
4.173.1.2 QDLDL_UNUSED . . . . .	346
4.173.1.3 QDLDL_USED . . . . .	346
4.173.2 Function Documentation . . . . .	347
4.173.2.1 QDLDL_etree() . . . . .	347
4.173.2.2 QDLDL_factor() . . . . .	348
4.173.2.3 QDLDL_Lsolve() . . . . .	349
4.173.2.4 QDLDL_Ltsolve() . . . . .	349
4.173.2.5 QDLDL_solve() . . . . .	350
4.174 qdldl.c . . . . .	350
4.175 source/abip.c File Reference . . . . .	353
4.175.1 Macro Definition Documentation . . . . .	354
4.175.1.1 _CRT_SECURE_NO_WARNINGS . . . . .	354
4.175.2 Function Documentation . . . . .	354
4.175.2.1 abip() . . . . .	355
4.175.2.2 ABIP() . . . . .	355
4.175.2.3 adjust_barrier() . . . . .	355
4.175.2.4 finish() . . . . .	355
4.175.2.5 init() . . . . .	356
4.175.2.6 init_problem() . . . . .	356
4.175.2.7 solve() . . . . .	356
4.175.2.8 update_work() . . . . .	356
4.176 abip.c . . . . .	357
4.177 source/abip_version.c File Reference . . . . .	372

4.177.1 Function Documentation	372
4.177.1.1 version()	373
4.178 abip_version.c	373
4.179 source/cones.c File Reference	373
4.179.1 Macro Definition Documentation	373
4.179.1.1 _CRT_SECURE_NO_WARNINGS	374
4.179.2 Function Documentation	374
4.179.2.1 free_barrier_subproblem()	374
4.179.2.2 get_cone_dims()	374
4.179.2.3 get_cone_header()	374
4.179.2.4 positive_orthant_barrier_subproblem()	375
4.179.2.5 rsoc_barrier_subproblem()	375
4.179.2.6 soc_barrier_subproblem()	375
4.179.2.7 validate_cones()	375
4.179.2.8 zero_barrier_subproblem()	376
4.180 cones.c	376
4.181 source/ctrlc.c File Reference	379
4.182 ctrlc.c	379
4.183 source/lasso_config.c File Reference	380
4.183.1 Macro Definition Documentation	381
4.183.1.1 MAX_SCALE	381
4.183.1.2 MIN_SCALE	382
4.183.2 Function Documentation	382
4.183.2.1 calc_lasso_residuals()	382
4.183.2.2 form_lasso_kkt()	382
4.183.2.3 free_lasso_linsys_work()	382
4.183.2.4 get_lasso_pcg_tol()	383
4.183.2.5 get_unscaled_s()	383
4.183.2.6 get_unscaled_x()	383
4.183.2.7 get_unscaled_y()	383
4.183.2.8 init_lasso()	384
4.183.2.9 init_lasso_linsys_work()	384
4.183.2.10 init_lasso_precon()	384
4.183.2.11 lasso_A_times()	384
4.183.2.12 lasso_AT_times()	385
4.183.2.13 lasso_inner_conv_check()	385
4.183.2.14 scaling_lasso_data()	385
4.183.2.15 solve_lasso_linsys()	385
4.183.2.16 un_scaling_lasso_sol()	386
4.184 lasso_config.c	386
4.185 source/linalg.c File Reference	394
4.185.1 Function Documentation	394



4.185.1.1	<a href="#">add_array()</a>	395
4.185.1.2	<a href="#">add_scaled_array()</a>	395
4.185.1.3	<a href="#">arr_ind()</a>	395
4.185.1.4	<a href="#">c_dot()</a>	395
4.185.1.5	<a href="#">cone_norm_1()</a>	396
4.185.1.6	<a href="#">csc_to_dense()</a>	396
4.185.1.7	<a href="#">dot()</a>	396
4.185.1.8	<a href="#">norm()</a>	396
4.185.1.9	<a href="#">norm_1()</a>	397
4.185.1.10	<a href="#">norm_diff()</a>	397
4.185.1.11	<a href="#">norm_inf()</a>	397
4.185.1.12	<a href="#">norm_inf_diff()</a>	397
4.185.1.13	<a href="#">norm_sq()</a>	398
4.185.1.14	<a href="#">scale_array()</a>	398
4.185.1.15	<a href="#">set_as_scaled_array()</a>	398
4.185.1.16	<a href="#">set_as_sq()</a>	398
4.185.1.17	<a href="#">set_as_sqrt()</a>	399
4.185.1.18	<a href="#">vec_mean()</a>	399
4.186	<a href="#">linalg.c</a>	399
4.187	<a href="#">source/linsys.c File Reference</a>	402
4.187.1	<a href="#">Macro Definition Documentation</a>	403
4.187.1.1	<a href="#">_CRT_SECURE_NO_WARNINGS</a>	403
4.187.1.2	<a href="#">MAX_SCALE</a>	403
4.187.1.3	<a href="#">MIN_SCALE</a>	403
4.187.2	<a href="#">Function Documentation</a>	403
4.187.2.1	<a href="#">abip_cholsol()</a>	403
4.187.2.2	<a href="#">accum_by_A()</a>	404
4.187.2.3	<a href="#">accum_by_Atrans()</a>	404
4.187.2.4	<a href="#">copy_A_matrix()</a>	404
4.187.2.5	<a href="#">dense_chol_free()</a>	404
4.187.2.6	<a href="#">dense_chol_sol()</a>	405
4.187.2.7	<a href="#">free_A_matrix()</a>	405
4.187.2.8	<a href="#">free_linsys()</a>	405
4.187.2.9	<a href="#">get_lin_sys_method()</a>	405
4.187.2.10	<a href="#">get_lin_sys_summary()</a>	405
4.187.2.11	<a href="#">init_dense_chol()</a>	406
4.187.2.12	<a href="#">init_linsys_work()</a>	406
4.187.2.13	<a href="#">init_mkl_work()</a>	406
4.187.2.14	<a href="#">init_pardiso()</a>	406
4.187.2.15	<a href="#">LDL_factor()</a>	406
4.187.2.16	<a href="#">mkl_solve_linsys()</a>	407
4.187.2.17	<a href="#">pardiso_free()</a>	407

4.187.2.18 pardiso_solve()	407
4.187.2.19 pcg()	407
4.187.2.20 permute_kkt()	407
4.187.2.21 qcp_pcg()	408
4.187.2.22 solve_linsys()	408
4.187.2.23 svmqp_pcg()	408
4.187.2.24 validate_lin_sys()	408
4.188 linsys.c	409
4.189 source/qcp_config.c File Reference	422
4.189.1 Macro Definition Documentation	423
4.189.1.1 MAX_SCALE	423
4.189.1.2 MIN_SCALE	423
4.189.2 Function Documentation	423
4.189.2.1 calc_qcp_residuals()	423
4.189.2.2 form_qcp_kkt()	424
4.189.2.3 free_qcp_linsys_work()	424
4.189.2.4 get_qcp_pcg_tol()	424
4.189.2.5 init_qcp()	424
4.189.2.6 init_qcp_linsys_work()	425
4.189.2.7 init_qcp_precon()	425
4.189.2.8 qcp_A_times()	425
4.189.2.9 qcp_AT_times()	425
4.189.2.10 qcp_inner_conv_check()	426
4.189.2.11 scaling_qcp_data()	426
4.189.2.12 solve_qcp_linsys()	426
4.189.2.13 un_scaling_qcp_sol()	426
4.190 qcp_config.c	427
4.191 source/svm_config.c File Reference	436
4.191.1 Macro Definition Documentation	437
4.191.1.1 MAX_SCALE	437
4.191.1.2 MIN_SCALE	437
4.191.2 Function Documentation	438
4.191.2.1 calc_svm_residuals()	438
4.191.2.2 form_svm_kkt()	438
4.191.2.3 free_svm_linsys_work()	438
4.191.2.4 get_svm_pcg_tol()	438
4.191.2.5 init_svm()	439
4.191.2.6 init_svm_linsys_work()	439
4.191.2.7 init_svm_precon()	439
4.191.2.8 scaling_svm_data()	439
4.191.2.9 solve_svm_linsys()	440
4.191.2.10 svm_A_times()	440

4.191.2.11 svm_AT_times()	440
4.191.2.12 svm_inner_conv_check()	440
4.191.2.13 un_scaling_svm_sol()	441
4.192 svm_config.c	441
4.193 source/svm_qp_config.c File Reference	450
4.193.1 Macro Definition Documentation	451
4.193.1.1 MAX_SCALE	451
4.193.1.2 MIN_SCALE	451
4.193.2 Function Documentation	451
4.193.2.1 calc_svmqp_residuals()	451
4.193.2.2 form_svmqp_kkt()	451
4.193.2.3 free_svmqp_linsys_work()	452
4.193.2.4 get_svmqp_pcg_tol()	452
4.193.2.5 init_svmqp()	452
4.193.2.6 init_svmqp_linsys_work()	452
4.193.2.7 init_svmqp_precon()	453
4.193.2.8 scaling_svmqp_data()	453
4.193.2.9 solve_svmqp_linsys()	453
4.193.2.10 svmqp_A_times()	453
4.193.2.11 svmqp_AT_times()	454
4.193.2.12 svmqp_inner_conv_check()	454
4.193.2.13 un_scaling_svmqp_sol()	454
4.194 svm_qp_config.c	454
4.195 source/util.c File Reference	465
4.195.1 Macro Definition Documentation	466
4.195.1.1 _CRT_SECURE_NO_WARNINGS	466
4.195.2 Function Documentation	466
4.195.2.1 free_cone()	466
4.195.2.2 free_data()	466
4.195.2.3 free_info()	467
4.195.2.4 free_sol()	467
4.195.2.5 print_array()	467
4.195.2.6 print_data()	467
4.195.2.7 print_work()	467
4.195.2.8 set_default_settings()	468
4.195.2.9 str_toc()	468
4.195.2.10 tic()	468
4.195.2.11 toc()	468
4.195.2.12 tocq()	468
4.196 util.c	469



# Chapter 1

## Class Index

### 1.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

ABIP_A_DATA_MATRIX	7
ABIP_CONE	8
ABIP_INFO	10
ABIP_LIN_SYS_WORK	13
ABIP_PROBLEM_DATA	17
ABIP_RESIDUALS	19
ABIP_SETTINGS	23
ABIP_SOL_VARS	29
ABIP_WORK	30
cs_dmperm_results	33
cs_numeric	35
cs_sparse	36
cs_symbolic	38
Lasso	40
qcp	46
solve_specific_problem	51
SuiteSparse_config_struct	56
Svm	58
SVMqp	66



## Chapter 2

# File Index

### 2.1 File List

Here is a list of all files with brief descriptions:

<a href="#">make_abip_qcp.m</a>	315
<a href="#">amd/amd.h</a>	73
<a href="#">amd/amd_1.c</a>	88
<a href="#">amd/amd_2.c</a>	91
<a href="#">amd/amd_aat.c</a>	115
<a href="#">amd/amd_control.c</a>	118
<a href="#">amd/amd_defaults.c</a>	119
<a href="#">amd/amd_dump.c</a>	120
<a href="#">amd/amd_global.c</a>	124
<a href="#">amd/amd_info.c</a>	127
<a href="#">amd/amd_internal.h</a>	129
<a href="#">amd/amd_order.c</a>	142
<a href="#">amd/amd_post_tree.c</a>	145
<a href="#">amd/amd_postorder.c</a>	147
<a href="#">amd/amd_preprocess.c</a>	151
<a href="#">amd/amd_valid.c</a>	153
<a href="#">amd/SuiteSparse_config.c</a>	155
<a href="#">amd/SuiteSparse_config.h</a>	164
<a href="#">csparse/Include/cs.h</a>	173
<a href="#">csparse/Source/cs_add.c</a>	193
<a href="#">csparse/Source/cs_amd.c</a>	194
<a href="#">csparse/Source/cs_chol.c</a>	199
<a href="#">csparse/Source/cs_cholsol.c</a>	200
<a href="#">csparse/Source/cs_compress.c</a>	201
<a href="#">csparse/Source/cs_counts.c</a>	202
<a href="#">csparse/Source/cs_cumsum.c</a>	204
<a href="#">csparse/Source/cs_dfs.c</a>	205
<a href="#">csparse/Source/cs_dmperm.c</a>	206
<a href="#">csparse/Source/cs_droptol.c</a>	208
<a href="#">csparse/Source/cs_dropzeros.c</a>	209
<a href="#">csparse/Source/cs_dupl.c</a>	210
<a href="#">csparse/Source/cs_entry.c</a>	211
<a href="#">csparse/Source/cs_ereach.c</a>	211
<a href="#">csparse/Source/cs_etree.c</a>	212
<a href="#">csparse/Source/cs_fkeep.c</a>	213

csparse/Source/cs_gaxpy.c	214
csparse/Source/cs_happly.c	215
csparse/Source/cs_house.c	216
csparse/Source/cs_ipvec.c	216
csparse/Source/cs_leaf.c	217
csparse/Source/cs_load.c	218
csparse/Source/cs_lsolve.c	219
csparse/Source/cs_ltsolve.c	220
csparse/Source/cs_lu.c	220
csparse/Source/cs_lusol.c	222
csparse/Source/cs_malloc.c	223
csparse/Source/cs_maxtrans.c	225
csparse/Source/cs_multiply.c	226
csparse/Source/cs_norm.c	227
csparse/Source/cs_permute.c	228
csparse/Source/cs_pinv.c	229
csparse/Source/cs_post.c	230
csparse/Source/cs_print.c	230
csparse/Source/cs_pvec.c	231
csparse/Source/cs_qr.c	232
csparse/Source/cs_qrsol.c	234
csparse/Source/cs_randperm.c	235
csparse/Source/cs_reach.c	236
csparse/Source/cs_scatter.c	237
csparse/Source/cs_scc.c	238
csparse/Source/cs_schol.c	239
csparse/Source/cs_spsolve.c	240
csparse/Source/cs_sqr.c	241
csparse/Source/cs_symperm.c	243
csparse/Source/cs_tdfs.c	244
csparse/Source/cs_transpose.c	245
csparse/Source/cs_updown.c	245
csparse/Source/cs_usolve.c	246
csparse/Source/cs_util.c	247
csparse/Source/cs_utsolve.c	251
include/abip.h	252
include/amatrix.h	260
include/cones.h	261
include/ctrlc.h	264
include/glbopts.h	266
include/lasso_config.h	280
include/linalg.h	284
include/linsys.h	292
include/qcp_config.h	297
include/svm_config.h	301
include/svm_qp_config.h	306
include/util.h	311
mex/abip_ml_mex.c	327
mex/abip_qcp_mex.c	332
qldl/include/qldl.h	339
qldl/include/qldl_types.h	344
qldl/src/qldl.c	345
source/abip.c	353
source/abip_version.c	372
source/cones.c	373
source/ctrlc.c	379
source/lasso_config.c	380
source/linalg.c	394



source/ <a href="#">linsys.c</a> . . . . .	402
source/ <a href="#">qcp_config.c</a> . . . . .	422
source/ <a href="#">svm_config.c</a> . . . . .	436
source/ <a href="#">svm_qp_config.c</a> . . . . .	450
source/ <a href="#">util.c</a> . . . . .	465



## Chapter 3

# Class Documentation

### 3.1 ABIP\_A\_DATA\_MATRIX Struct Reference

```
#include <amatrix.h>
```

#### Public Attributes

- [abip\\_float](#) \* x
- [abip\\_int](#) \* i
- [abip\\_int](#) \* p
- [abip\\_int](#) m
- [abip\\_int](#) n

#### 3.1.1 Detailed Description

Definition at line 11 of file [amatrix.h](#).

#### 3.1.2 Member Data Documentation

##### 3.1.2.1 i

```
abip\_int* ABIP_A_DATA_MATRIX::i
```

Definition at line 14 of file [amatrix.h](#).

### 3.1.2.2 m

```
abip_int ABIP_A_DATA_MATRIX::m
```

Definition at line 16 of file [amatrix.h](#).

### 3.1.2.3 n

```
abip_int ABIP_A_DATA_MATRIX::n
```

Definition at line 17 of file [amatrix.h](#).

### 3.1.2.4 p

```
abip_int* ABIP_A_DATA_MATRIX::p
```

Definition at line 15 of file [amatrix.h](#).

### 3.1.2.5 x

```
abip_float* ABIP_A_DATA_MATRIX::x
```

Definition at line 13 of file [amatrix.h](#).

The documentation for this struct was generated from the following file:

- include/[amatrix.h](#)

## 3.2 ABIP\_CONE Struct Reference

```
#include <abip.h>
```

### Public Attributes

- [abip\\_int](#) \* q
- [abip\\_int](#) qsize
- [abip\\_int](#) \* rq
- [abip\\_int](#) rqsize
- [abip\\_int](#) f
- [abip\\_int](#) z
- [abip\\_int](#) l

### 3.2.1 Detailed Description

Definition at line 67 of file [abip.h](#).

### 3.2.2 Member Data Documentation

#### 3.2.2.1 f

```
abip_int ABIP_CONE::f
```

Definition at line 72 of file [abip.h](#).

#### 3.2.2.2 l

```
abip_int ABIP_CONE::l
```

Definition at line 74 of file [abip.h](#).

#### 3.2.2.3 q

```
abip_int* ABIP_CONE::q
```

Definition at line 68 of file [abip.h](#).

#### 3.2.2.4 qsize

```
abip_int ABIP_CONE::qsize
```

Definition at line 69 of file [abip.h](#).

#### 3.2.2.5 rq

```
abip_int* ABIP_CONE::rq
```

Definition at line 70 of file [abip.h](#).

### 3.2.2.6 rsize

```
abip_int ABIP_CONE::rsize
```

Definition at line 71 of file [abip.h](#).

### 3.2.2.7 z

```
abip_int ABIP_CONE::z
```

Definition at line 73 of file [abip.h](#).

The documentation for this struct was generated from the following file:

- [include/abip.h](#)

## 3.3 ABIP\_INFO Struct Reference

```
#include <abip.h>
```

### Public Attributes

- char [status](#) [32]
- [abip\\_int](#) [status\\_val](#)
- [abip\\_int](#) [ipm\\_iter](#)
- [abip\\_int](#) [admm\\_iter](#)
- [abip\\_float](#) [pobj](#)
- [abip\\_float](#) [dobj](#)
- [abip\\_float](#) [res\\_pri](#)
- [abip\\_float](#) [res\\_dual](#)
- [abip\\_float](#) [rel\\_gap](#)
- [abip\\_float](#) [res\\_infeas](#)
- [abip\\_float](#) [res\\_unbdd](#)
- [abip\\_float](#) [setup\\_time](#)
- [abip\\_float](#) [solve\\_time](#)
- [abip\\_float](#) [avg\\_linsys\\_time](#)
- [abip\\_float](#) [avg\\_cg\\_iters](#)

### 3.3.1 Detailed Description

Definition at line 139 of file [abip.h](#).

### 3.3.2 Member Data Documentation

### 3.3.2.1 admm\_iter

`abip_int ABIP_INFO::admm_iter`

Definition at line 144 of file [abip.h](#).

### 3.3.2.2 avg\_cg\_iters

`abip_float ABIP_INFO::avg_cg_iters`

Definition at line 157 of file [abip.h](#).

### 3.3.2.3 avg\_linsys\_time

`abip_float ABIP_INFO::avg_linsys_time`

Definition at line 156 of file [abip.h](#).

### 3.3.2.4 dobj

`abip_float ABIP_INFO::dobj`

Definition at line 147 of file [abip.h](#).

### 3.3.2.5 ipm\_iter

`abip_int ABIP_INFO::ipm_iter`

Definition at line 143 of file [abip.h](#).

### 3.3.2.6 pobj

`abip_float ABIP_INFO::pobj`

Definition at line 146 of file [abip.h](#).

### 3.3.2.7 rel\_gap

`abip_float ABIP_INFO::rel_gap`

Definition at line 150 of file [abip.h](#).

### 3.3.2.8 res\_dual

`abip_float ABIP_INFO::res_dual`

Definition at line 149 of file [abip.h](#).

### 3.3.2.9 res\_infeas

`abip_float ABIP_INFO::res_infeas`

Definition at line 151 of file [abip.h](#).

### 3.3.2.10 res\_pri

`abip_float ABIP_INFO::res_pri`

Definition at line 148 of file [abip.h](#).

### 3.3.2.11 res\_unbdd

`abip_float ABIP_INFO::res_unbdd`

Definition at line 152 of file [abip.h](#).

### 3.3.2.12 setup\_time

`abip_float ABIP_INFO::setup_time`

Definition at line 154 of file [abip.h](#).



**3.3.2.13 solve\_time**

`abip_float` ABIP\_INFO::solve\_time

Definition at line 155 of file [abip.h](#).

**3.3.2.14 status**

`char` ABIP\_INFO::status[32]

Definition at line 141 of file [abip.h](#).

**3.3.2.15 status\_val**

`abip_int` ABIP\_INFO::status\_val

Definition at line 142 of file [abip.h](#).

The documentation for this struct was generated from the following file:

- [include/abip.h](#)

**3.4 ABIP\_LIN\_SYS\_WORK Struct Reference**

```
#include <linsys.h>
```

**Public Attributes**

- [abip\\_float](#) total\_solve\_time
- `cs` \* `K`
- `_MKL_DSS_HANDLE_t` [handle](#)
- `void` \* [pt](#) [64]
- `MKL_INT` [iparm](#) [64]
- `MKL_INT` [maxfct](#)
- `MKL_INT` [mnum](#)
- `MKL_INT` [error](#)
- `MKL_INT` [msglvl](#)
- [abip\\_float](#) [ddum](#)
- `MKL_INT` [idum](#)
- `MKL_INT` [mtype](#)
- [abip\\_float](#) \* `M`
- [abip\\_int](#) total\_cg\_iters
- `css` \* `S`
- `csn` \* `N`
- `cs` \* `L`
- [abip\\_float](#) \* `Dinv`
- [abip\\_int](#) nnz\_LDL
- [abip\\_int](#) \* `P`
- [abip\\_float](#) \* `bp`
- [abip\\_float](#) \* `U`

### 3.4.1 Detailed Description

Definition at line 23 of file [linsys.h](#).

### 3.4.2 Member Data Documentation

#### 3.4.2.1 bp

```
abip_float* ABIP_LIN_SYS_WORK::bp
```

Definition at line 59 of file [linsys.h](#).

#### 3.4.2.2 ddum

```
abip_float ABIP_LIN_SYS_WORK::ddum
```

Definition at line 38 of file [linsys.h](#).

#### 3.4.2.3 Dinv

```
abip_float* ABIP_LIN_SYS_WORK::Dinv
```

Definition at line 56 of file [linsys.h](#).

#### 3.4.2.4 error

```
MKL_INT ABIP_LIN_SYS_WORK::error
```

Definition at line 37 of file [linsys.h](#).

#### 3.4.2.5 handle

```
_MKL_DSS_HANDLE_t ABIP_LIN_SYS_WORK::handle
```

Definition at line 31 of file [linsys.h](#).

#### 3.4.2.6 idum

```
MKL_INT ABIP_LIN_SYS_WORK::idum
```

Definition at line 39 of file [linsys.h](#).

#### 3.4.2.7 iparm

```
MKL_INT ABIP_LIN_SYS_WORK::iparm[64]
```

Definition at line 36 of file [linsys.h](#).

#### 3.4.2.8 K

```
cs* ABIP_LIN_SYS_WORK::K
```

Definition at line 28 of file [linsys.h](#).

#### 3.4.2.9 L

```
cs* ABIP_LIN_SYS_WORK::L
```

Definition at line 55 of file [linsys.h](#).

#### 3.4.2.10 M

```
abip_float* ABIP_LIN_SYS_WORK::M
```

Definition at line 45 of file [linsys.h](#).

#### 3.4.2.11 maxfct

```
MKL_INT ABIP_LIN_SYS_WORK::maxfct
```

Definition at line 37 of file [linsys.h](#).

#### 3.4.2.12 mnum

```
MKL_INT ABIP_LIN_SYS_WORK::mnum
```

Definition at line 37 of file [linsys.h](#).

#### 3.4.2.13 msglvl

```
MKL_INT ABIP_LIN_SYS_WORK::msglvl
```

Definition at line 37 of file [linsys.h](#).

#### 3.4.2.14 mtype

```
MKL_INT ABIP_LIN_SYS_WORK::mtype
```

Definition at line 40 of file [linsys.h](#).

#### 3.4.2.15 N

```
csn* ABIP_LIN_SYS_WORK::N
```

Definition at line 51 of file [linsys.h](#).

#### 3.4.2.16 nnz\_LDL

```
abip_int ABIP_LIN_SYS_WORK::nnz_LDL
```

Definition at line 57 of file [linsys.h](#).

#### 3.4.2.17 P

```
abip_int* ABIP_LIN_SYS_WORK::P
```

Definition at line 58 of file [linsys.h](#).

#### 3.4.2.18 pt

```
void* ABIP_LIN_SYS_WORK::pt[64]
```

Definition at line 35 of file [linsys.h](#).

#### 3.4.2.19 S

```
css* ABIP_LIN_SYS_WORK::S
```

Definition at line 50 of file [linsys.h](#).

#### 3.4.2.20 total\_cg\_iters

```
abip_int ABIP_LIN_SYS_WORK::total_cg_iters
```

Definition at line 46 of file [linsys.h](#).

#### 3.4.2.21 total\_solve\_time

```
abip_float ABIP_LIN_SYS_WORK::total_solve_time
```

Definition at line 25 of file [linsys.h](#).

#### 3.4.2.22 U

```
abip_float* ABIP_LIN_SYS_WORK::U
```

Definition at line 64 of file [linsys.h](#).

The documentation for this struct was generated from the following file:

- [include/linsys.h](#)

## 3.5 ABIP\_PROBLEM\_DATA Struct Reference

```
#include <abip.h>
```

## Public Attributes

- [abip\\_int](#) m
- [abip\\_int](#) n
- [ABIPMatrix](#) \* A
- [ABIPMatrix](#) \* Q
- [abip\\_float](#) \* b
- [abip\\_float](#) \* c
- [abip\\_float](#) lambda
- [ABIPSettings](#) \* stgs

### 3.5.1 Detailed Description

Definition at line 79 of file [abip.h](#).

### 3.5.2 Member Data Documentation

#### 3.5.2.1 A

[ABIPMatrix](#)\* ABIP\_PROBLEM\_DATA::A

Definition at line 83 of file [abip.h](#).

#### 3.5.2.2 b

[abip\\_float](#)\* ABIP\_PROBLEM\_DATA::b

Definition at line 86 of file [abip.h](#).

#### 3.5.2.3 c

[abip\\_float](#)\* ABIP\_PROBLEM\_DATA::c

Definition at line 87 of file [abip.h](#).

#### 3.5.2.4 lambda

[abip\\_float](#) ABIP\_PROBLEM\_DATA::lambda

Definition at line 89 of file [abip.h](#).

#### 3.5.2.5 m

```
abip_int ABIP_PROBLEM_DATA::m
```

Definition at line 81 of file [abip.h](#).

#### 3.5.2.6 n

```
abip_int ABIP_PROBLEM_DATA::n
```

Definition at line 82 of file [abip.h](#).

#### 3.5.2.7 Q

```
ABIPMatrix* ABIP_PROBLEM_DATA::Q
```

Definition at line 84 of file [abip.h](#).

#### 3.5.2.8 stgs

```
ABIPSettings* ABIP_PROBLEM_DATA::stgs
```

Definition at line 90 of file [abip.h](#).

The documentation for this struct was generated from the following file:

- [include/abip.h](#)

## 3.6 ABIP\_RESIDUALS Struct Reference

```
#include <abip.h>
```

## Public Attributes

- [abip\\_int last\\_ipm\\_iter](#)
- [abip\\_int last\\_admm\\_iter](#)
- [abip\\_float last\\_mu](#)
- [abip\\_float res\\_pri](#)
- [abip\\_float res\\_dual](#)
- [abip\\_float rel\\_gap](#)
- [abip\\_float res\\_infeas](#)
- [abip\\_float res\\_unbdd](#)
- [abip\\_float ct\\_x\\_by\\_tau](#)
- [abip\\_float bt\\_y\\_by\\_tau](#)
- [abip\\_float pobj](#)
- [abip\\_float dobj](#)
- [abip\\_float tau](#)
- [abip\\_float kap](#)
- [abip\\_float res\\_dif](#)
- [abip\\_float error\\_ratio](#)
- [abip\\_float Ax\\_b\\_norm](#)
- [abip\\_float Qx\\_ATy\\_c\\_s\\_norm](#)

### 3.6.1 Detailed Description

Definition at line 182 of file [abip.h](#).

### 3.6.2 Member Data Documentation

#### 3.6.2.1 Ax\_b\_norm

[abip\\_float](#) ABIP\_RESIDUALS::Ax\_b\_norm

Definition at line 206 of file [abip.h](#).

#### 3.6.2.2 bt\_y\_by\_tau

[abip\\_float](#) ABIP\_RESIDUALS::bt\_y\_by\_tau

Definition at line 195 of file [abip.h](#).



### 3.6.2.3 ct\_x\_by\_tau

`abip_float ABIP_RESIDUALS::ct_x_by_tau`

Definition at line 194 of file [abip.h](#).

### 3.6.2.4 dobj

`abip_float ABIP_RESIDUALS::dobj`

Definition at line 198 of file [abip.h](#).

### 3.6.2.5 error\_ratio

`abip_float ABIP_RESIDUALS::error_ratio`

Definition at line 204 of file [abip.h](#).

### 3.6.2.6 kap

`abip_float ABIP_RESIDUALS::kap`

Definition at line 201 of file [abip.h](#).

### 3.6.2.7 last\_admm\_iter

`abip_int ABIP_RESIDUALS::last_admm_iter`

Definition at line 185 of file [abip.h](#).

### 3.6.2.8 last\_ipm\_iter

`abip_int ABIP_RESIDUALS::last_ipm_iter`

Definition at line 184 of file [abip.h](#).

### 3.6.2.9 last\_mu

`abip_float ABIP_RESIDUALS::last_mu`

Definition at line 186 of file [abip.h](#).

### 3.6.2.10 pobj

`abip_float ABIP_RESIDUALS::pobj`

Definition at line 197 of file [abip.h](#).

### 3.6.2.11 Qx\_ATy\_c\_s\_norm

`abip_float ABIP_RESIDUALS::Qx_ATy_c_s_norm`

Definition at line 207 of file [abip.h](#).

### 3.6.2.12 rel\_gap

`abip_float ABIP_RESIDUALS::rel_gap`

Definition at line 190 of file [abip.h](#).

### 3.6.2.13 res\_dif

`abip_float ABIP_RESIDUALS::res_dif`

Definition at line 203 of file [abip.h](#).

### 3.6.2.14 res\_dual

`abip_float ABIP_RESIDUALS::res_dual`

Definition at line 189 of file [abip.h](#).

#### 3.6.2.15 res\_infeas

`abip_float` ABIP\_RESIDUALS::res\_infeas

Definition at line 191 of file [abip.h](#).

#### 3.6.2.16 res\_pri

`abip_float` ABIP\_RESIDUALS::res\_pri

Definition at line 188 of file [abip.h](#).

#### 3.6.2.17 res\_unbdd

`abip_float` ABIP\_RESIDUALS::res\_unbdd

Definition at line 192 of file [abip.h](#).

#### 3.6.2.18 tau

`abip_float` ABIP\_RESIDUALS::tau

Definition at line 200 of file [abip.h](#).

The documentation for this struct was generated from the following file:

- [include/abip.h](#)

## 3.7 ABIP\_SETTINGS Struct Reference

```
#include <abip.h>
```

## Public Attributes

- [abip\\_int normalize](#)
- [abip\\_int scale\\_E](#)
- [abip\\_int scale\\_bc](#)
- [abip\\_float scale](#)
- [abip\\_float rho\\_x](#)
- [abip\\_float rho\\_y](#)
- [abip\\_float rho\\_tau](#)
- [abip\\_int max\\_ipm\\_iters](#)
- [abip\\_int max\\_admm\\_iters](#)
- [abip\\_float eps](#)
- [abip\\_float eps\\_p](#)
- [abip\\_float eps\\_d](#)
- [abip\\_float eps\\_g](#)
- [abip\\_float eps\\_inf](#)
- [abip\\_float eps\\_unb](#)
- [abip\\_float err\\_dif](#)
- [abip\\_float alpha](#)
- [abip\\_float cg\\_rate](#)
- [abip\\_int use\\_indirect](#)
- [abip\\_int inner\\_check\\_period](#)
- [abip\\_int outer\\_check\\_period](#)
- [abip\\_int verbose](#)
- [abip\\_int linsys\\_solver](#)
- [abip\\_int prob\\_type](#)
- [abip\\_float time\\_limit](#)
- [abip\\_float psi](#)
- [abip\\_int origin\\_scaling](#)
- [abip\\_int ruiz\\_scaling](#)
- [abip\\_int pc\\_scaling](#)

### 3.7.1 Detailed Description

Definition at line 93 of file [abip.h](#).

### 3.7.2 Member Data Documentation

#### 3.7.2.1 alpha

`abip_float ABIP_SETTINGS::alpha`

Definition at line 113 of file [abip.h](#).

### 3.7.2.2 cg\_rate

`abip_float ABIP_SETTINGS::cg_rate`

Definition at line 114 of file [abip.h](#).

### 3.7.2.3 eps

`abip_float ABIP_SETTINGS::eps`

Definition at line 105 of file [abip.h](#).

### 3.7.2.4 eps\_d

`abip_float ABIP_SETTINGS::eps_d`

Definition at line 107 of file [abip.h](#).

### 3.7.2.5 eps\_g

`abip_float ABIP_SETTINGS::eps_g`

Definition at line 108 of file [abip.h](#).

### 3.7.2.6 eps\_inf

`abip_float ABIP_SETTINGS::eps_inf`

Definition at line 109 of file [abip.h](#).

### 3.7.2.7 eps\_p

`abip_float ABIP_SETTINGS::eps_p`

Definition at line 106 of file [abip.h](#).

### 3.7.2.8 eps\_unb

`abip_float ABIP_SETTINGS::eps_unb`

Definition at line 110 of file [abip.h](#).

### 3.7.2.9 err\_dif

`abip_float ABIP_SETTINGS::err_dif`

Definition at line 112 of file [abip.h](#).

### 3.7.2.10 inner\_check\_period

`abip_int ABIP_SETTINGS::inner_check_period`

Definition at line 117 of file [abip.h](#).

### 3.7.2.11 linsys\_solver

`abip_int ABIP_SETTINGS::linsys_solver`

Definition at line 121 of file [abip.h](#).

### 3.7.2.12 max\_admm\_iters

`abip_int ABIP_SETTINGS::max_admm_iters`

Definition at line 104 of file [abip.h](#).

### 3.7.2.13 max\_ipm\_iters

`abip_int ABIP_SETTINGS::max_ipm_iters`

Definition at line 103 of file [abip.h](#).

#### 3.7.2.14 normalize

`abip_int ABIP_SETTINGS::normalize`

Definition at line 95 of file [abip.h](#).

#### 3.7.2.15 origin\_scaling

`abip_int ABIP_SETTINGS::origin_scaling`

Definition at line 126 of file [abip.h](#).

#### 3.7.2.16 outer\_check\_period

`abip_int ABIP_SETTINGS::outer_check_period`

Definition at line 118 of file [abip.h](#).

#### 3.7.2.17 pc\_scaling

`abip_int ABIP_SETTINGS::pc_scaling`

Definition at line 128 of file [abip.h](#).

#### 3.7.2.18 prob\_type

`abip_int ABIP_SETTINGS::prob_type`

Definition at line 122 of file [abip.h](#).

#### 3.7.2.19 psi

`abip_float ABIP_SETTINGS::psi`

Definition at line 124 of file [abip.h](#).

### 3.7.2.20 rho\_tau

`abip_float ABIP_SETTINGS::rho_tau`

Definition at line 101 of file [abip.h](#).

### 3.7.2.21 rho\_x

`abip_float ABIP_SETTINGS::rho_x`

Definition at line 99 of file [abip.h](#).

### 3.7.2.22 rho\_y

`abip_float ABIP_SETTINGS::rho_y`

Definition at line 100 of file [abip.h](#).

### 3.7.2.23 ruiz\_scaling

`abip_int ABIP_SETTINGS::ruiz_scaling`

Definition at line 127 of file [abip.h](#).

### 3.7.2.24 scale

`abip_float ABIP_SETTINGS::scale`

Definition at line 98 of file [abip.h](#).

### 3.7.2.25 scale\_bc

`abip_int ABIP_SETTINGS::scale_bc`

Definition at line 97 of file [abip.h](#).



### 3.7.2.26 scale\_E

`abip_int` ABIP\_SETTINGS::scale\_E

Definition at line 96 of file [abip.h](#).

### 3.7.2.27 time\_limit

`abip_float` ABIP\_SETTINGS::time\_limit

Definition at line 123 of file [abip.h](#).

### 3.7.2.28 use\_indirect

`abip_int` ABIP\_SETTINGS::use\_indirect

Definition at line 116 of file [abip.h](#).

### 3.7.2.29 verbose

`abip_int` ABIP\_SETTINGS::verbose

Definition at line 120 of file [abip.h](#).

The documentation for this struct was generated from the following file:

- [include/abip.h](#)

## 3.8 ABIP\_SOL\_VARS Struct Reference

```
#include <abip.h>
```

### Public Attributes

- `abip_float` \* x
- `abip_float` \* y
- `abip_float` \* s

### 3.8.1 Detailed Description

Definition at line 132 of file [abip.h](#).

## 3.8.2 Member Data Documentation

### 3.8.2.1 s

`abip_float* ABIP_SOL_VARS::s`

Definition at line 136 of file [abip.h](#).

### 3.8.2.2 x

`abip_float* ABIP_SOL_VARS::x`

Definition at line 134 of file [abip.h](#).

### 3.8.2.3 y

`abip_float* ABIP_SOL_VARS::y`

Definition at line 135 of file [abip.h](#).

The documentation for this struct was generated from the following file:

- [include/abip.h](#)

## 3.9 ABIP\_WORK Struct Reference

```
#include <abip.h>
```

### Public Attributes

- [abip\\_float sigma](#)
- [abip\\_float gamma](#)
- [abip\\_float mu](#)
- [abip\\_float beta](#)
- [abip\\_float \\* u](#)
- [abip\\_float \\* v](#)
- [abip\\_float \\* v\\_origin](#)
- [abip\\_float \\* u\\_t](#)
- [abip\\_float \\* rel\\_ut](#)
- [abip\\_float nm\\_inf\\_b](#)
- [abip\\_float nm\\_inf\\_c](#)
- [abip\\_int m](#)
- [abip\\_int n](#)
- [ABIPMatrix \\* A](#)
- [abip\\_float \\* r](#)
- [abip\\_float a](#)

### 3.9.1 Detailed Description

Definition at line 161 of file [abip.h](#).

### 3.9.2 Member Data Documentation

#### 3.9.2.1 A

```
ABIPMatrix* ABIP_WORK::A
```

Definition at line 176 of file [abip.h](#).

#### 3.9.2.2 a

```
abip_float ABIP_WORK::a
```

Definition at line 178 of file [abip.h](#).

#### 3.9.2.3 beta

```
abip_float ABIP_WORK::beta
```

Definition at line 166 of file [abip.h](#).

#### 3.9.2.4 gamma

```
abip_float ABIP_WORK::gamma
```

Definition at line 164 of file [abip.h](#).

#### 3.9.2.5 m

```
abip_int ABIP_WORK::m
```

Definition at line 174 of file [abip.h](#).

### 3.9.2.6 mu

`abip_float ABIP_WORK::mu`

Definition at line 165 of file [abip.h](#).

### 3.9.2.7 n

`abip_int ABIP_WORK::n`

Definition at line 175 of file [abip.h](#).

### 3.9.2.8 nm\_inf\_b

`abip_float ABIP_WORK::nm_inf_b`

Definition at line 172 of file [abip.h](#).

### 3.9.2.9 nm\_inf\_c

`abip_float ABIP_WORK::nm_inf_c`

Definition at line 173 of file [abip.h](#).

### 3.9.2.10 r

`abip_float* ABIP_WORK::r`

Definition at line 177 of file [abip.h](#).

### 3.9.2.11 rel\_ut

`abip_float* ABIP_WORK::rel_ut`

Definition at line 171 of file [abip.h](#).

### 3.9.2.12 sigma

`abip_float` ABIP\_WORK::sigma

Definition at line 163 of file [abip.h](#).

### 3.9.2.13 u

`abip_float*` ABIP\_WORK::u

Definition at line 167 of file [abip.h](#).

### 3.9.2.14 u\_t

`abip_float*` ABIP\_WORK::u\_t

Definition at line 170 of file [abip.h](#).

### 3.9.2.15 v

`abip_float*` ABIP\_WORK::v

Definition at line 168 of file [abip.h](#).

### 3.9.2.16 v\_origin

`abip_float*` ABIP\_WORK::v\_origin

Definition at line 169 of file [abip.h](#).

The documentation for this struct was generated from the following file:

- [include/abip.h](#)

## 3.10 cs\_dmperm\_results Struct Reference

```
#include <cs.h>
```

## Public Attributes

- `csi * p`
- `csi * q`
- `csi * r`
- `csi * s`
- `csi nb`
- `csi rr [5]`
- `csi cc [5]`

### 3.10.1 Detailed Description

Definition at line 82 of file [cs.h](#).

### 3.10.2 Member Data Documentation

#### 3.10.2.1 cc

```
csi cs_dmperm_results::cc[5]
```

Definition at line 90 of file [cs.h](#).

#### 3.10.2.2 nb

```
csi cs_dmperm_results::nb
```

Definition at line 88 of file [cs.h](#).

#### 3.10.2.3 p

```
csi* cs_dmperm_results::p
```

Definition at line 84 of file [cs.h](#).

#### 3.10.2.4 q

```
csi* cs_dmperm_results::q
```

Definition at line 85 of file [cs.h](#).

### 3.10.2.5 r

```
csi* cs_dmperm_results::r
```

Definition at line 86 of file [cs.h](#).

### 3.10.2.6 rr

```
csi cs_dmperm_results::rr[5]
```

Definition at line 89 of file [cs.h](#).

### 3.10.2.7 s

```
csi* cs_dmperm_results::s
```

Definition at line 87 of file [cs.h](#).

The documentation for this struct was generated from the following file:

- [csparse/Include/cs.h](#)

## 3.11 cs\_numeric Struct Reference

```
#include <cs.h>
```

### Public Attributes

- [cs \\* L](#)
- [cs \\* U](#)
- [csi \\* pinv](#)
- [double \\* B](#)

### 3.11.1 Detailed Description

Definition at line 74 of file [cs.h](#).

### 3.11.2 Member Data Documentation

### 3.11.2.1 B

```
double* cs_numeric::B
```

Definition at line 79 of file [cs.h](#).

### 3.11.2.2 L

```
cs* cs_numeric::L
```

Definition at line 76 of file [cs.h](#).

### 3.11.2.3 pinv

```
csi* cs_numeric::pinv
```

Definition at line 78 of file [cs.h](#).

### 3.11.2.4 U

```
cs* cs_numeric::U
```

Definition at line 77 of file [cs.h](#).

The documentation for this struct was generated from the following file:

- [csparse/Include/cs.h](#)

## 3.12 cs\_sparse Struct Reference

```
#include <cs.h>
```

### Public Attributes

- [csi nzmax](#)
- [csi m](#)
- [csi n](#)
- [csi \\* p](#)
- [csi \\* i](#)
- [double \\* x](#)
- [csi nz](#)



### 3.12.1 Detailed Description

Definition at line 28 of file [cs.h](#).

### 3.12.2 Member Data Documentation

#### 3.12.2.1 i

```
csi* cs_sparse::i
```

Definition at line 34 of file [cs.h](#).

#### 3.12.2.2 m

```
csi cs_sparse::m
```

Definition at line 31 of file [cs.h](#).

#### 3.12.2.3 n

```
csi cs_sparse::n
```

Definition at line 32 of file [cs.h](#).

#### 3.12.2.4 nz

```
csi cs_sparse::nz
```

Definition at line 36 of file [cs.h](#).

#### 3.12.2.5 nzmax

```
csi cs_sparse::nzmax
```

Definition at line 30 of file [cs.h](#).

### 3.12.2.6 p

```
csi* cs_sparse::p
```

Definition at line 33 of file [cs.h](#).

### 3.12.2.7 x

```
double* cs_sparse::x
```

Definition at line 35 of file [cs.h](#).

The documentation for this struct was generated from the following file:

- [csparse/Include/cs.h](#)

## 3.13 cs\_symbolic Struct Reference

```
#include <cs.h>
```

### Public Attributes

- [csi](#) \* [pinv](#)
- [csi](#) \* [q](#)
- [csi](#) \* [parent](#)
- [csi](#) \* [cp](#)
- [csi](#) \* [leftmost](#)
- [csi](#) [m2](#)
- double [lnz](#)
- double [unz](#)

### 3.13.1 Detailed Description

Definition at line 62 of file [cs.h](#).

### 3.13.2 Member Data Documentation

#### 3.13.2.1 cp

```
csi* cs_symbolic::cp
```

Definition at line 67 of file [cs.h](#).

### 3.13.2.2 leftmost

```
csi* cs_symbolic::leftmost
```

Definition at line 68 of file [cs.h](#).

### 3.13.2.3 lnz

```
double cs_symbolic::lnz
```

Definition at line 70 of file [cs.h](#).

### 3.13.2.4 m2

```
csi cs_symbolic::m2
```

Definition at line 69 of file [cs.h](#).

### 3.13.2.5 parent

```
csi* cs_symbolic::parent
```

Definition at line 66 of file [cs.h](#).

### 3.13.2.6 pinv

```
csi* cs_symbolic::pinv
```

Definition at line 64 of file [cs.h](#).

### 3.13.2.7 q

```
csi* cs_symbolic::q
```

Definition at line 65 of file [cs.h](#).

### 3.13.2.8 unz

```
double cs_symbolic::unz
```

Definition at line 71 of file [cs.h](#).

The documentation for this struct was generated from the following file:

- [csparse/Include/cs.h](#)

## 3.14 Lasso Struct Reference

```
#include <lasso_config.h>
```

### Public Attributes

- enum [problem\\_type](#) [pro\\_type](#)
- [abip\\_int](#) [m](#)
- [abip\\_int](#) [n](#)
- [abip\\_int](#) [p](#)
- [abip\\_int](#) [q](#)
- [ABIPLinSysWork](#) \* [L](#)
- [ABIPSettings](#) \* [stgs](#)
- [ABIPData](#) \* [data](#)
- [abip\\_float](#) [sparsity](#)
- [abip\\_float](#) \* [rho\\_dr](#)
- [ABIPMatrix](#) \* [A](#)
- [ABIPMatrix](#) \* [Q](#)
- [abip\\_float](#) \* [b](#)
- [abip\\_float](#) \* [c](#)
- void(\* [scaling\\_data](#) )(lasso \*self, [ABIPCone](#) \*k)
- void(\* [un\\_scaling\\_sol](#) )(lasso \*self, [ABIPSolution](#) \*sol)
- void(\* [calc\\_residuals](#) )(lasso \*self, [ABIPWork](#) \*w, [ABIPResiduals](#) \*r, [abip\\_int](#) ipm\_iter, [abip\\_int](#) admm\_iter)
- [abip\\_int](#)(\* [init\\_spe\\_linsys\\_work](#) )(lasso \*self)
- [abip\\_int](#)(\* [solve\\_spe\\_linsys](#) )(lasso \*self, [abip\\_float](#) \*b, [abip\\_float](#) \*pcg\_warm\_start, [abip\\_int](#) iter, [abip\\_float](#) error\_ratio)
- void(\* [free\\_spe\\_linsys\\_work](#) )(lasso \*self)
- void(\* [spe\\_A\\_times](#) )(lasso \*self, const [abip\\_float](#) \*x, [abip\\_float](#) \*y)
- void(\* [spe\\_AT\\_times](#) )(lasso \*self, const [abip\\_float](#) \*x, [abip\\_float](#) \*y)
- [abip\\_float](#)(\* [inner\\_conv\\_check](#) )(lasso \*self, [ABIPWork](#) \*w)
- [abip\\_float](#) [lambda](#)
- [abip\\_float](#) \* [D\\_hat](#)
- [abip\\_float](#) \* [D](#)
- [abip\\_float](#) \* [E](#)
- [abip\\_float](#) [sc\\_b](#)
- [abip\\_float](#) [sc\\_c](#)
- [abip\\_float](#) [sc](#)
- [abip\\_float](#) [sc\\_cone1](#)
- [abip\\_float](#) [sc\\_cone2](#)

### 3.14.1 Detailed Description

Definition at line 14 of file [lasso\\_config.h](#).

### 3.14.2 Member Data Documentation

#### 3.14.2.1 A

```
ABIPMatrix* Lasso::A
```

Definition at line 30 of file [lasso\\_config.h](#).

#### 3.14.2.2 b

```
abip_float* Lasso::b
```

Definition at line 32 of file [lasso\\_config.h](#).

#### 3.14.2.3 c

```
abip_float* Lasso::c
```

Definition at line 33 of file [lasso\\_config.h](#).

#### 3.14.2.4 calc\_residuals

```
void(* Lasso::calc_residuals) (lasso *self, ABIPWork *w, ABIPResiduals *r, abip_int ipm_iter,  
abip_int admm_iter)
```

Definition at line 38 of file [lasso\\_config.h](#).

#### 3.14.2.5 D

```
abip_float* Lasso::D
```

Definition at line 50 of file [lasso\\_config.h](#).

### 3.14.2.6 D\_hat

`abip_float* Lasso::D_hat`

Definition at line 49 of file [lasso\\_config.h](#).

### 3.14.2.7 data

`ABIPData* Lasso::data`

Definition at line 24 of file [lasso\\_config.h](#).

### 3.14.2.8 E

`abip_float* Lasso::E`

Definition at line 51 of file [lasso\\_config.h](#).

### 3.14.2.9 free\_spe\_linsys\_work

`void(* Lasso::free_spe_linsys_work) (lasso *self)`

Definition at line 41 of file [lasso\\_config.h](#).

### 3.14.2.10 init\_spe\_linsys\_work

`abip_int(* Lasso::init_spe_linsys_work) (lasso *self)`

Definition at line 39 of file [lasso\\_config.h](#).

### 3.14.2.11 inner\_conv\_check

`abip_float(* Lasso::inner_conv_check) (lasso *self, ABIPWork *w)`

Definition at line 44 of file [lasso\\_config.h](#).

### 3.14.2.12 L

`ABIPLinSysWork*` Lasso::L

Definition at line 22 of file [lasso\\_config.h](#).

### 3.14.2.13 lambda

`abip_float` Lasso::lambda

Definition at line 47 of file [lasso\\_config.h](#).

### 3.14.2.14 m

`abip_int` Lasso::m

Definition at line 18 of file [lasso\\_config.h](#).

### 3.14.2.15 n

`abip_int` Lasso::n

Definition at line 19 of file [lasso\\_config.h](#).

### 3.14.2.16 p

`abip_int` Lasso::p

Definition at line 20 of file [lasso\\_config.h](#).

### 3.14.2.17 pro\_type

enum `problem_type` Lasso::pro\_type

Definition at line 17 of file [lasso\\_config.h](#).

**3.14.2.18 q**

`abip_int` Lasso::q

Definition at line 21 of file [lasso\\_config.h](#).

**3.14.2.19 Q**

`ABIPMatrix*` Lasso::Q

Definition at line 31 of file [lasso\\_config.h](#).

**3.14.2.20 rho\_dr**

`abip_float*` Lasso::rho\_dr

Definition at line 27 of file [lasso\\_config.h](#).

**3.14.2.21 sc**

`abip_float` Lasso::sc

Definition at line 54 of file [lasso\\_config.h](#).

**3.14.2.22 sc\_b**

`abip_float` Lasso::sc\_b

Definition at line 52 of file [lasso\\_config.h](#).

**3.14.2.23 sc\_c**

`abip_float` Lasso::sc\_c

Definition at line 53 of file [lasso\\_config.h](#).



#### 3.14.2.24 `sc_cone1`

`abip_float` Lasso::sc\_cone1

Definition at line 55 of file `lasso_config.h`.

#### 3.14.2.25 `sc_cone2`

`abip_float` Lasso::sc\_cone2

Definition at line 56 of file `lasso_config.h`.

#### 3.14.2.26 `scaling_data`

```
void(* Lasso::scaling_data) (lasso *self, ABIPConc *k)
```

Definition at line 36 of file `lasso_config.h`.

#### 3.14.2.27 `solve_spe_linsys`

```
abip_int(* Lasso::solve_spe_linsys) (lasso *self, abip_float *b, abip_float *pcg_warm_start,  
abip_int iter, abip_float error_ratio)
```

Definition at line 40 of file `lasso_config.h`.

#### 3.14.2.28 `sparsity`

`abip_float` Lasso::sparsity

Definition at line 25 of file `lasso_config.h`.

#### 3.14.2.29 `spe_A_times`

```
void(* Lasso::spe_A_times) (lasso *self, const abip_float *x, abip_float *y)
```

Definition at line 42 of file `lasso_config.h`.

### 3.14.2.30 spe\_AT\_times

```
void(* Lasso::spe_AT_times) (lasso *self, const abip_float *x, abip_float *y)
```

Definition at line 43 of file [lasso\\_config.h](#).

### 3.14.2.31 stgs

```
ABIPSettings* Lasso::stgs
```

Definition at line 23 of file [lasso\\_config.h](#).

### 3.14.2.32 un\_scaling\_sol

```
void(* Lasso::un_scaling_sol) (lasso *self, ABIPSolution *sol)
```

Definition at line 37 of file [lasso\\_config.h](#).

The documentation for this struct was generated from the following file:

- [include/lasso\\_config.h](#)

## 3.15 qcp Struct Reference

```
#include <qcp_config.h>
```

### Public Attributes

- enum [problem\\_type](#) pro\_type
- [abip\\_int](#) m
- [abip\\_int](#) n
- [abip\\_int](#) p
- [abip\\_int](#) q
- [ABIPLinSysWork](#) \* L
- [ABIPSettings](#) \* stgs
- [ABIPData](#) \* data
- [abip\\_float](#) sparsity
- [abip\\_float](#) \* rho\_dr
- [ABIPMatrix](#) \* A
- [ABIPMatrix](#) \* Q
- [abip\\_float](#) \* b
- [abip\\_float](#) \* c
- void(\* [scaling\\_data](#) )(qcp \*self, [ABIPConc](#) \*k)
- void(\* [un\\_scaling\\_sol](#) )(qcp \*self, [ABIPSolution](#) \*sol)
- void(\* [calc\\_residuals](#) )(qcp \*self, [ABIPWork](#) \*w, [ABIPResiduals](#) \*r, [abip\\_int](#) ipm\_iter, [abip\\_int](#) admm\_iter)
- [abip\\_int](#)(\* [init\\_spe\\_linsys\\_work](#) )(qcp \*self)
- [abip\\_int](#)(\* [solve\\_spe\\_linsys](#) )(qcp \*self, [abip\\_float](#) \*b, [abip\\_float](#) \*pcg\_warm\_start, [abip\\_int](#) iter, [abip\\_float](#) error\_ratio)
- void(\* [free\\_spe\\_linsys\\_work](#) )(qcp \*self)
- void(\* [spe\\_A\\_times](#) )(qcp \*self, const [abip\\_float](#) \*x, [abip\\_float](#) \*y)
- void(\* [spe\\_AT\\_times](#) )(qcp \*self, const [abip\\_float](#) \*x, [abip\\_float](#) \*y)
- [abip\\_float](#)(\* [inner\\_conv\\_check](#) )(qcp \*self, [ABIPWork](#) \*w)
- [abip\\_float](#) \* D
- [abip\\_float](#) \* E
- [abip\\_float](#) sc\_b
- [abip\\_float](#) sc\_c

### 3.15.1 Detailed Description

Definition at line 14 of file [qcp\\_config.h](#).

### 3.15.2 Member Data Documentation

#### 3.15.2.1 A

[ABIPMatrix\\*](#) [qcp::A](#)

Definition at line 30 of file [qcp\\_config.h](#).

#### 3.15.2.2 b

[abip\\_float\\*](#) [qcp::b](#)

Definition at line 32 of file [qcp\\_config.h](#).

#### 3.15.2.3 c

[abip\\_float\\*](#) [qcp::c](#)

Definition at line 33 of file [qcp\\_config.h](#).

#### 3.15.2.4 calc\_residuals

```
void(* qcp::calc\_residuals) (qcp *self, ABIPWork *w, ABIPResiduals *r, abip\_int ipm_iter, abip\_int
admm_iter)
```

Definition at line 38 of file [qcp\\_config.h](#).

#### 3.15.2.5 D

[abip\\_float\\*](#) [qcp::D](#)

Definition at line 48 of file [qcp\\_config.h](#).

### 3.15.2.6 data

`ABIPData*` `qcp::data`

Definition at line 24 of file `qcp_config.h`.

### 3.15.2.7 E

`abip_float*` `qcp::E`

Definition at line 49 of file `qcp_config.h`.

### 3.15.2.8 free\_spe\_linsys\_work

`void` (\* `qcp::free_spe_linsys_work`) (`qcp *self`)

Definition at line 41 of file `qcp_config.h`.

### 3.15.2.9 init\_spe\_linsys\_work

`abip_int` (\* `qcp::init_spe_linsys_work`) (`qcp *self`)

Definition at line 39 of file `qcp_config.h`.

### 3.15.2.10 inner\_conv\_check

`abip_float` (\* `qcp::inner_conv_check`) (`qcp *self`, `ABIPWork *w`)

Definition at line 44 of file `qcp_config.h`.

### 3.15.2.11 L

`ABIPLinSysWork*` `qcp::L`

Definition at line 22 of file `qcp_config.h`.

### 3.15.2.12 m

`abip_int` `qcp::m`

Definition at line 18 of file `qcp_config.h`.

### 3.15.2.13 n

`abip_int` `qcp::n`

Definition at line 19 of file `qcp_config.h`.

### 3.15.2.14 p

`abip_int` `qcp::p`

Definition at line 20 of file `qcp_config.h`.

### 3.15.2.15 pro\_type

`enum` `problem_type` `qcp::pro_type`

Definition at line 17 of file `qcp_config.h`.

### 3.15.2.16 q

`abip_int` `qcp::q`

Definition at line 21 of file `qcp_config.h`.

### 3.15.2.17 Q

`ABIPMatrix*` `qcp::Q`

Definition at line 31 of file `qcp_config.h`.

**3.15.2.18 rho\_dr**

```
abip_float* qcp::rho_dr
```

Definition at line 27 of file [qcp\\_config.h](#).

**3.15.2.19 sc\_b**

```
abip_float qcp::sc_b
```

Definition at line 50 of file [qcp\\_config.h](#).

**3.15.2.20 sc\_c**

```
abip_float qcp::sc_c
```

Definition at line 51 of file [qcp\\_config.h](#).

**3.15.2.21 scaling\_data**

```
void(* qcp::scaling_data) (qcp *self, ABIPCone *k)
```

Definition at line 36 of file [qcp\\_config.h](#).

**3.15.2.22 solve\_spe\_linsys**

```
abip_int(* qcp::solve_spe_linsys) (qcp *self, abip_float *b, abip_float *pcg_warm_start, abip_int  
iter, abip_float error_ratio)
```

Definition at line 40 of file [qcp\\_config.h](#).

**3.15.2.23 sparsity**

```
abip_float qcp::sparsity
```

Definition at line 25 of file [qcp\\_config.h](#).

### 3.15.2.24 spe\_A\_times

```
void(* qcp::spe_A_times) (qcp *self, const abip_float *x, abip_float *y)
```

Definition at line 42 of file [qcp\\_config.h](#).

### 3.15.2.25 spe\_AT\_times

```
void(* qcp::spe_AT_times) (qcp *self, const abip_float *x, abip_float *y)
```

Definition at line 43 of file [qcp\\_config.h](#).

### 3.15.2.26 stgs

```
ABIPSettings* qcp::stgs
```

Definition at line 23 of file [qcp\\_config.h](#).

### 3.15.2.27 un\_scaling\_sol

```
void(* qcp::un_scaling_sol) (qcp *self, ABIPSolution *sol)
```

Definition at line 37 of file [qcp\\_config.h](#).

The documentation for this struct was generated from the following file:

- [include/qcp\\_config.h](#)

## 3.16 solve\_specific\_problem Struct Reference

```
#include <abip.h>
```

## Public Attributes

- enum `problem_type` `pro_type`
- `abip_int` `m`
- `abip_int` `n`
- `abip_int` `p`
- `abip_int` `q`
- `ABIPLinSysWork` \* `L`
- `ABIPSettings` \* `stgs`
- `ABIPData` \* `data`
- `abip_float` `sparsity`
- `abip_float` \* `rho_dr`
- `ABIPMatrix` \* `A`
- `ABIPMatrix` \* `Q`
- `abip_float` \* `b`
- `abip_float` \* `c`
- `void(* scaling_data )(spe_problem *self, ABIPConc *k)`
- `void(* un_scaling_sol )(spe_problem *self, ABIPSolution *sol)`
- `void(* calc_residuals )(spe_problem *self, ABIPWork *w, ABIPResiduals *r, abip_int ipm_iter, abip_int admm_iter)`
- `abip_int(* init_spe_linsys_work )(spe_problem *self)`
- `abip_int(* solve_spe_linsys )(spe_problem *self, abip_float *b, abip_float *pcg_warm_start, abip_int iter, abip_float error_ratio)`
- `void(* free_spe_linsys_work )(spe_problem *self)`
- `void(* spe_A_times )(spe_problem *self, const abip_float *x, abip_float *y)`
- `void(* spe_AT_times )(spe_problem *self, const abip_float *x, abip_float *y)`
- `abip_float(* inner_conv_check )(spe_problem *self, ABIPWork *w)`

### 3.16.1 Detailed Description

Definition at line 29 of file `abip.h`.

### 3.16.2 Member Data Documentation

#### 3.16.2.1 A

`ABIPMatrix*` `solve_specific_problem::A`

Definition at line 44 of file `abip.h`.

#### 3.16.2.2 b

`abip_float*` `solve_specific_problem::b`

Definition at line 46 of file `abip.h`.



### 3.16.2.3 c

`abip_float*` solve\_specific\_problem::c

Definition at line 47 of file [abip.h](#).

### 3.16.2.4 calc\_residuals

```
void(* solve_specific_problem::calc_residuals) (spe_problem *self, ABIPWork *w, ABIPResiduals *r, abip_int ipm_iter, abip_int admm_iter)
```

Definition at line 52 of file [abip.h](#).

### 3.16.2.5 data

`ABIPData*` solve\_specific\_problem::data

Definition at line 38 of file [abip.h](#).

### 3.16.2.6 free\_spe\_linsys\_work

```
void(* solve_specific_problem::free_spe_linsys_work) (spe_problem *self)
```

Definition at line 55 of file [abip.h](#).

### 3.16.2.7 init\_spe\_linsys\_work

```
abip_int(* solve_specific_problem::init_spe_linsys_work) (spe_problem *self)
```

Definition at line 53 of file [abip.h](#).

### 3.16.2.8 inner\_conv\_check

```
abip_float(* solve_specific_problem::inner_conv_check) (spe_problem *self, ABIPWork *w)
```

Definition at line 58 of file [abip.h](#).

### 3.16.2.9 L

`ABIPLinSysWork* solve_specific_problem::L`

Definition at line 36 of file [abip.h](#).

### 3.16.2.10 m

`abip_int solve_specific_problem::m`

Definition at line 32 of file [abip.h](#).

### 3.16.2.11 n

`abip_int solve_specific_problem::n`

Definition at line 33 of file [abip.h](#).

### 3.16.2.12 p

`abip_int solve_specific_problem::p`

Definition at line 34 of file [abip.h](#).

### 3.16.2.13 pro\_type

`enum problem_type solve_specific_problem::pro_type`

Definition at line 31 of file [abip.h](#).

### 3.16.2.14 q

`abip_int solve_specific_problem::q`

Definition at line 35 of file [abip.h](#).

### 3.16.2.15 Q

`ABIPMatrix*` solve\_specific\_problem::Q

Definition at line 45 of file [abip.h](#).

### 3.16.2.16 rho\_dr

`abip_float*` solve\_specific\_problem::rho\_dr

Definition at line 41 of file [abip.h](#).

### 3.16.2.17 scaling\_data

`void(* solve_specific_problem::scaling_data) (spe_problem *self, ABIPConc *k)`

Definition at line 50 of file [abip.h](#).

### 3.16.2.18 solve\_spe\_linsys

`abip_int(* solve_specific_problem::solve_spe_linsys) (spe_problem *self, abip_float *b, abip_float *pcg_warm_start, abip_int iter, abip_float error_ratio)`

Definition at line 54 of file [abip.h](#).

### 3.16.2.19 sparsity

`abip_float` solve\_specific\_problem::sparsity

Definition at line 39 of file [abip.h](#).

### 3.16.2.20 spe\_A\_times

`void(* solve_specific_problem::spe_A_times) (spe_problem *self, const abip_float *x, abip_float *y)`

Definition at line 56 of file [abip.h](#).

### 3.16.2.21 spe\_AT\_times

```
void(* solve_specific_problem::spe_AT_times) (spe_problem *self, const abip_float *x, abip_float *y)
```

Definition at line 57 of file [abip.h](#).

### 3.16.2.22 stgs

```
ABIPSettings* solve_specific_problem::stgs
```

Definition at line 37 of file [abip.h](#).

### 3.16.2.23 un\_scaling\_sol

```
void(* solve_specific_problem::un_scaling_sol) (spe_problem *self, ABIPSolution *sol)
```

Definition at line 51 of file [abip.h](#).

The documentation for this struct was generated from the following file:

- [include/abip.h](#)

## 3.17 SuiteSparse\_config\_struct Struct Reference

```
#include <SuiteSparse_config.h>
```

### Public Attributes

- void>(\* [malloc\\_func](#))(size\_t)
- void>(\* [calloc\\_func](#))(size\_t, size\_t)
- void>(\* [realloc\\_func](#))(void \*, size\_t)
- void(\* [free\\_func](#))(void \*)
- int(\* [printf\\_func](#))(const char \*,...)
- [abip\\_float](#)(\* [hypot\\_func](#))([abip\\_float](#), [abip\\_float](#))
- int(\* [divcomplex\\_func](#))([abip\\_float](#), [abip\\_float](#), [abip\\_float](#), [abip\\_float](#), [abip\\_float](#) \*, [abip\\_float](#) \*)

### 3.17.1 Detailed Description

Definition at line 87 of file [SuiteSparse\\_config.h](#).

## 3.17.2 Member Data Documentation

### 3.17.2.1 calloc\_func

```
void *(* SuiteSparse_config_struct::calloc_func) (size_t, size_t)
```

Definition at line 90 of file [SuiteSparse\\_config.h](#).

### 3.17.2.2 divcomplex\_func

```
int (* SuiteSparse_config_struct::divcomplex_func) (abip_float, abip_float, abip_float, abip_float,  
abip_float *, abip_float *)
```

Definition at line 95 of file [SuiteSparse\\_config.h](#).

### 3.17.2.3 free\_func

```
void (* SuiteSparse_config_struct::free_func) (void *)
```

Definition at line 92 of file [SuiteSparse\\_config.h](#).

### 3.17.2.4 hypot\_func

```
abip_float (* SuiteSparse_config_struct::hypot_func) (abip_float, abip_float)
```

Definition at line 94 of file [SuiteSparse\\_config.h](#).

### 3.17.2.5 malloc\_func

```
void *(* SuiteSparse_config_struct::malloc_func) (size_t)
```

Definition at line 89 of file [SuiteSparse\\_config.h](#).

### 3.17.2.6 printf\_func

```
int (* SuiteSparse_config_struct::printf_func) (const char *, ...)
```

Definition at line 93 of file [SuiteSparse\\_config.h](#).

### 3.17.2.7 realloc\_func

```
void *(* SuiteSparse_config_struct::realloc_func) (void *, size_t)
```

Definition at line 91 of file [SuiteSparse\\_config.h](#).

The documentation for this struct was generated from the following file:

- [amd/SuiteSparse\\_config.h](#)

## 3.18 Svm Struct Reference

```
#include <svm_config.h>
```

### Public Attributes

- enum [problem\\_type](#) [pro\\_type](#)
- [abip\\_int](#) [m](#)
- [abip\\_int](#) [n](#)
- [abip\\_int](#) [p](#)
- [abip\\_int](#) [q](#)
- [ABIPLinSysWork](#) \* [L](#)
- [ABIPSettings](#) \* [stgs](#)
- [ABIPData](#) \* [data](#)
- [abip\\_float](#) [sparsity](#)
- [abip\\_float](#) \* [rho\\_dr](#)
- [ABIPMatrix](#) \* [A](#)
- [ABIPMatrix](#) \* [Q](#)
- [abip\\_float](#) \* [b](#)
- [abip\\_float](#) \* [c](#)
- void(\* [scaling\\_data](#) )(svm \*self, [ABIPConc](#) \*k)
- void(\* [un\\_scaling\\_sol](#) )(svm \*self, [ABIPSolution](#) \*sol)
- void(\* [calc\\_residuals](#) )(svm \*self, [ABIPWork](#) \*w, [ABIPResiduals](#) \*r, [abip\\_int](#) ipm\_iter, [abip\\_int](#) admm\_iter)
- [abip\\_int](#)(\* [init\\_spe\\_linsys\\_work](#) )(svm \*self)
- [abip\\_int](#)(\* [solve\\_spe\\_linsys](#) )(svm \*self, [abip\\_float](#) \*b, [abip\\_float](#) \*pcg\_warm\_start, [abip\\_int](#) iter, [abip\\_float](#) error\_ratio)
- void(\* [free\\_spe\\_linsys\\_work](#) )(svm \*self)
- void(\* [spe\\_A\\_times](#) )(svm \*self, const [abip\\_float](#) \*x, [abip\\_float](#) \*y)
- void(\* [spe\\_AT\\_times](#) )(svm \*self, const [abip\\_float](#) \*x, [abip\\_float](#) \*y)
- [abip\\_float](#)(\* [inner\\_conv\\_check](#) )(svm \*self, [ABIPWork](#) \*w)
- [abip\\_float](#) [lambda](#)
- [abip\\_float](#) \* [sc\\_D](#)

- [abip\\_float](#) \* [sc\\_E](#)
- [abip\\_float](#) \* [sc\\_F](#)
- [abip\\_float](#) [sc\\_b](#)
- [abip\\_float](#) [sc\\_c](#)
- [abip\\_float](#) [sc](#)
- [abip\\_float](#) [sc\\_cone1](#)
- [abip\\_float](#) [sc\\_cone2](#)
- [ABIPMatrix](#) \* [wA](#)
- [abip\\_float](#) \* [wy](#)
- [abip\\_float](#) \* [wB](#)
- [abip\\_float](#) \* [wC](#)
- [abip\\_float](#) \* [wD](#)
- [abip\\_float](#) \* [wE](#)
- [abip\\_float](#) \* [wF](#)
- [abip\\_float](#) \* [wG](#)
- [abip\\_float](#) \* [wH](#)
- [ABIPMatrix](#) \* [wX](#)

### 3.18.1 Detailed Description

Definition at line 14 of file [svm\\_config.h](#).

### 3.18.2 Member Data Documentation

#### 3.18.2.1 A

[ABIPMatrix](#)\* [Svm::A](#)

Definition at line 30 of file [svm\\_config.h](#).

#### 3.18.2.2 b

[abip\\_float](#)\* [Svm::b](#)

Definition at line 32 of file [svm\\_config.h](#).

#### 3.18.2.3 c

[abip\\_float](#)\* [Svm::c](#)

Definition at line 33 of file [svm\\_config.h](#).

#### 3.18.2.4 calc\_residuals

```
void(* Svm::calc_residuals) (svm *self, ABIPWork *w, ABIPResiduals *r, abip_int ipm_iter, abip_int  
admm_iter)
```

Definition at line 38 of file [svm\\_config.h](#).

#### 3.18.2.5 data

```
ABIPData* Svm::data
```

Definition at line 24 of file [svm\\_config.h](#).

#### 3.18.2.6 free\_spe\_linsys\_work

```
void(* Svm::free_spe_linsys_work) (svm *self)
```

Definition at line 41 of file [svm\\_config.h](#).

#### 3.18.2.7 init\_spe\_linsys\_work

```
abip_int(* Svm::init_spe_linsys_work) (svm *self)
```

Definition at line 39 of file [svm\\_config.h](#).

#### 3.18.2.8 inner\_conv\_check

```
abip_float(* Svm::inner_conv_check) (svm *self, ABIPWork *w)
```

Definition at line 44 of file [svm\\_config.h](#).

#### 3.18.2.9 L

```
ABIPLinSysWork* Svm::L
```

Definition at line 22 of file [svm\\_config.h](#).



### 3.18.2.10 `lambda`

`abip_float` `Svm::lambda`

Definition at line 47 of file `svm_config.h`.

### 3.18.2.11 `m`

`abip_int` `Svm::m`

Definition at line 18 of file `svm_config.h`.

### 3.18.2.12 `n`

`abip_int` `Svm::n`

Definition at line 19 of file `svm_config.h`.

### 3.18.2.13 `p`

`abip_int` `Svm::p`

Definition at line 20 of file `svm_config.h`.

### 3.18.2.14 `pro_type`

enum `problem_type` `Svm::pro_type`

Definition at line 17 of file `svm_config.h`.

### 3.18.2.15 `q`

`abip_int` `Svm::q`

Definition at line 21 of file `svm_config.h`.

### 3.18.2.16 Q

`ABIPMatrix* Svm::Q`

Definition at line 31 of file [svm\\_config.h](#).

### 3.18.2.17 rho\_dr

`abip_float* Svm::rho_dr`

Definition at line 27 of file [svm\\_config.h](#).

### 3.18.2.18 sc

`abip_float Svm::sc`

Definition at line 54 of file [svm\\_config.h](#).

### 3.18.2.19 sc\_b

`abip_float Svm::sc_b`

Definition at line 52 of file [svm\\_config.h](#).

### 3.18.2.20 sc\_c

`abip_float Svm::sc_c`

Definition at line 53 of file [svm\\_config.h](#).

### 3.18.2.21 sc\_cone1

`abip_float Svm::sc_cone1`

Definition at line 55 of file [svm\\_config.h](#).

### 3.18.2.22 `sc_cone2`

`abip_float` `Svm::sc_cone2`

Definition at line 56 of file `svm_config.h`.

### 3.18.2.23 `sc_D`

`abip_float*` `Svm::sc_D`

Definition at line 49 of file `svm_config.h`.

### 3.18.2.24 `sc_E`

`abip_float*` `Svm::sc_E`

Definition at line 50 of file `svm_config.h`.

### 3.18.2.25 `sc_F`

`abip_float*` `Svm::sc_F`

Definition at line 51 of file `svm_config.h`.

### 3.18.2.26 `scaling_data`

`void(* Svm::scaling_data) (svm *self, ABIPConc *k)`

Definition at line 36 of file `svm_config.h`.

### 3.18.2.27 `solve_spe_linsys`

`abip_int(* Svm::solve_spe_linsys) (svm *self, abip_float *b, abip_float *pcg_warm_start, abip_int iter, abip_float error_ratio)`

Definition at line 40 of file `svm_config.h`.

### 3.18.2.28 sparsity

`abip_float` `Svm::sparsity`

Definition at line 25 of file `svm_config.h`.

### 3.18.2.29 spe\_A\_times

`void(* Svm::spe_A_times)` (`svm *self`, `const abip_float *x`, `abip_float *y`)

Definition at line 42 of file `svm_config.h`.

### 3.18.2.30 spe\_AT\_times

`void(* Svm::spe_AT_times)` (`svm *self`, `const abip_float *x`, `abip_float *y`)

Definition at line 43 of file `svm_config.h`.

### 3.18.2.31 stgs

`ABIPSettings*` `Svm::stgs`

Definition at line 23 of file `svm_config.h`.

### 3.18.2.32 un\_scaling\_sol

`void(* Svm::un_scaling_sol)` (`svm *self`, `ABIPSolution *sol`)

Definition at line 37 of file `svm_config.h`.

### 3.18.2.33 wA

`ABIPMatrix*` `Svm::wA`

Definition at line 58 of file `svm_config.h`.

### 3.18.2.34 wB

`abip_float*` Svm: :wB

Definition at line 60 of file [svm\\_config.h](#).

### 3.18.2.35 wC

`abip_float*` Svm: :wC

Definition at line 61 of file [svm\\_config.h](#).

### 3.18.2.36 wD

`abip_float*` Svm: :wD

Definition at line 62 of file [svm\\_config.h](#).

### 3.18.2.37 wE

`abip_float*` Svm: :wE

Definition at line 63 of file [svm\\_config.h](#).

### 3.18.2.38 wF

`abip_float*` Svm: :wF

Definition at line 64 of file [svm\\_config.h](#).

### 3.18.2.39 wG

`abip_float*` Svm: :wG

Definition at line 65 of file [svm\\_config.h](#).

### 3.18.2.40 wH

`abip_float*` `Svm::wH`

Definition at line 66 of file [svm\\_config.h](#).

### 3.18.2.41 wX

`ABIPMatrix*` `Svm::wX`

Definition at line 67 of file [svm\\_config.h](#).

### 3.18.2.42 wy

`abip_float*` `Svm::wy`

Definition at line 59 of file [svm\\_config.h](#).

The documentation for this struct was generated from the following file:

- [include/svm\\_config.h](#)

## 3.19 SVMqp Struct Reference

```
#include <svm_qp_config.h>
```

### Public Attributes

- enum [problem\\_type](#) `pro_type`
- [abip\\_int](#) `m`
- [abip\\_int](#) `n`
- [abip\\_int](#) `p`
- [abip\\_int](#) `q`
- [ABIPLinSysWork](#) \* `L`
- [ABIPSettings](#) \* `stgs`
- [ABIPData](#) \* `data`
- [abip\\_float](#) `sparsity`
- [abip\\_float](#) \* `rho_dr`
- [ABIPMatrix](#) \* `A`
- [ABIPMatrix](#) \* `Q`
- [abip\\_float](#) \* `b`
- [abip\\_float](#) \* `c`
- void(\* [scaling\\_data](#) )(svmqp \*self, [ABIPCone](#) \*k)
- void(\* [un\\_scaling\\_sol](#) )(svmqp \*self, [ABIPSolution](#) \*sol)
- void(\* [calc\\_residuals](#) )(svmqp \*self, [ABIPWork](#) \*w, [ABIPResiduals](#) \*r, [abip\\_int](#) ipm\_iter, [abip\\_int](#) admm\_iter)

- [abip\\_int](#)(\* [init\\_spe\\_linsys\\_work](#) )(svmqp \*self)
- [abip\\_int](#)(\* [solve\\_spe\\_linsys](#) )(svmqp \*self, [abip\\_float](#) \*b, [abip\\_float](#) \*pcg\_warm\_start, [abip\\_int](#) iter, [abip\\_float](#) error\_ratio)
- void(\* [free\\_spe\\_linsys\\_work](#) )(svmqp \*self)
- void(\* [spe\\_A\\_times](#) )(svmqp \*self, const [abip\\_float](#) \*x, [abip\\_float](#) \*y)
- void(\* [spe\\_AT\\_times](#) )(svmqp \*self, const [abip\\_float](#) \*x, [abip\\_float](#) \*y)
- [abip\\_float](#)(\* [inner\\_conv\\_check](#) )(svmqp \*self, [ABIPWork](#) \*w)
- [abip\\_float](#) lambda
- [abip\\_float](#) \* D
- [abip\\_float](#) \* E
- [abip\\_float](#) \* F
- [abip\\_float](#) \* H
- [abip\\_float](#) sc\_b
- [abip\\_float](#) sc\_c

### 3.19.1 Detailed Description

Definition at line 14 of file [svm\\_qp\\_config.h](#).

### 3.19.2 Member Data Documentation

#### 3.19.2.1 A

[ABIPMatrix](#)\* SVMqp::A

Definition at line 30 of file [svm\\_qp\\_config.h](#).

#### 3.19.2.2 b

[abip\\_float](#)\* SVMqp::b

Definition at line 32 of file [svm\\_qp\\_config.h](#).

#### 3.19.2.3 c

[abip\\_float](#)\* SVMqp::c

Definition at line 33 of file [svm\\_qp\\_config.h](#).

#### 3.19.2.4 calc\_residuals

```
void(* SVMqp::calc_residuals) (svmqp *self, ABIPWork *w, ABIPResiduals *r, abip_int ipm_iter,  
abip_int admm_iter)
```

Definition at line 38 of file [svm\\_qp\\_config.h](#).

#### 3.19.2.5 D

```
abip_float* SVMqp::D
```

Definition at line 49 of file [svm\\_qp\\_config.h](#).

#### 3.19.2.6 data

```
ABIPData* SVMqp::data
```

Definition at line 24 of file [svm\\_qp\\_config.h](#).

#### 3.19.2.7 E

```
abip_float* SVMqp::E
```

Definition at line 50 of file [svm\\_qp\\_config.h](#).

#### 3.19.2.8 F

```
abip_float* SVMqp::F
```

Definition at line 51 of file [svm\\_qp\\_config.h](#).

#### 3.19.2.9 free\_spe\_linsys\_work

```
void(* SVMqp::free_spe_linsys_work) (svmqp *self)
```

Definition at line 41 of file [svm\\_qp\\_config.h](#).



### 3.19.2.10 H

`abip_float*` SVMqp::H

Definition at line 52 of file [svm\\_qp\\_config.h](#).

### 3.19.2.11 init\_spe\_linsys\_work

`abip_int` (\* SVMqp::init\_spe\_linsys\_work) (svmqp \*self)

Definition at line 39 of file [svm\\_qp\\_config.h](#).

### 3.19.2.12 inner\_conv\_check

`abip_float` (\* SVMqp::inner\_conv\_check) (svmqp \*self, ABIPWork \*w)

Definition at line 44 of file [svm\\_qp\\_config.h](#).

### 3.19.2.13 L

`ABIPLinSysWork*` SVMqp::L

Definition at line 22 of file [svm\\_qp\\_config.h](#).

### 3.19.2.14 lambda

`abip_float` SVMqp::lambda

Definition at line 47 of file [svm\\_qp\\_config.h](#).

### 3.19.2.15 m

`abip_int` SVMqp::m

Definition at line 18 of file [svm\\_qp\\_config.h](#).

### 3.19.2.16 n

`abip_int SVMqp::n`

Definition at line 19 of file [svm\\_qp\\_config.h](#).

### 3.19.2.17 p

`abip_int SVMqp::p`

Definition at line 20 of file [svm\\_qp\\_config.h](#).

### 3.19.2.18 pro\_type

`enum problem_type SVMqp::pro_type`

Definition at line 17 of file [svm\\_qp\\_config.h](#).

### 3.19.2.19 q

`abip_int SVMqp::q`

Definition at line 21 of file [svm\\_qp\\_config.h](#).

### 3.19.2.20 Q

`ABIPMatrix* SVMqp::Q`

Definition at line 31 of file [svm\\_qp\\_config.h](#).

### 3.19.2.21 rho\_dr

`abip_float* SVMqp::rho_dr`

Definition at line 27 of file [svm\\_qp\\_config.h](#).

### 3.19.2.22 sc\_b

`abip_float SVMqp::sc_b`

Definition at line 53 of file [svm\\_qp\\_config.h](#).

### 3.19.2.23 sc\_c

`abip_float SVMqp::sc_c`

Definition at line 54 of file [svm\\_qp\\_config.h](#).

### 3.19.2.24 scaling\_data

`void(* SVMqp::scaling_data) (svmqp *self, ABIPConc *k)`

Definition at line 36 of file [svm\\_qp\\_config.h](#).

### 3.19.2.25 solve\_spe\_linsys

`abip_int(* SVMqp::solve_spe_linsys) (svmqp *self, abip_float *b, abip_float *pcg_warm_start,  
abip_int iter, abip_float error_ratio)`

Definition at line 40 of file [svm\\_qp\\_config.h](#).

### 3.19.2.26 sparsity

`abip_float SVMqp::sparsity`

Definition at line 25 of file [svm\\_qp\\_config.h](#).

### 3.19.2.27 spe\_A\_times

`void(* SVMqp::spe_A_times) (svmqp *self, const abip_float *x, abip_float *y)`

Definition at line 42 of file [svm\\_qp\\_config.h](#).

### 3.19.2.28 spe\_AT\_times

```
void(* SVMqp::spe_AT_times) (svmqp *self, const abip_float *x, abip_float *y)
```

Definition at line 43 of file [svm\\_qp\\_config.h](#).

### 3.19.2.29 stgs

```
ABIPSettings* SVMqp::stgs
```

Definition at line 23 of file [svm\\_qp\\_config.h](#).

### 3.19.2.30 un\_scaling\_sol

```
void(* SVMqp::un_scaling_sol) (svmqp *self, ABIPSolution *sol)
```

Definition at line 37 of file [svm\\_qp\\_config.h](#).

The documentation for this struct was generated from the following file:

- [include/svm\\_qp\\_config.h](#)

## Chapter 4

# File Documentation

### 4.1 amd/amd.h File Reference

```
#include <stddef.h>
#include "SuiteSparse_config.h"
```

#### Macros

- `#define EXTERN` extern
- `#define AMD_CONTROL` 5 /\* size of Control array \*/
- `#define AMD_INFO` 20 /\* size of Info array \*/
- `#define AMD_DENSE` 0 /\* "dense" if degree > Control [0] \* sqrt (n) \*/
- `#define AMD_AGGRESSIVE` 1 /\* do aggressive absorption if Control [1] != 0 \*/
- `#define AMD_DEFAULT_DENSE` 10.0 /\* default "dense" degree 10\*sqrt(n) \*/
- `#define AMD_DEFAULT_AGGRESSIVE` 1 /\* do aggressive absorption by default \*/
- `#define AMD_STATUS` 0 /\* return value of `amd_order` and `amd_l_order` \*/
- `#define AMD_N` 1 /\* A is n-by-n \*/
- `#define AMD_NZ` 2 /\* number of nonzeros in A \*/
- `#define AMD_SYMMETRY` 3 /\* symmetry of pattern (1 is sym., 0 is unsym.) \*/
- `#define AMD_NZDIAG` 4 /\* # of entries on diagonal \*/
- `#define AMD_NZ_A_PLUS_AT` 5 /\* nz in A+A' \*/
- `#define AMD_NDENSE` 6 /\* number of "dense" rows/columns in A \*/
- `#define AMD_MEMORY` 7 /\* amount of memory used by AMD \*/
- `#define AMD_NCMPA` 8 /\* number of garbage collections in AMD \*/
- `#define AMD_LNZ` 9 /\* approx. nz in L, excluding the diagonal \*/
- `#define AMD_NDIV` 10 /\* number of fl. point divides for LU and LDL' \*/
- `#define AMD_NMULTSUBS_LDL` 11 /\* number of fl. point (\*,-) pairs for LDL' \*/
- `#define AMD_NMULTSUBS_LU` 12 /\* number of fl. point (\*,-) pairs for LU \*/
- `#define AMD_DMAX` 13 /\* max nz. in any column of L, incl. diagonal \*/
- `#define AMD_OK` 0 /\* success \*/
- `#define AMD_OUT_OF_MEMORY` -1 /\* malloc failed, or problem too large \*/
- `#define AMD_INVALID` -2 /\* input arguments are not valid \*/
- `#define AMD_OK_BUT_JUMBLED`
- `#define AMD_DATE` "May 4, 2016"
- `#define AMD_VERSION_CODE(main, sub)` ((main) \* 1000 + (sub))
- `#define AMD_MAIN_VERSION` 2
- `#define AMD_SUB_VERSION` 4
- `#define AMD_SUBSUB_VERSION` 6
- `#define AMD_VERSION` AMD\_VERSION\_CODE(AMD\_MAIN\_VERSION,AMD\_SUB\_VERSION)

## Functions

- `int amd_order` (int n, const int Ap[], const int Ai[], int P[], `abip_float` Control[], `abip_float` ABIPInfo[])
- `SuiteSparse_long amd_l_order` (`SuiteSparse_long` n, const `SuiteSparse_long` Ap[], const `SuiteSparse_long` Ai[], `SuiteSparse_long` P[], `abip_float` Control[], `abip_float` ABIPInfo[])
- `void amd_2` (int n, int Pe[], int lw[], int Len[], int iwlen, int pfree, int Nv[], int Next[], int Last[], int Head[], int Elen[], int Degree[], int W[], `abip_float` Control[], `abip_float` ABIPInfo[])
- `void amd_l2` (`SuiteSparse_long` n, `SuiteSparse_long` Pe[], `SuiteSparse_long` lw[], `SuiteSparse_long` Len[], `SuiteSparse_long` iwlen, `SuiteSparse_long` pfree, `SuiteSparse_long` Nv[], `SuiteSparse_long` Next[], `SuiteSparse_long` Last[], `SuiteSparse_long` Head[], `SuiteSparse_long` Elen[], `SuiteSparse_long` Degree[], `SuiteSparse_long` W[], `abip_float` Control[], `abip_float` ABIPInfo[])
- `int amd_valid` (int n\_row, int n\_col, const int Ap[], const int Ai[])
- `SuiteSparse_long amd_l_valid` (`SuiteSparse_long` n\_row, `SuiteSparse_long` n\_col, const `SuiteSparse_long` Ap[], const `SuiteSparse_long` Ai[])
- `void amd_defaults` (`abip_float` Control[])
- `void amd_l_defaults` (`abip_float` Control[])
- `void amd_control` (`abip_float` Control[])
- `void amd_l_control` (`abip_float` Control[])
- `void amd_info` (`abip_float` ABIPInfo[])
- `void amd_l_info` (`abip_float` ABIPInfo[])

## Variables

- `EXTERN void` (\* `amd_malloc`) (size\_t)
- `EXTERN void` (\* `amd_free`) (void \*)
- `EXTERN void` (\* `amd_realloc`) (void \*, size\_t)
- `EXTERN void` (\* `amd_calloc`) (size\_t, size\_t)
- `EXTERN int` (\* `amd_printf`) (const char \*,...)

## 4.1.1 Macro Definition Documentation

### 4.1.1.1 AMD\_AGGRESSIVE

```
#define AMD_AGGRESSIVE 1 /* do aggressive absorption if Control [1] != 0 */
```

Definition at line 341 of file `amd.h`.

### 4.1.1.2 AMD\_CONTROL

```
#define AMD_CONTROL 5 /* size of Control array */
```

Definition at line 336 of file `amd.h`.

#### 4.1.1.3 AMD\_DATE

```
#define AMD_DATE "May 4, 2016"
```

Definition at line 394 of file [amd.h](#).

#### 4.1.1.4 AMD\_DEFAULT\_AGGRESSIVE

```
#define AMD_DEFAULT_AGGRESSIVE 1 /* do aggressive absorption by default */
```

Definition at line 345 of file [amd.h](#).

#### 4.1.1.5 AMD\_DEFAULT\_DENSE

```
#define AMD_DEFAULT_DENSE 10.0 /* default "dense" degree 10*sqrt(n) */
```

Definition at line 344 of file [amd.h](#).

#### 4.1.1.6 AMD\_DENSE

```
#define AMD_DENSE 0 /* "dense" if degree > Control [0] * sqrt (n) */
```

Definition at line 340 of file [amd.h](#).

#### 4.1.1.7 AMD\_DMAX

```
#define AMD_DMAX 13 /* max nz. in any column of L, incl. diagonal */
```

Definition at line 361 of file [amd.h](#).

#### 4.1.1.8 AMD\_INFO

```
#define AMD_INFO 20 /* size of Info array */
```

Definition at line 337 of file [amd.h](#).

#### 4.1.1.9 AMD\_INVALID

```
#define AMD_INVALID -2 /* input arguments are not valid */
```

Definition at line 369 of file [amd.h](#).

#### 4.1.1.10 AMD\_LNZ

```
#define AMD_LNZ 9 /* approx. nz in L, excluding the diagonal */
```

Definition at line 357 of file [amd.h](#).

#### 4.1.1.11 AMD\_MAIN\_VERSION

```
#define AMD_MAIN_VERSION 2
```

Definition at line 396 of file [amd.h](#).

#### 4.1.1.12 AMD\_MEMORY

```
#define AMD_MEMORY 7 /* amount of memory used by AMD */
```

Definition at line 355 of file [amd.h](#).

#### 4.1.1.13 AMD\_N

```
#define AMD_N 1 /* A is n-by-n */
```

Definition at line 349 of file [amd.h](#).

#### 4.1.1.14 AMD\_NCMPPA

```
#define AMD_NCMPPA 8 /* number of garbage collections in AMD */
```

Definition at line 356 of file [amd.h](#).



#### 4.1.1.15 AMD\_NDENSE

```
#define AMD_NDENSE 6 /* number of "dense" rows/columns in A */
```

Definition at line 354 of file [amd.h](#).

#### 4.1.1.16 AMD\_NDIV

```
#define AMD_NDIV 10 /* number of fl. point divides for LU and LDL' */
```

Definition at line 358 of file [amd.h](#).

#### 4.1.1.17 AMD\_NMULTSUBS\_LDL

```
#define AMD_NMULTSUBS_LDL 11 /* number of fl. point (*,-) pairs for LDL' */
```

Definition at line 359 of file [amd.h](#).

#### 4.1.1.18 AMD\_NMULTSUBS\_LU

```
#define AMD_NMULTSUBS_LU 12 /* number of fl. point (*,-) pairs for LU */
```

Definition at line 360 of file [amd.h](#).

#### 4.1.1.19 AMD\_NZ

```
#define AMD_NZ 2 /* number of nonzeros in A */
```

Definition at line 350 of file [amd.h](#).

#### 4.1.1.20 AMD\_NZ\_A\_PLUS\_AT

```
#define AMD_NZ_A_PLUS_AT 5 /* nz in A+A' */
```

Definition at line 353 of file [amd.h](#).

#### 4.1.1.21 AMD\_NZDIAG

```
#define AMD_NZDIAG 4 /* # of entries on diagonal */
```

Definition at line 352 of file [amd.h](#).

#### 4.1.1.22 AMD\_OK

```
#define AMD_OK 0 /* success */
```

Definition at line 367 of file [amd.h](#).

#### 4.1.1.23 AMD\_OK\_BUT\_JUMBLED

```
#define AMD_OK_BUT_JUMBLED
```

**Value:**

```
1      /* input matrix is OK for amd_order, but
 * columns were not sorted, and/or duplicate entries were present.  AMD had
 * to do extra work before ordering the matrix.  This is a warning, not an
 * error.  */
```

Definition at line 370 of file [amd.h](#).

#### 4.1.1.24 AMD\_OUT\_OF\_MEMORY

```
#define AMD_OUT_OF_MEMORY -1 /* malloc failed, or problem too large */
```

Definition at line 368 of file [amd.h](#).

#### 4.1.1.25 AMD\_STATUS

```
#define AMD_STATUS 0 /* return value of amd\_order and amd\_l\_order */
```

Definition at line 348 of file [amd.h](#).

#### 4.1.1.26 AMD\_SUB\_VERSION

```
#define AMD_SUB_VERSION 4
```

Definition at line 397 of file [amd.h](#).

#### 4.1.1.27 AMD\_SUBSUB\_VERSION

```
#define AMD_SUBSUB_VERSION 6
```

Definition at line 398 of file [amd.h](#).

#### 4.1.1.28 AMD\_SYMMETRY

```
#define AMD_SYMMETRY 3 /* symmetry of pattern (1 is sym., 0 is unsym.) */
```

Definition at line 351 of file [amd.h](#).

#### 4.1.1.29 AMD\_VERSION

```
#define AMD_VERSION AMD_VERSION_CODE(AMD_MAIN_VERSION,AMD_SUB_VERSION)
```

Definition at line 399 of file [amd.h](#).

#### 4.1.1.30 AMD\_VERSION\_CODE

```
#define AMD_VERSION_CODE(  
    main,  
    sub ) ((main) * 1000 + (sub))
```

Definition at line 395 of file [amd.h](#).

#### 4.1.1.31 EXTERN

```
#define EXTERN extern
```

Definition at line 311 of file [amd.h](#).

### 4.1.2 Function Documentation

#### 4.1.2.1 amd\_2()

```
void amd_2 (
    int n,
    int Pe[],
    int Iw[],
    int Len[],
    int iwlen,
    int pfree,
    int Nv[],
    int Next[],
    int Last[],
    int Head[],
    int Elen[],
    int Degree[],
    int W[],
    abip_float Control[],
    abip_float ABIPInfo[] )
```

#### 4.1.2.2 amd\_control()

```
void amd_control (
    abip_float Control[] )
```

#### 4.1.2.3 amd\_defaults()

```
void amd_defaults (
    abip_float Control[] )
```

#### 4.1.2.4 amd\_info()

```
void amd_info (
    abip_float ABIPInfo[] )
```

#### 4.1.2.5 amd\_l2()

```
void amd_l2 (
    SuiteSparse_long n,
    SuiteSparse_long Pe[],
    SuiteSparse_long Iw[],
    SuiteSparse_long Len[],
    SuiteSparse_long iwlen,
    SuiteSparse_long pfree,
    SuiteSparse_long Nv[],
    SuiteSparse_long Next[],
    SuiteSparse_long Last[],
    SuiteSparse_long Head[],
    SuiteSparse_long Elen[],
    SuiteSparse_long Degree[],
    SuiteSparse_long W[],
    abip_float Control[],
    abip_float ABIPInfo[] )
```

#### 4.1.2.6 amd\_l\_control()

```
void amd_l_control (
    abip_float Control[] )
```

#### 4.1.2.7 amd\_l\_defaults()

```
void amd_l_defaults (
    abip_float Control[] )
```

#### 4.1.2.8 amd\_l\_info()

```
void amd_l_info (
    abip_float ABIPInfo[] )
```

#### 4.1.2.9 amd\_l\_order()

```
SuiteSparse_long amd_l_order (
    SuiteSparse_long n,
    const SuiteSparse_long Ap[],
    const SuiteSparse_long Ai[],
    SuiteSparse_long P[],
    abip_float Control[],
    abip_float ABIPInfo[] )
```

#### 4.1.2.10 amd\_l\_valid()

```
SuiteSparse_long amd_l_valid (
    SuiteSparse_long n_row,
    SuiteSparse_long n_col,
    const SuiteSparse_long Ap[],
    const SuiteSparse_long Ai[] )
```

#### 4.1.2.11 amd\_order()

```
int amd_order (
    int n,
    const int Ap[],
    const int Ai[],
    int P[],
    abip_float Control[],
    abip_float ABIPInfo[] )
```

#### 4.1.2.12 amd\_valid()

```
int amd_valid (
    int n_row,
    int n_col,
    const int Ap[],
    const int Ai[] )
```

### 4.1.3 Variable Documentation

#### 4.1.3.1 amd\_calloc

```
EXTERN void *(* amd_calloc) (size_t, size_t) (
    size_t ,
    size_t )
```

Definition at line 317 of file [amd.h](#).

#### 4.1.3.2 amd\_free

```
EXTERN void(* amd_free) (void *) (
    void * )
```

Definition at line 315 of file [amd.h](#).

### 4.1.3.3 amd\_malloc

```
EXTERN void *(* amd_malloc) (size_t) (
    size_t )
```

Definition at line 314 of file [amd.h](#).

### 4.1.3.4 amd\_printf

```
EXTERN int(* amd_printf) (const char *,...) (
    const char * ,
    ... )
```

Definition at line 318 of file [amd.h](#).

### 4.1.3.5 amd\_realloc

```
EXTERN void *(* amd_realloc) (void *, size_t) (
    void * ,
    size_t )
```

Definition at line 316 of file [amd.h](#).

## 4.2 amd.h

[Go to the documentation of this file.](#)

```
00001 /* ===== */
00002 /* == AMD: approximate minimum degree ordering ===== */
00003 /* ===== */
00004
00005 /* ----- */
00006 /* AMD Version 2.4, Copyright (c) 1996-2013 by Timothy A. Davis, */
00007 /* Patrick R. Amestoy, and Iain S. Duff. See ../README.txt for License. */
00008 /* email: DrTimothyAldenDavis@gmail.com */
00009 /* ----- */
00010
00011 /* AMD finds a symmetric ordering P of a matrix A so that the Cholesky
00012 * factorization of P*A*P' has fewer nonzeros and takes less work than the
00013 * Cholesky factorization of A. If A is not symmetric, then it performs its
00014 * ordering on the matrix A+A'. Two sets of user-callable routines are
00015 * provided, one for int integers and the other for SuiteSparse_long integers.
00016 *
00017 * The method is based on the approximate minimum degree algorithm, discussed
00018 * in Amestoy, Davis, and Duff, "An approximate degree ordering algorithm",
00019 * SIAM Journal of Matrix Analysis and Applications, vol. 17, no. 4, pp.
00020 * 886-905, 1996. This package can perform both the AMD ordering (with
00021 * aggressive absorption), and the AMDBAR ordering (without aggressive
00022 * absorption) discussed in the above paper. This package differs from the
00023 * Fortran codes discussed in the paper:
00024 *
00025 * (1) it can ignore "dense" rows and columns, leading to faster run times
00026 * (2) it computes the ordering of A+A' if A is not symmetric
00027 * (3) it is followed by a depth-first post-ordering of the assembly tree
00028 * (or supernodal elimination tree)
00029 *
00030 * For historical reasons, the Fortran versions, amd.f and amdbar.f, have
00031 * been left (nearly) unchanged. They compute the identical ordering as
00032 * described in the above paper.
00033 */
```

```

00034
00035 #ifndef AMD_H
00036 #define AMD_H
00037
00038 /* make it easy for C++ programs to include AMD */
00039 #ifdef __cplusplus
00040 extern "C" {
00041 #endif
00042
00043 /* get the definition of size_t: */
00044 #include <stddef.h>
00045
00046 #include "SuiteSparse_config.h"
00047
00048 int amd_order                                /* returns AMD_OK, AMD_OK_BUT_JUMBLED,
00049                                           * AMD_INVALID, or AMD_OUT_OF_MEMORY */
00050 (
00051     int n,                                  /* A is n-by-n. n must be >= 0. */
00052     const int Ap [ ],                      /* column pointers for A, of size n+1 */
00053     const int Ai [ ],                      /* row indices of A, of size nz = Ap [n] */
00054     int P [ ],                             /* output permutation, of size n */
00055     abip_float Control [ ],                /* input Control settings, of size AMD_CONTROL */
00056     abip_float ABIPInfo [ ]               /* output ABIPInfo statistics, of size AMD_INFO */
00057 );
00058
00059 SuiteSparse_long amd_l_order               /* see above for description of arguments */
00060 (
00061     SuiteSparse_long n,
00062     const SuiteSparse_long Ap [ ],
00063     const SuiteSparse_long Ai [ ],
00064     SuiteSparse_long P [ ],
00065     abip_float Control [ ],
00066     abip_float ABIPInfo [ ]
00067 );
00068
00069 /* Input arguments (not modified):
00070 *
00071 *     n: the matrix A is n-by-n.
00072 *     Ap: an int/SuiteSparse_long array of size n+1, containing column
00073 *         pointers of A.
00074 *     Ai: an int/SuiteSparse_long array of size nz, containing the row
00075 *         indices of A, where nz = Ap [n].
00076 *     Control: a double array of size AMD_CONTROL, containing control
00077 *         parameters. Defaults are used if Control is ABIP_NULL.
00078 *
00079 * Output arguments (not defined on input):
00080 *
00081 *     P: an int/SuiteSparse_long array of size n, containing the output
00082 *         permutation. If row i is the kth pivot row, then P [k] = i. In
00083 *         MATLAB notation, the reordered matrix is A (P,P).
00084 *     ABIPInfo: a double array of size AMD_INFO, containing statistical
00085 *         information. Ignored if ABIPInfo is ABIP_NULL.
00086 *
00087 * On input, the matrix A is stored in column-oriented form. The row indices
00088 * of nonzero entries in column j are stored in Ai [Ap [j] ... Ap [j+1]-1].
00089 *
00090 * If the row indices appear in ascending order in each column, and there
00091 * are no duplicate entries, then amd_order is slightly more efficient in
00092 * terms of time and memory usage. If this condition does not hold, a copy
00093 * of the matrix is created (where these conditions do hold), and the copy is
00094 * ordered. This feature is new to v2.0 (v1.2 and earlier required this
00095 * condition to hold for the input matrix).
00096 *
00097 * Row indices must be in the range 0 to
00098 * n-1. Ap [0] must be zero, and thus nz = Ap [n] is the number of nonzeros
00099 * in A. The array Ap is of size n+1, and the array Ai is of size nz = Ap [n].
00100 * The matrix does not need to be symmetric, and the diagonal does not need to
00101 * be present (if diagonal entries are present, they are ignored except for
00102 * the output statistic Info [AMD_NZDIAG]). The arrays Ai and Ap are not
00103 * modified. This form of the Ap and Ai arrays to represent the nonzero
00104 * pattern of the matrix A is the same as that used internally by MATLAB.
00105 * If you wish to use a more flexible input structure, please see the
00106 * umfpack*_triplet_to_col routines in the UMFPACK package, at
00107 * http://www.suitesparse.com.
00108 *
00109 * Restrictions: n >= 0. Ap [0] = 0. Ap [j] <= Ap [j+1] for all j in the
00110 * range 0 to n-1. nz = Ap [n] >= 0. Ai [0..nz-1] must be in the range 0
00111 * to n-1. Finally, Ai, Ap, and P must not be NULL. If any of these
00112 * restrictions are not met, AMD returns AMD_INVALID.
00113 *
00114 * AMD returns:
00115 *
00116 *     AMD_OK if the matrix is valid and sufficient memory can be allocated to
00117 *         perform the ordering.
00118 *
00119 *     AMD_OUT_OF_MEMORY if not enough memory can be allocated.
00120 *

```



```

00121 *      AMD_INVALID if the input arguments n, Ap, Ai are invalid, or if P is
00122 *      NULL.
00123 *
00124 *      AMD_OK_BUT_JUMBLED if the matrix had unsorted columns, and/or duplicate
00125 *      entries, but was otherwise valid.
00126 *
00127 * The AMD routine first forms the pattern of the matrix A+A', and then
00128 * computes a fill-reducing ordering, P. If P [k] = i, then row/column i of
00129 * the original is the kth pivotal row. In MATLAB notation, the permuted
00130 * matrix is A (P,P), except that 0-based indexing is used instead of the
00131 * 1-based indexing in MATLAB.
00132 *
00133 * The Control array is used to set various parameters for AMD. If a NULL
00134 * pointer is passed, default values are used. The Control array is not
00135 * modified.
00136 *
00137 *      Control [AMD_DENSE]: controls the threshold for "dense" rows/columns.
00138 *      A dense row/column in A+A' can cause AMD to spend a lot of time in
00139 *      ordering the matrix. If Control [AMD_DENSE] >= 0, rows/columns
00140 *      with more than Control [AMD_DENSE] * sqrt (n) entries are ignored
00141 *      during the ordering, and placed last in the output order. The
00142 *      default value of Control [AMD_DENSE] is 10. If negative, no
00143 *      rows/columns are treated as "dense". Rows/columns with 16 or
00144 *      fewer off-diagonal entries are never considered "dense".
00145 *
00146 *      Control [AMD_AGGRESSIVE]: controls whether or not to use aggressive
00147 *      absorption, in which a prior element is absorbed into the current
00148 *      element if it is a subset of the current element, even if it is not
00149 *      adjacent to the current pivot element (refer to Amestoy, Davis,
00150 *      & Duff, 1996, for more details). The default value is nonzero,
00151 *      which means to perform aggressive absorption. This nearly always
00152 *      leads to a better ordering (because the approximate degrees are
00153 *      more accurate) and a lower execution time. There are cases where
00154 *      it can lead to a slightly worse ordering, however. To turn it off,
00155 *      set Control [AMD_AGGRESSIVE] to 0.
00156 *
00157 *      Control [2..4] are not used in the current version, but may be used in
00158 *      future versions.
00159 *
00160 * The ABIPInfo array provides statistics about the ordering on output. If it is
00161 * not present, the statistics are not returned. This is not an error
00162 * condition.
00163 *
00164 *      ABIPInfo [AMD_STATUS]: the return value of AMD, either AMD_OK,
00165 *      AMD_OK_BUT_JUMBLED, AMD_OUT_OF_MEMORY, or AMD_INVALID.
00166 *
00167 *      ABIPInfo [AMD_N]: n, the size of the input matrix
00168 *
00169 *      ABIPInfo [AMD_NZ]: the number of nonzeros in A, nz = Ap [n]
00170 *
00171 *      ABIPInfo [AMD_SYMMETRY]: the symmetry of the matrix A. It is the number
00172 *      of "matched" off-diagonal entries divided by the total number of
00173 *      off-diagonal entries. An entry A(i,j) is matched if A(j,i) is also
00174 *      an entry, for any pair (i,j) for which i != j. In MATLAB notation,
00175 *      S = spones (A) ;
00176 *      B = tril (S, -1) + triu (S, 1) ;
00177 *      symmetry = nnz (B & B') / nnz (B) ;
00178 *
00179 *      ABIPInfo [AMD_NZDIAG]: the number of entries on the diagonal of A.
00180 *
00181 *      ABIPInfo [AMD_NZ_A_PLUS_AT]: the number of nonzeros in A+A', excluding the
00182 *      diagonal. If A is perfectly symmetric (Info [AMD_SYMMETRY] = 1)
00183 *      with a fully nonzero diagonal, then Info [AMD_NZ_A_PLUS_AT] = nz-n
00184 *      (the smallest possible value). If A is perfectly unsymmetric
00185 *      (Info [AMD_SYMMETRY] = 0, for an upper triangular matrix, for
00186 *      example) with no diagonal, then Info [AMD_NZ_A_PLUS_AT] = 2*nz
00187 *      (the largest possible value).
00188 *
00189 *      ABIPInfo [AMD_NDENSE]: the number of "dense" rows/columns of A+A' that were
00190 *      removed from A prior to ordering. These are placed last in the
00191 *      output order P.
00192 *
00193 *      ABIPInfo [AMD_MEMORY]: the amount of memory used by AMD, in bytes. In the
00194 *      current version, this is 1.2 * Info [AMD_NZ_A_PLUS_AT] + 9*n
00195 *      times the size of an integer. This is at most 2.4*nz + 9n. This
00196 *      excludes the size of the input arguments Ai, Ap, and P, which have
00197 *      a total size of nz + 2*n + 1 integers.
00198 *
00199 *      ABIPInfo [AMD_NCMPA]: the number of garbage collections performed.
00200 *
00201 *      ABIPInfo [AMD_LNZ]: the number of nonzeros in L (excluding the diagonal).
00202 *      This is a slight upper bound because mass elimination is combined
00203 *      with the approximate degree update. It is a rough upper bound if
00204 *      there are many "dense" rows/columns. The rest of the statistics,
00205 *      below, are also slight or rough upper bounds, for the same reasons.
00206 *      The post-ordering of the assembly tree might also not exactly
00207 *      correspond to a true elimination tree postordering.

```

```

00208 *
00209 *      ABIPInfo [AMD_NDIV]: the number of divide operations for a subsequent LDL'
00210 *      or LU factorization of the permuted matrix A (P,P).
00211 *
00212 *      ABIPInfo [AMD_NMULTSUBS_LDL]: the number of multiply-subtract pairs for a
00213 *      subsequent LDL' factorization of A (P,P).
00214 *
00215 *      ABIPInfo [AMD_NMULTSUBS_LU]: the number of multiply-subtract pairs for a
00216 *      subsequent LU factorization of A (P,P), assuming that no numerical
00217 *      pivoting is required.
00218 *
00219 *      ABIPInfo [AMD_DMAX]: the maximum number of nonzeros in any column of L,
00220 *      including the diagonal.
00221 *
00222 *      ABIPInfo [14..19] are not used in the current version, but may be used in
00223 *      future versions.
00224 */
00225
00226 /* ----- */
00227 /* direct interface to AMD */
00228 /* ----- */
00229
00230 /* amd_2 is the primary AMD ordering routine. It is not meant to be
00231 * user-callable because of its restrictive inputs and because it destroys
00232 * the user's input matrix. It does not check its inputs for errors, either.
00233 * However, if you can work with these restrictions it can be faster than
00234 * amd_order and use less memory (assuming that you can create your own copy
00235 * of the matrix for AMD to destroy). Refer to AMD/Source/amd_2.c for a
00236 * description of each parameter. */
00237
00238 void amd_2
00239 (
00240     int n,
00241     int Pe [ ],
00242     int Iw [ ],
00243     int Len [ ],
00244     int iwlen,
00245     int pfree,
00246     int Nv [ ],
00247     int Next [ ],
00248     int Last [ ],
00249     int Head [ ],
00250     int Elen [ ],
00251     int Degree [ ],
00252     int W [ ],
00253     abip_float Control [ ],
00254     abip_float ABIPInfo [ ]
00255 );
00256
00257 void amd_l2
00258 (
00259     SuiteSparse_long n,
00260     SuiteSparse_long Pe [ ],
00261     SuiteSparse_long Iw [ ],
00262     SuiteSparse_long Len [ ],
00263     SuiteSparse_long iwlen,
00264     SuiteSparse_long pfree,
00265     SuiteSparse_long Nv [ ],
00266     SuiteSparse_long Next [ ],
00267     SuiteSparse_long Last [ ],
00268     SuiteSparse_long Head [ ],
00269     SuiteSparse_long Elen [ ],
00270     SuiteSparse_long Degree [ ],
00271     SuiteSparse_long W [ ],
00272     abip_float Control [ ],
00273     abip_float ABIPInfo [ ]
00274 );
00275
00276 /* ----- */
00277 /* amd_valid */
00278 /* ----- */
00279
00280 /* Returns AMD_OK or AMD_OK_BUT_JUMBLED if the matrix is valid as input to
00281 * amd_order; the latter is returned if the matrix has unsorted and/or
00282 * duplicate row indices in one or more columns. Returns AMD_INVALID if the
00283 * matrix cannot be passed to amd_order. For amd_order, the matrix must also
00284 * be square. The first two arguments are the number of rows and the number
00285 * of columns of the matrix. For its use in AMD, these must both equal n.
00286 *
00287 * NOTE: this routine returned TRUE/FALSE in v1.2 and earlier.
00288 */
00289
00290 int amd_valid
00291 (
00292     int n_row,                /* # of rows */
00293     int n_col,                /* # of columns */
00294     const int Ap [ ],         /* column pointers, of size n_col+1 */

```

```

00295         const int Ai [ ]           /* row indices, of size Ap [n_col] */
00296     );
00297
00298 SuiteSparse_long amd_l_valid
00299 (
00300     SuiteSparse_long n_row,
00301     SuiteSparse_long n_col,
00302     const SuiteSparse_long Ap [ ],
00303     const SuiteSparse_long Ai [ ]
00304 );
00305
00306 /* ----- */
00307 /* AMD memory manager and printf routines */
00308 /* ----- */
00309
00310 #ifndef EXTERN
00311 #define EXTERN extern
00312 #endif
00313
00314 EXTERN void (*amd_malloc) (size_t);           /* pointer to malloc */
00315 EXTERN void (*amd_free) (void *);            /* pointer to free */
00316 EXTERN void (*amd_realloc) (void *, size_t);  /* pointer to realloc */
00317 EXTERN void (*amd_calloc) (size_t, size_t);  /* pointer to calloc */
00318 EXTERN int (*amd_printf) (const char *, ...); /* pointer to printf */
00319
00320 /* ----- */
00321 /* AMD Control and Info arrays */
00322 /* ----- */
00323
00324 /* amd_defaults: sets the default control settings */
00325 void amd_defaults (abip_float Control [ ] );
00326 void amd_l_defaults (abip_float Control [ ] );
00327
00328 /* amd_control: prints the control settings */
00329 void amd_control (abip_float Control [ ] );
00330 void amd_l_control (abip_float Control [ ] );
00331
00332 /* amd_info: prints the statistics */
00333 void amd_info (abip_float ABIPInfo [ ] );
00334 void amd_l_info (abip_float ABIPInfo [ ] );
00335
00336 #define AMD_CONTROL 5           /* size of Control array */
00337 #define AMD_INFO 20            /* size of Info array */
00338
00339 /* contents of Control */
00340 #define AMD_DENSE 0             /* "dense" if degree > Control [0] * sqrt (n) */
00341 #define AMD_AGGRESSIVE 1        /* do aggressive absorption if Control [1] != 0 */
00342
00343 /* default Control settings */
00344 #define AMD_DEFAULT_DENSE 10.0  /* default "dense" degree 10*sqrt(n) */
00345 #define AMD_DEFAULT_AGGRESSIVE 1 /* do aggressive absorption by default */
00346
00347 /* contents of Info */
00348 #define AMD_STATUS 0            /* return value of amd_order and amd_l_order */
00349 #define AMD_N 1                 /* A is n-by-n */
00350 #define AMD_NZ 2                /* number of nonzeros in A */
00351 #define AMD_SYMMETRY 3          /* symmetry of pattern (1 is sym., 0 is unsym.) */
00352 #define AMD_NZDIAG 4           /* # of entries on diagonal */
00353 #define AMD_NZ_A_PLUS_AT 5      /* nz in A+A' */
00354 #define AMD_NDENSE 6            /* number of "dense" rows/columns in A */
00355 #define AMD_MEMORY 7            /* amount of memory used by AMD */
00356 #define AMD_NCMPA 8             /* number of garbage collections in AMD */
00357 #define AMD_LNZ 9               /* approx. nz in L, excluding the diagonal */
00358 #define AMD_NDIV 10             /* number of fl. point divides for LU and LDL' */
00359 #define AMD_NMULTSUBS_LDL 11    /* number of fl. point (*,-) pairs for LDL' */
00360 #define AMD_NMULTSUBS_LU 12     /* number of fl. point (*,-) pairs for LU */
00361 #define AMD_DMAX 13             /* max nz. in any column of L, incl. diagonal */
00362
00363 /* ----- */
00364 /* return values of AMD */
00365 /* ----- */
00366
00367 #define AMD_OK 0                /* success */
00368 #define AMD_OUT_OF_MEMORY -1    /* malloc failed, or problem too large */
00369 #define AMD_INVALID -2         /* input arguments are not valid */
00370 #define AMD_OK_BUT_JUMBLED 1    /* input matrix is OK for amd_order, but
00371  * columns were not sorted, and/or duplicate entries were present. AMD had
00372  * to do extra work before ordering the matrix. This is a warning, not an
00373  * error. */
00374
00375 /* ===== */
00376 /* == AMD version == */
00377 /* ===== */
00378
00379 /* AMD Version 1.2 and later include the following definitions.
00380  * As an example, to test if the version you are using is 1.2 or later:
00381  *

```

```

00382 * #ifdef AMD_VERSION
00383 *     if (AMD_VERSION >= AMD_VERSION_CODE (1,2)) ...
00384 * #endif
00385 *
00386 * This also works during compile-time:
00387 *
00388 *     #if defined(AMD_VERSION) && (AMD_VERSION >= AMD_VERSION_CODE (1,2))
00389 *         printf ("This is version 1.2 or later\n") ;
00390 *     #else
00391 *         printf ("This is an early version\n") ;
00392 *     #endif
00393 *
00394 * Versions 1.1 and earlier of AMD do not include a #define'd version number.
00395 */
00396
00397 #define AMD_DATE "May 4, 2016"
00398 #define AMD_VERSION_CODE(main,sub) ((main) * 1000 + (sub))
00399 #define AMD_MAIN_VERSION 2
00400 #define AMD_SUB_VERSION 4
00401 #define AMD_SUBSUB_VERSION 6
00402 #define AMD_VERSION AMD_VERSION_CODE(AMD_MAIN_VERSION,AMD_SUB_VERSION)
00403
00404 #ifdef __cplusplus
00405 }
00406 #endif
00407
00408 #endif

```

## 4.3 amd/amd\_1.c File Reference

```
#include "amd_internal.h"
```

### Functions

- **GLOBAL** void **AMD\_1** (Int n, const Int Ap[], const Int Ai[], Int P[], Int Pinv[], Int Len[], Int slen, Int S[], abip\_float Control[], abip\_float ABIPInfo[])

### 4.3.1 Function Documentation

#### 4.3.1.1 AMD\_1()

```

GLOBAL void AMD_1 (
    Int n,
    const Int Ap[],
    const Int Ai[],
    Int P[],
    Int Pinv[],
    Int Len[],
    Int slen,
    Int S[],
    abip_float Control[],
    abip_float ABIPInfo[] )

```

Definition at line 29 of file [amd\\_1.c](#).

## 4.4 amd\_1.c

[Go to the documentation of this file.](#)

```

00001 /* ===== */
00002 /* == AMD_1 == */
00003 /* ===== */
00004
00005 /* ----- */
00006 /* AMD, Copyright (c) Timothy A. Davis, */
00007 /* Patrick R. Amestoy, and Iain S. Duff. See ../README.txt for License. */
00008 /* email: DrTimothyAldenDavis@gmail.com */
00009 /* ----- */
00010
00011 /* AMD_1: Construct A+A' for a sparse matrix A and perform the AMD ordering.
00012  *
00013  * The n-by-n sparse matrix A can be unsymmetric. It is stored in MATLAB-style
00014  * compressed-column form, with sorted row indices in each column, and no
00015  * duplicate entries. Diagonal entries may be present, but they are ignored.
00016  * Row indices of column j of A are stored in Ai [Ap [j] ... Ap [j+1]-1].
00017  * Ap [0] must be zero, and nz = Ap [n] is the number of entries in A. The
00018  * size of the matrix, n, must be greater than or equal to zero.
00019  *
00020  * This routine must be preceded by a call to AMD_aat, which computes the
00021  * number of entries in each row/column in A+A', excluding the diagonal.
00022  * Len [j], on input, is the number of entries in row/column j of A+A'. This
00023  * routine constructs the matrix A+A' and then calls AMD_2. No error checking
00024  * is performed (this was done in AMD_valid).
00025  */
00026
00027 #include "amd_internal.h"
00028
00029 GLOBAL void AMD_1
00030 (
00031     Int n, /* n > 0 */
00032     const Int Ap [ ], /* input of size n+1, not modified */
00033     const Int Ai [ ], /* input of size nz = Ap [n], not modified */
00034     Int P [ ], /* size n output permutation */
00035     Int Pinv [ ], /* size n output inverse permutation */
00036     Int Len [ ], /* size n input, undefined on output */
00037     Int slen, /* slen >= sum (Len [0..n-1]) + 7n,
00038              * ideally slen = 1.2 * sum (Len) + 8n */
00039     Int S [ ], /* size slen workspace */
00040     abip_float Control [ ], /* input array of size AMD_CONTROL */
00041     abip_float ABIPInfo [ ] /* output array of size AMD_INFO */
00042 )
00043 {
00044     Int i;
00045     Int j;
00046     Int k;
00047     Int p;
00048     Int pfree;
00049     Int iwlen;
00050     Int pj;
00051     Int p1;
00052     Int p2;
00053     Int pj2;
00054
00055     Int *Iw;
00056     Int *Pe;
00057     Int *Nv;
00058     Int *Head;
00059     Int *Elen;
00060     Int *Degree;
00061     Int *s;
00062     Int *W;
00063     Int *Sp;
00064     Int *Tp;
00065
00066     /* ----- */
00067     /* construct the matrix for AMD_2 */
00068     /* ----- */
00069
00070     ASSERT (n > 0) ;
00071
00072     iwlen = slen - 6*n ;
00073     s = S ;
00074     Pe = s ;
00075     s += n ;
00076     Nv = s ;
00077     s += n ;
00078     Head = s ;
00079     s += n ;
00080     Elen = s ;
00081     s += n ;
00082     Degree = s ;

```

```

00083     s += n ;
00084     W = s ;
00085     s += n ;
00086     Iw = s ;
00087     s += iwlen ;
00088
00089     ASSERT (AMD_valid (n, n, Ap, Ai) == AMD_OK) ;
00090
00091     /* construct the pointers for A+A' */
00092     Sp = Nv ;           /* use Nv and W as workspace for Sp and Tp [ */
00093     Tp = W ;
00094     pfree = 0 ;
00095
00096     for (j = 0 ; j < n ; j++)
00097     {
00098         Pe [j] = pfree ;
00099         Sp [j] = pfree ;
00100         pfree += Len [j] ;
00101     }
00102
00103     /* Note that this restriction on iwlen is slightly more restrictive than
00104     * what is strictly required in AMD_2. AMD_2 can operate with no elbow
00105     * room at all, but it will be very slow. For better performance, at
00106     * least size-n elbow room is enforced. */
00107     ASSERT (iwlen >= pfree + n) ;
00108
00109 #ifndef NDEBUG
00110     for (p = 0 ; p < iwlen ; p++) Iw [p] = EMPTY ;
00111 #endif
00112
00113     for (k = 0 ; k < n ; k++)
00114     {
00115         AMD_DEBUG1 (("Construct row/column k= "ID" of A+A'\n", k)) ;
00116         p1 = Ap [k] ;
00117         p2 = Ap [k+1] ;
00118
00119         /* construct A+A' */
00120         for (p = p1 ; p < p2 ; )
00121         {
00122             /* scan the upper triangular part of A */
00123             j = Ai [p] ;
00124             ASSERT (j >= 0 && j < n) ;
00125
00126             if (j < k)
00127             {
00128                 /* entry A (j,k) in the strictly upper triangular part */
00129                 ASSERT (Sp [j] < (j == n-1 ? pfree : Pe [j+1])) ;
00130                 ASSERT (Sp [k] < (k == n-1 ? pfree : Pe [k+1])) ;
00131                 Iw [Sp [j]++] = k ;
00132                 Iw [Sp [k]++] = j ;
00133                 p++ ;
00134             }
00135             else if (j == k)
00136             {
00137                 /* skip the diagonal */
00138                 p++ ;
00139                 break ;
00140             }
00141             else /* j > k */
00142             {
00143                 /* first entry below the diagonal */
00144                 break ;
00145             }
00146
00147             /* scan lower triangular part of A, in column j until reaching
00148             * row k. Start where last scan left off. */
00149             ASSERT (Ap [j] <= Tp [j] && Tp [j] <= Ap [j+1]) ;
00150             pj2 = Ap [j+1] ;
00151
00152             for (pj = Tp [j] ; pj < pj2 ; )
00153             {
00154                 i = Ai [pj] ;
00155                 ASSERT (i >= 0 && i < n) ;
00156
00157                 if (i < k)
00158                 {
00159                     /* A (i,j) is only in the lower part, not in upper */
00160                     ASSERT (Sp [i] < (i == n-1 ? pfree : Pe [i+1])) ;
00161                     ASSERT (Sp [j] < (j == n-1 ? pfree : Pe [j+1])) ;
00162                     Iw [Sp [i]++] = j ;
00163                     Iw [Sp [j]++] = i ;
00164                     pj++ ;
00165                 }
00166                 else if (i == k)
00167                 {
00168                     /* entry A (k,j) in lower part and A (j,k) in upper */
00169                     pj++ ;

```

```

00170             break ;
00171         }
00172         else /* i > k */
00173         {
00174             /* consider this entry later, when k advances to i */
00175             break ;
00176         }
00177     }
00178     Tp [j] = pj ;
00179 }
00180 Tp [k] = p ;
00181 }
00182
00183 /* clean up, for remaining mismatched entries */
00184 for (j = 0 ; j < n ; j++)
00185 {
00186     for (pj = Tp [j] ; pj < Ap [j+1] ; pj++)
00187     {
00188         i = Ai [pj] ;
00189         ASSERT (i >= 0 && i < n) ;
00190
00191         /* A (i,j) is only in the lower part, not in upper */
00192         ASSERT (Sp [i] < (i == n-1 ? pfree : Pe [i+1])) ;
00193         ASSERT (Sp [j] < (j == n-1 ? pfree : Pe [j+1])) ;
00194         Iw [Sp [i]++] = j ;
00195         Iw [Sp [j]++] = i ;
00196     }
00197 }
00198
00199 #ifndef NDEBUG
00200
00201     for (j = 0 ; j < n-1 ; j++) ASSERT (Sp [j] == Pe [j+1]) ;
00202     ASSERT (Sp [n-1] == pfree) ;
00203
00204 #endif
00205
00206     /* Tp and Sp no longer needed */
00207
00208     /* ----- */
00209     /* order the matrix */
00210     /* ----- */
00211
00212     AMD_2 (n, Pe, Iw, Len, iwlen, pfree, Nv, Pinv, P, Head, Elen, Degree, W, Control, ABIPInfo) ;
00213 }

```

## 4.5 amd/amd\_2.c File Reference

```
#include "amd_internal.h"
```

### Functions

- **GLOBAL** void **AMD\_2** (Int n, Int Pe[], Int lw[], Int Len[], Int iwlen, Int pfree, Int Nv[], Int Next[], Int Last[], Int Head[], Int Elen[], Int Degree[], Int W[], abip\_float Control[], abip\_float ABIPInfo[])

### 4.5.1 Function Documentation

#### 4.5.1.1 AMD\_2()

```

GLOBAL void AMD_2 (
    Int n,
    Int Pe[],
    Int Iw[],

```

```

Int Len[ ],
Int iwlen,
Int pfree,
Int Nv[ ],
Int Next[ ],
Int Last[ ],
Int Head[ ],
Int Elen[ ],
Int Degree[ ],
Int W[ ],
abip_float Control[ ],
abip_float ABIPInfo[ ] )

```

Definition at line 43 of file [amd\\_2.c](#).

## 4.6 amd\_2.c

[Go to the documentation of this file.](#)

```

00001 /* ===== */
00002 /* === AMD_2 ===== */
00003 /* ===== */
00004
00005 /* ----- */
00006 /* AMD, Copyright (c) Timothy A. Davis, */
00007 /* Patrick R. Amestoy, and Iain S. Duff. See ../README.txt for License. */
00008 /* email: DrTimothyAldenDavis@gmail.com */
00009 /* ----- */
00010
00011 /* AMD_2: performs the AMD ordering on a symmetric sparse matrix A, followed
00012 * by a postordering (via depth-first search) of the assembly tree using the
00013 * AMD_postorder routine.
00014 */
00015
00016 #include "amd_internal.h"
00017
00018 /* ===== */
00019 /* === clear_flag ===== */
00020 /* ===== */
00021
00022 static Int clear_flag (Int wflg, Int wbig, Int W [ ], Int n)
00023 {
00024     Int x ;
00025     if (wflg < 2 || wflg >= wbig)
00026     {
00027         for (x = 0 ; x < n ; x++)
00028         {
00029             if (W [x] != 0) W [x] = 1 ;
00030         }
00031         wflg = 2 ;
00032     }
00033
00034     /* at this point, W [0..n-1] < wflg holds */
00035     return (wflg) ;
00036 }
00037
00038
00039 /* ===== */
00040 /* === AMD_2 ===== */
00041 /* ===== */
00042
00043 GLOBAL void AMD_2
00044 (
00045     Int n, /* A is n-by-n, where n > 0 */
00046     Int Pe [ ], /* Pe [0..n-1]: index in Iw of row i on input */
00047     Int Iw [ ], /* workspace of size iwlen. Iw [0..pfree-1] holds the matrix on
input */
00048     Int Len [ ], /* Len [0..n-1]: length for row/column i on input */
00049     Int iwlen, /* length of Iw. iwlen >= pfree + n */
00050     Int pfree, /* Iw [pfree ... iwlen-1] is empty on input */
00051
00052     /* 7 size-n workspaces, not defined on input: */
00053     Int Nv [ ], /* the size of each supernode on output */
00054     Int Next [ ], /* the output inverse permutation */
00055     Int Last [ ], /* the output permutation */
00056     Int Head [ ],

```



```

00057         Int Elen [ ],                /* the size columns of L for each supernode */
00058         Int Degree [ ],
00059         Int W [ ],
00060
00061         /* control parameters and output statistics */
00062         abip_float Control [ ],      /* array of size AMD_CONTROL */
00063         abip_float ABIPInfo [ ]     /* array of size AMD_INFO */
00064 )
00065 {
00066     /*
00067     * Given a representation of the nonzero pattern of a symmetric matrix, A,
00068     * (excluding the diagonal) perform an approximate minimum (UMFPACK/MA38-style)
00069     * degree ordering to compute a pivot order such that the introduction of
00070     * nonzeros (fill-in) in the Cholesky factors  $A = LL'$  is kept low. At each
00071     * step, the pivot selected is the one with the minimum UMFPACK/MA38-style
00072     * upper-bound on the external degree. This routine can optionally perform
00073     * aggressive absorption (as done by MC47B in the Harwell Subroutine
00074     * Library).
00075     *
00076     * The approximate degree algorithm implemented here is the symmetric analog of
00077     * the degree update algorithm in MA38 and UMFPACK (the Unsymmetric-pattern
00078     * MultiFrontal PACKage, both by Davis and Duff). The routine is based on the
00079     * MA27 minimum degree ordering algorithm by Iain Duff and John Reid.
00080     *
00081     * This routine is a translation of the original AMDBAR and MC47B routines,
00082     * in Fortran, with the following modifications:
00083     *
00084     * (1) dense rows/columns are removed prior to ordering the matrix, and placed
00085     * last in the output order. The presence of a dense row/column can
00086     * increase the ordering time by up to  $O(n^2)$ , unless they are removed
00087     * prior to ordering.
00088     *
00089     * (2) the minimum degree ordering is followed by a postordering (depth-first
00090     * search) of the assembly tree. Note that mass elimination (discussed
00091     * below) combined with the approximate degree update can lead to the mass
00092     * elimination of nodes with lower exact degree than the current pivot
00093     * element. No additional fill-in is caused in the representation of the
00094     * Schur complement. The mass-eliminated nodes merge with the current
00095     * pivot element. They are ordered prior to the current pivot element.
00096     * Because they can have lower exact degree than the current element, the
00097     * merger of two or more of these nodes in the current pivot element can
00098     * lead to a single element that is not a "fundamental supernode". The
00099     * diagonal block can have zeros in it. Thus, the assembly tree used here
00100     * is not guaranteed to be the precise supernodal elimination tree (with
00101     * "fundamental" supernodes), and the postordering performed by this
00102     * routine is not guaranteed to be a precise postordering of the
00103     * elimination tree.
00104     *
00105     * (3) input parameters are added, to control aggressive absorption and the
00106     * detection of "dense" rows/columns of A.
00107     *
00108     * (4) additional statistical information is returned, such as the number of
00109     * nonzeros in L, and the flop counts for subsequent LDL' and LU
00110     * factorizations. These are slight upper bounds, because of the mass
00111     * elimination issue discussed above.
00112     *
00113     * (5) additional routines are added to interface this routine to MATLAB
00114     * to provide a simple C-callable user-interface, to check inputs for
00115     * errors, compute the symmetry of the pattern of A and the number of
00116     * nonzeros in each row/column of  $A+A'$ , to compute the pattern of  $A+A'$ ,
00117     * to perform the assembly tree postordering, and to provide debugging
00118     * output. Many of these functions are also provided by the Fortran
00119     * Harwell Subroutine Library routine MC47A.
00120     *
00121     * (6) both int and SuiteSparse_long versions are provided. In the
00122     * descriptions below and integer is and int or SuiteSparse_long depending
00123     * on which version is being used.
00124     *
00125     *****
00126     ***** CAUTION: ARGUMENTS ARE NOT CHECKED FOR ERRORS ON INPUT. *****
00127     *****
00128     ** If you want error checking, a more versatile input format, and a **
00129     ** simpler user interface, use amd_order or amd_l_order instead. **
00130     ** This routine is not meant to be user-callable. **
00131     *****
00132
00133     * -----
00134     * References:
00135     * -----
00136     *
00137     * [1] Timothy A. Davis and Iain Duff, "An unsymmetric-pattern multifrontal
00138     * method for sparse LU factorization", SIAM J. Matrix Analysis and
00139     * Applications, vol. 18, no. 1, pp. 140-158. Discusses UMFPACK / MA38,
00140     * which first introduced the approximate minimum degree used by this
00141     * routine.
00142     *
00143     *

```

```

00144 * [2] Patrick Amestoy, Timothy A. Davis, and Iain S. Duff, "An approximate
00145 * minimum degree ordering algorithm," SIAM J. Matrix Analysis and
00146 * Applications, vol. 17, no. 4, pp. 886-905, 1996. Discusses AMDBAR and
00147 * MC47B, which are the Fortran versions of this routine.
00148 *
00149 * [3] Alan George and Joseph Liu, "The evolution of the minimum degree
00150 * ordering algorithm," SIAM Review, vol. 31, no. 1, pp. 1-19, 1989.
00151 * We list below the features mentioned in that paper that this code
00152 * includes:
00153 *
00154 * mass elimination:
00155 *     Yes. MA27 relied on supervariable detection for mass elimination.
00156 *
00157 * indistinguishable nodes:
00158 *     Yes (we call these "supervariables"). This was also in the MA27
00159 *     code - although we modified the method of detecting them (the
00160 *     previous hash was the true degree, which we no longer keep track
00161 *     of). A supervariable is a set of rows with identical nonzero
00162 *     pattern. All variables in a supervariable are eliminated together.
00163 *     Each supervariable has as its numerical name that of one of its
00164 *     variables (its principal variable).
00165 *
00166 * quotient graph representation:
00167 *     Yes. We use the term "element" for the cliques formed during
00168 *     elimination. This was also in the MA27 code. The algorithm can
00169 *     operate in place, but it will work more efficiently if given some
00170 *     "elbow room."
00171 *
00172 * element absorption:
00173 *     Yes. This was also in the MA27 code.
00174 *
00175 * external degree:
00176 *     Yes. The MA27 code was based on the true degree.
00177 *
00178 * incomplete degree update and multiple elimination:
00179 *     No. This was not in MA27, either. Our method of degree update
00180 *     within MC47B is element-based, not variable-based. It is thus
00181 *     not well-suited for use with incomplete degree update or multiple
00182 *     elimination.
00183 *
00184 * Authors, and Copyright (C) 2004 by:
00185 * Timothy A. Davis, Patrick Amestoy, Iain S. Duff, John K. Reid.
00186 *
00187 * Acknowledgements: This work (and the UMFPACK package) was supported by the
00188 * National Science Foundation (ASC-9111263, DMS-9223088, and CCR-0203270).
00189 * The UMFPACK/MA38 approximate degree update algorithm, the unsymmetric analog
00190 * which forms the basis of AMD, was developed while Tim Davis was supported by
00191 * CERFACS (Toulouse, France) in a post-doctoral position. This C version, and
00192 * the etree postorder, were written while Tim Davis was on sabbatical at
00193 * Stanford University and Lawrence Berkeley National Laboratory.
00194 *
00195 * -----
00196 * INPUT ARGUMENTS (unaltered):
00197 * -----
00198 *
00199 * n: The matrix order. Restriction: n >= 1.
00200 *
00201 * iwlen: The size of the Iw array. On input, the matrix is stored in
00202 * Iw [0..pfree-1]. However, Iw [0..iwlen-1] should be slightly larger
00203 * than what is required to hold the matrix, at least iwlen >= pfree + n.
00204 * Otherwise, excessive compressions will take place. The recommended
00205 * value of iwlen is 1.2 * pfree + n, which is the value used in the
00206 * user-callable interface to this routine (amd_order.c). The algorithm
00207 * will not run at all if iwlen < pfree. Restriction: iwlen >= pfree + n.
00208 * Note that this is slightly more restrictive than the actual minimum
00209 * (iwlen >= pfree), but AMD_2 will be very slow with no elbow room.
00210 * Thus, this routine enforces a bare minimum elbow room of size n.
00211 *
00212 * pfree: On input the tail end of the array, Iw [pfree..iwlen-1], is empty,
00213 * and the matrix is stored in Iw [0..pfree-1]. During execution,
00214 * additional data is placed in Iw, and pfree is modified so that
00215 * Iw [pfree..iwlen-1] is always the unused part of Iw.
00216 *
00217 * Control: A abip_float array of size AMD_CONTROL containing input parameters
00218 * that affect how the ordering is computed. If ABIP_NULL, then default
00219 * settings are used.
00220 *
00221 * Control [AMD_DENSE] is used to determine whether or not a given input
00222 * row is "dense". A row is "dense" if the number of entries in the row
00223 * exceeds Control [AMD_DENSE] times sqrt (n), except that rows with 16 or
00224 * fewer entries are never considered "dense". To turn off the detection
00225 * of dense rows, set Control [AMD_DENSE] to a negative number, or to a
00226 * number larger than sqrt (n). The default value of Control [AMD_DENSE]
00227 * is AMD_DEFAULT_DENSE, which is defined in amd.h as 10.
00228 *
00229 * Control [AMD_AGGRESSIVE] is used to determine whether or not aggressive
00230 * absorption is to be performed. If nonzero, then aggressive absorption

```

```

00231 * is performed (this is the default).
00232
00233 * -----
00234 * INPUT/OUTPUT ARGUMENTS:
00235 * -----
00236 *
00237 * Pe: An integer array of size n. On input, Pe [i] is the index in Iw of
00238 * the start of row i. Pe [i] is ignored if row i has no off-diagonal
00239 * entries. Thus Pe [i] must be in the range 0 to pfree-1 for non-empty
00240 * rows.
00241 *
00242 * During execution, it is used for both supervariables and elements:
00243 *
00244 * Principal supervariable i: index into Iw of the description of
00245 * supervariable i. A supervariable represents one or more rows of
00246 * the matrix with identical nonzero pattern. In this case,
00247 * Pe [i] >= 0.
00248 *
00249 * Non-principal supervariable i: if i has been absorbed into another
00250 * supervariable j, then Pe [i] = FLIP (j), where FLIP (j) is defined
00251 * as -(j)-2). Row j has the same pattern as row i. Note that j
00252 * might later be absorbed into another supervariable j2, in which
00253 * case Pe [i] is still FLIP (j), and Pe [j] = FLIP (j2) which is
00254 * < EMPTY, where EMPTY is defined as (-1) in amd_internal.h.
00255 *
00256 * Unabsorbed element e: the index into Iw of the description of element
00257 * e, if e has not yet been absorbed by a subsequent element. Element
00258 * e is created when the supervariable of the same name is selected as
00259 * the pivot. In this case, Pe [i] >= 0.
00260 *
00261 * Absorbed element e: if element e is absorbed into element e2, then
00262 * Pe [e] = FLIP (e2). This occurs when the pattern of e (which we
00263 * refer to as Le) is found to be a subset of the pattern of e2 (that
00264 * is, Le2). In this case, Pe [i] < EMPTY. If element e is "null"
00265 * (it has no nonzeros outside its pivot block), then Pe [e] = EMPTY,
00266 * and e is the root of an assembly subtree (or the whole tree if
00267 * there is just one such root).
00268 *
00269 * Dense variable i: if i is "dense", then Pe [i] = EMPTY.
00270 *
00271 * On output, Pe holds the assembly tree/forest, which implicitly
00272 * represents a pivot order with identical fill-in as the actual order
00273 * (via a depth-first search of the tree), as follows. If Nv [i] > 0,
00274 * then i represents a node in the assembly tree, and the parent of i is
00275 * Pe [i], or EMPTY if i is a root. If Nv [i] = 0, then (i, Pe [i])
00276 * represents an edge in a subtree, the root of which is a node in the
00277 * assembly tree. Note that i refers to a row/column in the original
00278 * matrix, not the permuted matrix.
00279 *
00280 * ABIPInfo: A abip_float array of size AMD_INFO. If present, (that is, not ABIP_NULL),
00281 * then statistics about the ordering are returned in the ABIPInfo array.
00282 * See amd.h for a description.
00283
00284 * -----
00285 * INPUT/MODIFIED (undefined on output):
00286 * -----
00287 *
00288 * Len: An integer array of size n. On input, Len [i] holds the number of
00289 * entries in row i of the matrix, excluding the diagonal. The contents
00290 * of Len are undefined on output.
00291 *
00292 * Iw: An integer array of size iwlen. On input, Iw [0..pfree-1] holds the
00293 * description of each row i in the matrix. The matrix must be symmetric,
00294 * and both upper and lower triangular parts must be present. The
00295 * diagonal must not be present. Row i is held as follows:
00296 *
00297 * Len [i]: the length of the row i data structure in the Iw array.
00298 * Iw [Pe [i] ... Pe [i] + Len [i] - 1]:
00299 * the list of column indices for nonzeros in row i (simple
00300 * supervariables), excluding the diagonal. All supervariables
00301 * start with one row/column each (supervariable i is just row i).
00302 * If Len [i] is zero on input, then Pe [i] is ignored on input.
00303 *
00304 * Note that the rows need not be in any particular order, and there
00305 * may be empty space between the rows.
00306 *
00307 * During execution, the supervariable i experiences fill-in. This is
00308 * represented by placing in i a list of the elements that cause fill-in
00309 * in supervariable i:
00310 *
00311 * Len [i]: the length of supervariable i in the Iw array.
00312 * Iw [Pe [i] ... Pe [i] + Elen [i] - 1]:
00313 * the list of elements that contain i. This list is kept short
00314 * by removing absorbed elements.
00315 * Iw [Pe [i] + Elen [i] ... Pe [i] + Len [i] - 1]:
00316 * the list of supervariables in i. This list is kept short by
00317 * removing nonprincipal variables, and any entry j that is also

```

```

00318 *      contained in at least one of the elements (j in Le) in the list
00319 *      for i (e in row i).
00320 *
00321 * When supervariable i is selected as pivot, we create an element e of
00322 * the same name (e=i):
00323 *
00324 *      Len [e]: the length of element e in the Iw array.
00325 *      Iw [Pe [e] ... Pe [e] + Len [e] - 1]:
00326 *      the list of supervariables in element e.
00327 *
00328 * An element represents the fill-in that occurs when supervariable i is
00329 * selected as pivot (which represents the selection of row i and all
00330 * non-principal variables whose principal variable is i). We use the
00331 * term Le to denote the set of all supervariables in element e. Absorbed
00332 * supervariables and elements are pruned from these lists when
00333 * computationally convenient.
00334 *
00335 * CAUTION: THE INPUT MATRIX IS OVERWRITTEN DURING COMPUTATION.
00336 * The contents of Iw are undefined on output.
00337 *
00338 * -----
00339 * OUTPUT (need not be set on input):
00340 * -----
00341 *
00342 * Nv: An integer array of size n. During execution, ABS (Nv [i]) is equal to
00343 * the number of rows that are represented by the principal supervariable
00344 * i. If i is a nonprincipal or dense variable, then Nv [i] = 0.
00345 * Initially, Nv [i] = 1 for all i. Nv [i] < 0 signifies that i is a
00346 * principal variable in the pattern Lme of the current pivot element me.
00347 * After element me is constructed, Nv [i] is set back to a positive
00348 * value.
00349 *
00350 * On output, Nv [i] holds the number of pivots represented by super
00351 * row/column i of the original matrix, or Nv [i] = 0 for non-principal
00352 * rows/columns. Note that i refers to a row/column in the original
00353 * matrix, not the permuted matrix.
00354 *
00355 * Elen: An integer array of size n. See the description of Iw above. At the
00356 * start of execution, Elen [i] is set to zero for all rows i. During
00357 * execution, Elen [i] is the number of elements in the list for
00358 * supervariable i. When e becomes an element, Elen [e] = FLIP (esize) is
00359 * set, where esize is the size of the element (the number of pivots, plus
00360 * the number of nonpivotal entries). Thus Elen [e] < EMPTY.
00361 * Elen (i) = EMPTY set when variable i becomes nonprincipal.
00362 *
00363 * For variables, Elen (i) >= EMPTY holds until just before the
00364 * postordering and permutation vectors are computed. For elements,
00365 * Elen [e] < EMPTY holds.
00366 *
00367 * On output, Elen [i] is the degree of the row/column in the Cholesky
00368 * factorization of the permuted matrix, corresponding to the original row
00369 * i, if i is a super row/column. It is equal to EMPTY if i is
00370 * non-principal. Note that i refers to a row/column in the original
00371 * matrix, not the permuted matrix.
00372 *
00373 * Note that the contents of Elen on output differ from the Fortran
00374 * version (Elen holds the inverse permutation in the Fortran version,
00375 * which is instead returned in the Next array in this C version,
00376 * described below).
00377 *
00378 * Last: In a degree list, Last [i] is the supervariable preceding i, or EMPTY
00379 * if i is the head of the list. In a hash bucket, Last [i] is the hash
00380 * key for i.
00381 *
00382 * Last [Head [hash]] is also used as the head of a hash bucket if
00383 * Head [hash] contains a degree list (see the description of Head,
00384 * below).
00385 *
00386 * On output, Last [0..n-1] holds the permutation. That is, if
00387 * i = Last [k], then row i is the kth pivot row (where k ranges from 0 to
00388 * n-1). Row Last [k] of A is the kth row in the permuted matrix, PAP'.
00389 *
00390 * Next: Next [i] is the supervariable following i in a link list, or EMPTY if
00391 * i is the last in the list. Used for two kinds of lists: degree lists
00392 * and hash buckets (a supervariable can be in only one kind of list at a
00393 * time).
00394 *
00395 * On output Next [0..n-1] holds the inverse permutation. That is, if
00396 * k = Next [i], then row i is the kth pivot row. Row i of A appears as
00397 * the (Next[i])-th row in the permuted matrix, PAP'.
00398 *
00399 * Note that the contents of Next on output differ from the Fortran
00400 * version (Next is undefined on output in the Fortran version).
00401 *
00402 * -----
00403 * LOCAL WORKSPACE (not input or output - used only during execution):
00404 * -----

```

```

00405 *
00406 * Degree: An integer array of size n. If i is a supervariable, then
00407 * Degree [i] holds the current approximation of the external degree of
00408 * row i (an upper bound). The external degree is the number of nonzeros
00409 * in row i, minus ABS (Nv [i]), the diagonal part. The bound is equal to
00410 * the exact external degree if Elen [i] is less than or equal to two.
00411 *
00412 * We also use the term "external degree" for elements e to refer to
00413 * |Le \ Lme|. If e is an element, then Degree [e] is |Le|, which is the
00414 * degree of the off-diagonal part of the element e (not including the
00415 * diagonal part).
00416 *
00417 * Head: An integer array of size n. Head is used for degree lists.
00418 * Head [deg] is the first supervariable in a degree list. All
00419 * supervariables i in a degree list Head [deg] have the same approximate
00420 * degree, namely, deg = Degree [i]. If the list Head [deg] is empty then
00421 * Head [deg] = EMPTY.
00422 *
00423 * During supervariable detection Head [hash] also serves as a pointer to
00424 * a hash bucket. If Head [hash] >= 0, there is a degree list of degree
00425 * hash. The hash bucket head pointer is Last [Head [hash]]. If
00426 * Head [hash] = EMPTY, then the degree list and hash bucket are both
00427 * empty. If Head [hash] < EMPTY, then the degree list is empty, and
00428 * FLIP (Head [hash]) is the head of the hash bucket. After supervariable
00429 * detection is complete, all hash buckets are empty, and the
00430 * (Last [Head [hash]] = EMPTY) condition is restored for the non-empty
00431 * degree lists.
00432 *
00433 * W: An integer array of size n. The flag array W determines the status of
00434 * elements and variables, and the external degree of elements.
00435 *
00436 * for elements:
00437 *   if W [e] = 0, then the element e is absorbed.
00438 *   if W [e] >= wflg, then W [e] - wflg is the size of the set
00439 *   |Le \ Lme|, in terms of nonzeros (the sum of ABS (Nv [i]) for
00440 *   each principal variable i that is both in the pattern of
00441 *   element e and NOT in the pattern of the current pivot element,
00442 *   me).
00443 *   if wflg > W [e] > 0, then e is not absorbed and has not yet been
00444 *   seen in the scan of the element lists in the computation of
00445 *   |Le\Lme| in Scan 1 below.
00446 *
00447 * for variables:
00448 *   during supervariable detection, if W [j] != wflg then j is
00449 *   not in the pattern of variable i.
00450 *
00451 * The W array is initialized by setting W [i] = 1 for all i, and by
00452 * setting wflg = 2. It is reinitialized if wflg becomes too large (to
00453 * ensure that wflg+n does not cause integer overflow).
00454 *
00455 * -----
00456 * LOCAL INTEGERS:
00457 * -----
00458 */
00459
00460     Int deg;
00461     Int degme;
00462     Int dext;
00463     Int lemax;
00464
00465     Int e;
00466     Int elenme;
00467     Int eln;
00468
00469     Int i;
00470     Int ilast;
00471     Int inext;
00472
00473     Int j;
00474     Int jlast;
00475     Int jnext;
00476
00477     Int k;
00478     Int knt1;
00479     Int knt2;
00480     Int knt3;
00481
00482     Int lenj;
00483     Int ln;
00484     Int me;
00485     Int mindeg;
00486     Int nel;
00487     Int nleft;
00488     Int nvi;
00489     Int nvj;
00490     Int npiv;
00491     Int slenme;

```

```

00492
00493         Int wbig;
00494         Int we;
00495         Int wflg;
00496         Int wnvi;
00497
00498         Int ok;
00499         Int ndense;
00500         Int ncmpa;
00501         Int dense;
00502         Int aggressive;
00503
00504         unsigned Int hash ;          /* unsigned, so that hash % n is well defined.*/
00505
00506 /*
00507 * deg:      the degree of a variable or element
00508 * degme:    size, |Lme|, of the current element, me (= Degree [me])
00509 * dext:     external degree, |Le \ Lme|, of some element e
00510 * lemax:    largest |Le| seen so far (called dmax in Fortran version)
00511 * e:        an element
00512 * elenme:   the length, Elen [me], of element list of pivotal variable
00513 * eln:      the length, Elen [...], of an element list
00514 * hash:     the computed value of the hash function
00515 * i:        a supervariable
00516 * ilast:    the entry in a link list preceding i
00517 * inext:    the entry in a link list following i
00518 * j:        a supervariable
00519 * jlast:    the entry in a link list preceding j
00520 * jnext:    the entry in a link list, or path, following j
00521 * k:        the pivot order of an element or variable
00522 * knt1:     loop counter used during element construction
00523 * knt2:     loop counter used during element construction
00524 * knt3:     loop counter used during compression
00525 * lenj:     Len [j]
00526 * ln:       length of a supervariable list
00527 * me:       current supervariable being eliminated, and the current
00528 *           element created by eliminating that supervariable
00529 * mindeg:   current minimum degree
00530 * nel:      number of pivots selected so far
00531 * nleft:    n - nel, the number of nonpivotal rows/columns remaining
00532 * nvi:      the number of variables in a supervariable i (= Nv [i])
00533 * nvj:      the number of variables in a supervariable j (= Nv [j])
00534 * nvpiv:    number of pivots in current element
00535 * slenme:   number of variables in variable list of pivotal variable
00536 * wbig:     = (INT_MAX - n) for the int version, (SuiteSparse_long_max - n)
00537 *           for the SuiteSparse_long version. wflg is not allowed to
00538 *           be >= wbig.
00539 * we:       W [e]
00540 * wflg:     used for flagging the W array. See description of Iw.
00541 * wnvi:     wflg - Nv [i]
00542 * x:        either a supervariable or an element
00543 *
00544 * ok:       true if supervariable j can be absorbed into i
00545 * ndense:   number of "dense" rows/columns
00546 * dense:    rows/columns with initial degree > dense are considered "dense"
00547 * aggressive: true if aggressive absorption is being performed
00548 * ncmpa:    number of garbage collections
00549
00550 * -----
00551 * LOCAL DOUBLES, used for statistical output only (except for alpha):
00552 * -----
00553 */
00554
00555         abip_float f;
00556         abip_float r;
00557         abip_float ndiv;
00558         abip_float s;
00559         abip_float nms_lu;
00560         abip_float nms_ldl;
00561         abip_float dmax;
00562         abip_float alpha;
00563         abip_float lnz;
00564         abip_float lnzme;
00565
00566 /*
00567 * f:        nvpiv
00568 * r:        degme + nvpiv
00569 * ndiv:     number of divisions for LU or LDL' factorizations
00570 * s:        number of multiply-subtract pairs for LU factorization, for the
00571 *           current element me
00572 * nms_lu    number of multiply-subtract pairs for LU factorization
00573 * nms_ldl   number of multiply-subtract pairs for LDL' factorization
00574 * dmax:     the largest number of entries in any column of L, including the
00575 *           diagonal
00576 * alpha:    "dense" degree ratio
00577 * lnz:      the number of nonzeros in L (excluding the diagonal)
00578 * lnzme:    the number of nonzeros in L (excl. the diagonal) for the

```

```

00579 *          current element me
00580
00581 * -----
00582 * LOCAL "POINTERS" (indices into the Iw array)
00583 * -----
00584 */
00585
00586     Int p;
00587     Int p1;
00588     Int p2;
00589     Int p3;
00590     Int p4;
00591     Int pdst;
00592     Int pend;
00593     Int pj;
00594     Int pme;
00595     Int pme1;
00596     Int pme2;
00597     Int pn;
00598     Int psrc;
00599
00600 /*
00601 * Any parameter (Pe [...] or pfree) or local variable starting with "p" (for
00602 * Pointer) is an index into Iw, and all indices into Iw use variables starting
00603 * with "p." The only exception to this rule is the iwlen input argument.
00604 *
00605 * p:          pointer into lots of things
00606 * p1:         Pe [i] for some variable i (start of element list)
00607 * p2:         Pe [i] + Elen [i] - 1 for some variable i
00608 * p3:         index of first supervariable in clean list
00609 * p4:
00610 * pdst:       destination pointer, for compression
00611 * pend:       end of memory to compress
00612 * pj:         pointer into an element or variable
00613 * pme:        pointer into the current element (pme1...pme2)
00614 * pme1:       the current element, me, is stored in Iw [pme1...pme2]
00615 * pme2:       the end of the current element
00616 * pn:         pointer into a "clean" variable, also used to compress
00617 * psrc:       source pointer, for compression
00618 */
00619
00620 /* ===== */
00621 /* INITIALIZATIONS */
00622 /* ===== */
00623
00624     /* Note that this restriction on iwlen is slightly more restrictive than
00625     * what is actually required in AMD_2. AMD_2 can operate with no elbow
00626     * room at all, but it will be slow. For better performance, at least
00627     * size-n elbow room is enforced. */
00628     ASSERT (iwlen >= pfree + n) ;
00629     ASSERT (n > 0) ;
00630
00631     /* initialize output statistics */
00632     lnz = 0 ;
00633     ndiv = 0 ;
00634     nms_lu = 0 ;
00635     nms_ldl = 0 ;
00636     dmax = 1 ;
00637     me = EMPTY ;
00638
00639     mindeg = 0 ;
00640     ncmpa = 0 ;
00641     nel = 0 ;
00642     lemax = 0 ;
00643
00644     /* get control parameters */
00645     if (Control != (abip_float *) ABIP_NULL)
00646     {
00647         alpha = Control [AMD_DENSE] ;
00648         aggressive = (Control [AMD_AGGRESSIVE] != 0) ;
00649     }
00650     else
00651     {
00652         alpha = AMD_DEFAULT_DENSE ;
00653         aggressive = AMD_DEFAULT_AGGRESSIVE ;
00654     }
00655
00656     /* Note: if alpha is NaN, this is undefined: */
00657     if (alpha < 0)
00658     {
00659         /* only remove completely dense rows/columns */
00660         dense = n-2 ;
00661     }
00662     else
00663     {
00664         dense = alpha * sqrt ((abip_float) n) ;
00665     }

```

```

00666
00667     dense = MAX (16, dense) ;
00668     dense = MIN (n, dense) ;
00669     AMD_DEBUG1 (("AMD (debug), alpha %g, aggr. "ID"\n", alpha, aggressive)) ;
00670
00671     for (i = 0 ; i < n ; i++)
00672     {
00673         Last [i] = EMPTY ;
00674         Head [i] = EMPTY ;
00675         Next [i] = EMPTY ;
00676
00677         /* if separate Hhead array is used for hash buckets: Hhead [i] = EMPTY ; */
00678         Nv [i] = 1 ;
00679         W [i] = 1 ;
00680         Elen [i] = 0 ;
00681         Degree [i] = Len [i] ;
00682     }
00683
00684     #ifndef NDEBUG
00685
00686     AMD_DEBUG1 (("====Nel "ID" initial\n", nel)) ;
00687     AMD_dump (n, Pe, Iw, Len, iwlen, pfree, Nv, Next, Last, Head, Elen, Degree, W, -1) ;
00688
00689     #endif
00690
00691     /* initialize wflg */
00692     wbig = Int_MAX - n ;
00693     wflg = clear_flag (0, wbig, W, n) ;
00694
00695     /* ----- */
00696     /* initialize degree lists and eliminate dense and empty rows */
00697     /* ----- */
00698
00699     ndense = 0 ;
00700
00701     for (i = 0 ; i < n ; i++)
00702     {
00703         deg = Degree [i] ;
00704         ASSERT (deg >= 0 && deg < n) ;
00705
00706         if (deg == 0)
00707         {
00708             /* ----- */
00709             * we have a variable that can be eliminated at once because
00710             * there is no off-diagonal non-zero in its row. Note that
00711             * Nv [i] = 1 for an empty variable i. It is treated just
00712             * the same as an eliminated element i.
00713             * ----- */
00714
00715             Elen [i] = FLIP (1) ;
00716             nel++ ;
00717             Pe [i] = EMPTY ;
00718             W [i] = 0 ;
00719         }
00720         else if (deg > dense)
00721         {
00722             /* ----- */
00723             * Dense variables are not treated as elements, but as unordered,
00724             * non-principal variables that have no parent. They do not take
00725             * part in the postorder, since Nv [i] = 0. Note that the Fortran
00726             * version does not have this option.
00727             * ----- */
00728
00729             AMD_DEBUG1 (("Dense node "ID" degree "ID"\n", i, deg)) ;
00730             ndense++ ;
00731             Nv [i] = 0 ; /* do not postorder this node */
00732             Elen [i] = EMPTY ;
00733             nel++ ;
00734             Pe [i] = EMPTY ;
00735         }
00736         else
00737         {
00738             /* ----- */
00739             * place i in the degree list corresponding to its degree
00740             * ----- */
00741
00742             inext = Head [deg] ;
00743             ASSERT (inext >= EMPTY && inext < n) ;
00744             if (inext != EMPTY) Last [inext] = i ;
00745             Next [i] = inext ;
00746             Head [deg] = i ;
00747         }
00748     }
00749
00750     /* ===== */
00751     /* WHILE (selecting pivots) DO */
00752     /* ===== */

```



```

00753
00754     while (nel < n)
00755     {
00756
00757         #ifndef NDEBUG
00758
00759             AMD_DEBUG1 (("n====Nel "ID"\n", nel)) ;
00760         if (AMD_debug >= 2)
00761         {
00762             AMD_dump (n, Pe, Iw, Len, iwlen, pfree, Nv, Next, Last, Head, Elen, Degree,
W, nel) ;
00763         }
00764
00765         #endif
00766
00767         /* =====
00768
00769         /* GET PIVOT OF MINIMUM DEGREE */
00770         /* =====
00771
00772         /* ----- */
00773         /* find next supervariable for elimination */
00774         /* ----- */
00775
00776         ASSERT (mindeg >= 0 && mindeg < n) ;
00777         for (deg = mindeg ; deg < n ; deg++)
00778         {
00779             me = Head [deg] ;
00780             if (me != EMPTY) break ;
00781         }
00782         mindeg = deg ;
00783         ASSERT (me >= 0 && me < n) ;
00784         AMD_DEBUG1 (("=====me: "ID"\n", me)) ;
00785
00786         /* ----- */
00787         /* remove chosen variable from link list */
00788         /* ----- */
00789
00790         inext = Next [me] ;
00791         ASSERT (inext >= EMPTY && inext < n) ;
00792         if (inext != EMPTY) Last [inext] = EMPTY ;
00793         Head [deg] = inext ;
00794
00795         /* ----- */
00796         /* me represents the elimination of pivots nel to nel+Nv[me]-1. */
00797         /* place me itself as the first in this set. */
00798         /* ----- */
00799
00800         elenme = Elen [me] ;
00801         nvpiv = Nv [me] ;
00802         ASSERT (nvpiv > 0) ;
00803         nel += nvpiv ;
00804
00805         /* =====
00806
00807         /* CONSTRUCT NEW ELEMENT */
00808         /* =====
00809
00810         /* ----- */
00811         /* At this point, me is the pivotal supervariable. It will be
00812         /* converted into the current element. Scan list of the pivotal
00813         /* supervariable, me, setting tree pointers and constructing new list
00814         /* of supervariables for the new element, me. p is a pointer to the
00815         /* current position in the old list.
00816         /* ----- */
00817
00818         /* flag the variable "me" as being in Lme by negating Nv [me] */
00819         Nv [me] = -nvpiv ;
00820         degme = 0 ;
00821         ASSERT (Pe [me] >= 0 && Pe [me] < iwlen) ;
00822
00823         if (elenme == 0)
00824         {
00825             /* -----
00826
00827             /* construct the new element in place */
00828             /* ----- */
00829
00830             pme1 = Pe [me] ;
00831             pme2 = pme1 - 1 ;
00832
00833             for (p = pme1 ; p <= pme1 + Len [me] - 1 ; p++)
00834             {
00835                 i = Iw [p] ;
00836                 ASSERT (i >= 0 && i < n && Nv [i] >= 0) ;

```

```

00834             nvi = Nv [i] ;
00835
00836             if (nvi > 0)
00837             {
00838
00839 ----- */
00839             /* i is a principal variable not yet placed in Lme. */
00840             /* store i in new list */
00841             /* -----
00842             */
00843             /* flag i as being in Lme by negating Nv [i] */
00844             degme += nvi ;
00845             Nv [i] = -nvi ;
00846             Iw [pme2] = i ;
00847
00848             /* -----
00849             */
00849             /* remove variable i from degree list. */
00850             /* -----
00851             */
00851             ilast = Last [i] ;
00852             inext = Next [i] ;
00853             ASSERT (ilast >= EMPTY && ilast < n) ;
00854             ASSERT (inext >= EMPTY && inext < n) ;
00855             if (inext != EMPTY) Last [inext] = ilast ;
00856             if (ilast != EMPTY)
00857             {
00858                 Next [ilast] = inext ;
00859             }
00860             else
00861             {
00862                 /* i is at the head of the degree list */
00863                 ASSERT (Degree [i] >= 0 && Degree [i] < n) ;
00864                 Head [Degree [i]] = inext ;
00865             }
00866             }
00867         }
00868     }
00869     else
00870     {
00871
00872             /* ----- */
00873             /* construct the new element in empty space, Iw [pfree ...] */
00874             /* ----- */
00875
00876             p = Pe [me] ;
00877             pme1 = pfree ;
00878             slenme = Len [me] - elenme ;
00879
00880             for (knt1 = 1 ; knt1 <= elenme + 1 ; knt1++)
00881             {
00882
00883                 if (knt1 > elenme)
00884                 {
00885                     /* search the supervariables in me. */
00886                     e = me ;
00887                     pj = p ;
00888                     ln = slenme ;
00889                     AMD_DEBUG2 (("Search sv: "ID" "ID" "ID"\n", me,pj,ln)) ;
00890                 }
00891                 else
00892                 {
00893                     /* search the elements in me. */
00894                     e = Iw [p++] ;
00895                     ASSERT (e >= 0 && e < n) ;
00896                     pj = Pe [e] ;
00897                     ln = Len [e] ;
00898                     AMD_DEBUG2 (("Search element e "ID" in me "ID"\n",
00899 e,me)) ;
00900                     ASSERT (Elen [e] < EMPTY && W [e] > 0 && pj >= 0) ;
00901                 }
00902
00903                 ASSERT (ln >= 0 && (ln == 0 || (pj >= 0 && pj <
00904 iwlen))) ;
00905
00906             /* -----
00907             * search for different supervariables and add them to the
00908             * new list, compressing when necessary. this loop is
00909             * executed once for each element in the list and once for
00910             * all the supervariables in the list.
00911             * ----- */
00912
00913             for (knt2 = 1 ; knt2 <= ln ; knt2++)
00914             {
00915                 i = Iw [pj++] ;
00916                 ASSERT (i >= 0 && i < n && (i == me || Elen [i] >=

```

```

EMPTY));
00915
00916 [i], wflg)) ;
00917
00918
00919
00920
00921
00922 */
00923
00924 */
00925
00926
00927
00928
00929
00930
00931 adjusting pointers
00932 searched in
00933 the
00934
00935
00936
00937
00938
00939 nothing left of supervariable me */
00940
00941
00942
00943 of element e */
00944
00945
00946
00947 */
00948
00949 */
00950
00951
00952
00953
00954
00955 < iwlen) ;
00956
00957
00958
00959
00960
00961
00962 */
00963
00964
00965
00966
00967
00968
00969
00970
00971
00972 object j: "ID\n", j)) ;
00973
00974
00975
00976
00977
00978 /* copy from source to destination */
00979
00980
00981 [psrc++];
00982
00983
00984
nvi = Nv [i] ;
AMD_DEBUG2 (("ID" "ID" "ID" "ID\n", i, Elen [i], Nv
if (nvi > 0)
{
/* -----
/* compress Iw, if necessary */
/* -----
if (pfree >= iwlen)
{
AMD_DEBUG1 (("GARBAGE COLLECTION\n")) ;
/* prepare for compressing Iw by
* and lengths so that the lists being
* the inner and outer loops contain only
* remaining entries. */
Pe [me] = p ;
Len [me] -= knt1 ;
/* check if
if (Len [me] == 0) Pe [me] = EMPTY ;
Pe [e] = pj ;
Len [e] = ln - knt2 ;
/* nothing left
if (Len [e] == 0) Pe [e] = EMPTY ;
ncmpa++ ; /* one more garbage collection
/* store first entry of each object in Pe
/* FLIP the first entry in each object */
for (j = 0 ; j < n ; j++)
{
pn = Pe [j] ;
if (pn >= 0)
{
ASSERT (pn >= 0 && pn
Pe [j] = Iw [pn] ;
Iw [pn] = FLIP (j) ;
}
}
/* psrc/pdst point to source/destination
psrc = 0 ;
pdst = 0 ;
pend = pme1 - 1 ;
while (psrc <= pend)
{
/* search for next FLIP'd entry */
j = FLIP (Iw [psrc++]) ;
if (j >= 0)
{
AMD_DEBUG2 (("Got
Iw [pdst] = Pe [j] ;
Pe [j] = pdst++ ;
lenj = Len [j] ;
for (knt3 = 0 ; knt3
{
Iw [pdst++] = Iw
}
}
}
}

```

```

00985                                     /* move the new partially-constructed
00986 element */
00987                                     pl = pdst ;
00988                                     for (psrc = pme1 ; psrc <= pfree-1 ;
00989 psrc++)
00988                                     {
00989                                     Iw [pdst++] = Iw [psrc] ;
00990                                     }
00991                                     pme1 = pl ;
00992                                     pfree = pdst ;
00993                                     pj = Pe [e] ;
00994                                     p = Pe [me] ;
00995                                     }
00996                                     /* -----
00997                                     */
00998                                     /* i is a principal variable not yet placed in Lme */
00999                                     /* store i in new list */
01000                                     /* -----
01001                                     */
01002                                     /* flag i as being in Lme by negating Nv [i] */
01003                                     degme += nvi ;
01004                                     Nv [i] = -nvi ;
01005                                     Iw [pfree++] = i ;
01006                                     AMD_DEBUG2 (("      s: "ID"      nv "ID"\n", i, Nv
[i]));
01007
01008                                     /* -----
01009                                     */
01009                                     /* remove variable i from degree link list */
01010                                     /* -----
01011                                     */
01012                                     ilast = Last [i] ;
01013                                     inext = Next [i] ;
01014                                     ASSERT (ilast >= EMPTY && ilast < n) ;
01015                                     ASSERT (inext >= EMPTY && inext < n) ;
01016                                     if (inext != EMPTY) Last [inext] = ilast ;
01017                                     if (ilast != EMPTY)
01018                                     {
01019                                         Next [ilast] = inext ;
01020                                     }
01021                                     else
01022                                     {
01023                                         /* i is at the head of the degree list */
01024                                         ASSERT (Degree [i] >= 0 && Degree [i] <
n) ;
01025                                         Head [Degree [i]] = inext ;
01026                                     }
01027                                     }
01028                                     }
01029                                     if (e != me)
01030                                     {
01031                                         /* set tree pointer and flag to indicate element e is
01032                                         * absorbed into new element me (the parent of e is me)
01033                                         */
01034                                         AMD_DEBUG1 ((" Element "ID" => "ID"\n", e, me)) ;
01035                                         Pe [e] = FLIP (me) ;
01036                                         W [e] = 0 ;
01037                                     }
01038                                     }
01039                                     pme2 = pfree - 1 ;
01040                                     }
01041                                     /* -----
01042                                     */
01043                                     /* me has now been converted into an element in Iw [pme1..pme2] */
01044                                     /* -----
01045                                     */
01046                                     /* degme holds the external degree of new element */
01047                                     Degree [me] = degme ;
01048                                     Pe [me] = pme1 ;
01049                                     Len [me] = pme2 - pme1 + 1 ;
01050                                     ASSERT (Pe [me] >= 0 && Pe [me] < iwlen) ;
01051                                     Elen [me] = FLIP (nv piv + degme) ;
01052                                     /* FLIP (Elen (me)) is now the degree of pivot (including
01053                                     * diagonal part). */
01054                                     #ifndef NDEBUG
01055                                     AMD_DEBUG2 (("New element structure: length= "ID"\n", pme2-pme1+1)) ;
01056                                     for (pme = pme1 ; pme <= pme2 ; pme++) AMD_DEBUG3 ((" "ID"", Iw[pme]));
01057                                     AMD_DEBUG3 ((" \n")) ;
01058                                     }

```

```

01063             #endif
01064
01065             /* ----- */
01066             /* make sure that wflg is not too large. */
01067             /* ----- */
01068
01069             /* With the current value of wflg, wflg+n must not cause integer
01070             * overflow */
01071
01072             wflg = clear_flag (wflg, wbig, W, n) ;
01073
01074             /* =====
01075             */
01076             /* COMPUTE (W [e] - wflg) = |Le\Lme| FOR ALL ELEMENTS */
01077             /* =====
01078             */
01079             /* Scan 1: compute the external degrees of previous elements with
01080             * respect to the current element. That is:
01081             * (W [e] - wflg) = |Le \ Lme|
01082             * for each element e that appears in any supervariable in Lme. The
01083             * notation Le refers to the pattern (list of supervariables) of a
01084             * previous element e, where e is not yet absorbed, stored in
01085             * Iw [Pe [e] + 1 ... Pe [e] + Len [e]]. The notation Lme
01086             * refers to the pattern of the current element (stored in
01087             * Iw [pme1..pme2]). If aggressive absorption is enabled, and
01088             * (W [e] - wflg) becomes zero, then the element e will be absorbed
01089             * in Scan 2.
01090             * ----- */
01091
01092             AMD_DEBUG2 (("me: ") ;
01093             for (pme = pme1 ; pme <= pme2 ; pme++)
01094             {
01095                 i = Iw [pme] ;
01096                 ASSERT (i >= 0 && i < n) ;
01097                 eln = Elen [i] ;
01098                 AMD_DEBUG3 (("ID" Elen "ID": \n", i, eln)) ;
01099
01100                 if (eln > 0)
01101                 {
01102                     /* note that Nv [i] has been negated to denote i in Lme: */
01103                     nvi = -Nv [i] ;
01104                     ASSERT (nvi > 0 && Pe [i] >= 0 && Pe [i] < iwlen) ;
01105                     wnvi = wflg - nvi ;
01106                     for (p = Pe [i] ; p <= Pe [i] + eln - 1 ; p++)
01107                     {
01108                         e = Iw [p] ;
01109                         ASSERT (e >= 0 && e < n) ;
01110                         we = W [e] ;
01111                         AMD_DEBUG4 ((" e "ID" we "ID" ", e, we)) ;
01112
01113                         if (we >= wflg)
01114                         {
01115                             /* unabsorbed element e has been seen in this loop */
01116                             AMD_DEBUG4 ((" unabsorbed, first time seen")) ;
01117                             we -= nvi ;
01118                         }
01119                         else if (we != 0)
01120                         {
01121                             /* e is an unabsorbed element */
01122                             /* this is the first we have seen e in all of Scan 1
01123
01124                             AMD_DEBUG4 ((" unabsorbed")) ;
01125                             we = Degree [e] + wnvi ;
01126                         }
01127
01128                         AMD_DEBUG4 ((" \n")) ;
01129
01130                         W [e] = we ;
01131                     }
01132                 }
01133             }
01134
01135             AMD_DEBUG2 ((" \n")) ;
01136
01137             /* =====
01138             */
01139             /* DEGREE UPDATE AND ELEMENT ABSORPTION */
01140             /* =====
01141             */
01142             /* -----
01143             * Scan 2: for each i in Lme, sum up the degree of Lme (which is
01144             * degme), plus the sum of the external degrees of each Le for the
01145             * elements e appearing within i, plus the supervariables in i.
01146             * Place i in hash list.
01147             * ----- */

```

```

01145
01146         for (pme = pme1 ; pme <= pme2 ; pme++)
01147         {
01148             i = Iw [pme] ;
01149             ASSERT (i >= 0 && i < n && Nv [i] < 0 && Elen [i] >= 0) ;
01150             AMD_DEBUG2 (("Updating: i "ID" "ID" "ID"\n", i, Elen[i], Len [i]));
01151             p1 = Pe [i] ;
01152             p2 = p1 + Elen [i] - 1 ;
01153             pn = p1 ;
01154             hash = 0 ;
01155             deg = 0 ;
01156             ASSERT (p1 >= 0 && p1 < iwlen && p2 >= -1 && p2 < iwlen) ;
01157
01158             /* ----- */
01159             /* scan the element list associated with supervariable i */
01160             /* ----- */
01161
01162             /* UMFPACK/MA38-style approximate degree: */
01163             if (aggressive)
01164             {
01165                 for (p = p1 ; p <= p2 ; p++)
01166                 {
01167                     e = Iw [p] ;
01168                     ASSERT (e >= 0 && e < n) ;
01169                     we = W [e] ;
01170                     if (we != 0)
01171                     {
01172                         /* e is an unabsorbed element */
01173                         /* dext = | Le \ Lme | */
01174                         dext = we - wflg ;
01175                         if (dext > 0)
01176                         {
01177                             deg += dext ;
01178                             Iw [pn++] = e ;
01179                             hash += e ;
01180                             AMD_DEBUG4 ((" e: "ID" hash =
01181 "ID"\n",e,hash)) ;
01182                         }
01183                         else
01184                         {
01185                             /* external degree of e is zero, absorb e
01186
01187                             AMD_DEBUG1 ((" Element "ID" =>"ID"
01188 (aggressive)\n", e, me)) ;
01189
01190                             ASSERT (dext == 0) ;
01191                             Pe [e] = FLIP (me) ;
01192                             W [e] = 0 ;
01193                         }
01194                     }
01195                 }
01196             }
01197             else
01198             {
01199                 for (p = p1 ; p <= p2 ; p++)
01200                 {
01201                     e = Iw [p] ;
01202                     ASSERT (e >= 0 && e < n) ;
01203                     we = W [e] ;
01204                     if (we != 0)
01205                     {
01206                         /* e is an unabsorbed element */
01207                         dext = we - wflg ;
01208                         ASSERT (dext >= 0) ;
01209                         deg += dext ;
01210                         Iw [pn++] = e ;
01211                         hash += e ;
01212                         AMD_DEBUG4 ((" e: "ID" hash = "ID"\n",e,hash)) ;
01213                     }
01214                 }
01215             }
01216
01217             /* count the number of elements in i (including me): */
01218             Elen [i] = pn - p1 + 1 ;
01219
01220             /* ----- */
01221             /* scan the supervariables in the list associated with i */
01222             /* ----- */
01223
01224             /* The bulk of the AMD run time is typically spent in this loop,
01225             * particularly if the matrix has many dense rows that are not
01226             * removed prior to ordering. */
01227             p3 = pn ;
01228             p4 = p1 + Len [i] ;
01229             for (p = p2 + 1 ; p < p4 ; p++)
01230             {
01231                 j = Iw [p] ;
01232                 ASSERT (j >= 0 && j < n) ;

```

```

01229         nvj = Nv [j] ;
01230         if (nvj > 0)
01231         {
01232             /* j is unabsorbed, and not in Lme. */
01233             /* add to degree and add to new list */
01234             deg += nvj ;
01235             Iw [pn++] = j ;
01236             hash += j ;
01237             AMD_DEBUG4 ((" s: "ID" hash "ID" Nv[j]= "ID"\n", j,
hash, nvj)) ;
01238         }
01239     }
01240
01241     /* ----- */
01242     /* update the degree and check for mass elimination */
01243     /* ----- */
01244
01245     /* with aggressive absorption, deg==0 is identical to the
01246     * Elen [i] == 1 && p3 == pn test, below. */
01247
01248     ASSERT (IMPLIES (aggressive, (deg==0) == (Elen[i]==1 && p3==pn)))
;
01249
01250     if (Elen [i] == 1 && p3 == pn)
01251     {
01252
01253         /* ----- */
01254         /* mass elimination */
01255         /* ----- */
01256
01257         /* There is nothing left of this node except for an edge to
01258         * the current pivot element. Elen [i] is 1, and there are
01259         * no variables adjacent to node i. Absorb i into the
01260         * current pivot element, me. Note that if there are two or
01261         * more mass eliminations, fillin due to mass elimination is
01262         * possible within the npiv-by-npiv pivot block. It is this
01263         * step that causes AMD's analysis to be an upper bound.
01264         *
01265         * The reason is that the selected pivot has a lower
01266         * approximate degree than the true degree of the two mass
01267         * eliminated nodes. There is no edge between the two mass
01268         * eliminated nodes. They are merged with the current pivot
01269         * anyway.
01270         *
01271         * No fillin occurs in the Schur complement, in any case,
01272         * and this effect does not decrease the quality of the
01273         * ordering itself, just the quality of the nonzero and
01274         * flop count analysis. It also means that the post-ordering
01275         * is not an exact elimination tree post-ordering. */
01276
01277         AMD_DEBUG1 ((" MASS i "ID" => parent e "ID"\n", i, me)) ;
01278         Pe [i] = FLIP (me) ;
01279         nvi = -Nv [i] ;
01280         degme -= nvi ;
01281         npiv += nvi ;
01282         nel += nvi ;
01283         Nv [i] = 0 ;
01284         Elen [i] = EMPTY ;
01285     }
01286     else
01287     {
01288
01289         /* ----- */
01290         /* update the upper-bound degree of i */
01291         /* ----- */
01292
01293         /* the following degree does not yet include the size
01294         * of the current element, which is added later: */
01295
01296         Degree [i] = MIN (Degree [i], deg) ;
01297
01298         /* ----- */
01299         /* add me to the list for i */
01300         /* ----- */
01301
01302         /* move first supervariable to end of list */
01303         Iw [pn] = Iw [p3] ;
01304
01305         /* move first element to end of element part of
list */
01306         Iw [p3] = Iw [p1] ;
01307
01308         /* add new element, me, to front of list. */
01309         Iw [p1] = me ;
01310
01311         /* store the new length of the list in Len [i] */
01312         Len [i] = pn - p1 + 1 ;

```

```

01312
01313
01314
01315
01316
01317
01318
01319
01320
01321
01322
01323
01324
01325
01326
01327
01328
01329
01330
01331
01332
01333
01334
01335
01336
01337
01338
01339
01340
01341
01342
01343
01344
01345
01346
01347
01348
01349
01350
01351
01352
01353
01354
01355
01356
01357
01358
01359
01360
01361
01362
01363
01364
01365
01366
01367
01368
01369
01370
01371
01372
01373
01374
01375
01376
01377
01378
01379
01380
01381
01382
01383
01384
01385
01386
01387
01388
01389
01390
01391
01392
01393
01394
01395
01396

/* ----- */
/* place in hash bucket. Save hash key of i in Last [i]. */
/* ----- */

/* NOTE: this can fail if hash is negative, because the ANSI C
 * standard does not define a % b when a and/or b are negative.
 * That's why hash is defined as an unsigned Int, to avoid this
 * problem. */
hash = hash % n ;
ASSERT (((Int) hash) >= 0 && ((Int) hash) < n) ;

/* if the Hhead array is not used: */
j = Head [hash] ;
if (j <= EMPTY)
{
    /* degree list is empty, hash head is FLIP (j) */
    Next [i] = FLIP (j) ;
    Head [hash] = FLIP (i) ;
}
else
{
    /* degree list is not empty, use Last [Head [hash]] as
     * hash head. */
    Next [i] = Last [j] ;
    Last [j] = i ;
}

/* if a separate Hhead array is used: */
Next [i] = Hhead [hash] ;
Hhead [hash] = i ;
*/

Last [i] = hash ;
}

Degree [me] = degree ;

/* ----- */
/* Clear the counter array, W [...], by incrementing wflg. */
/* ----- */

/* make sure that wflg+n does not cause integer overflow */
lemax = MAX (lemax, degme) ;
wflg += lemax ;
wflg = clear_flag (wflg, wbig, W, n) ;
/* at this point, W [0..n-1] < wflg holds */

/* ===== */
/* SUPERVARIABLE DETECTION */
/* ===== */

AMD_DEBUG1 (("Detecting supervariables:\n")) ;
for (pme = pme1 ; pme <= pme2 ; pme++)
{
    i = Iw [pme] ;
    ASSERT (i >= 0 && i < n) ;
    AMD_DEBUG2 (("Consider i "ID" nv "ID"\n", i, Nv [i])) ;
    if (Nv [i] < 0)
    {
        /* i is a principal variable in Lme */

        /* -----
         * examine all hash buckets with 2 or more variables. We do
         * this by examining all unique hash keys for supervariables in
         * the pattern Lme of the current element, me
         * ----- */

        /* let i = head of hash bucket, and empty the hash bucket */
        ASSERT (Last [i] >= 0 && Last [i] < n) ;
        hash = Last [i] ;

        /* if Hhead array is not used: */
        j = Head [hash] ;
        if (j == EMPTY)
        {
            /* hash bucket and degree list are both empty */
            i = EMPTY ;
        }
        else if (j < EMPTY)
        {
            /* degree list is empty */
            i = FLIP (j) ;
            Head [hash] = EMPTY ;
        }
    }
}

```



```

01397     }
01398     else
01399     {
01400         /* degree list is not empty, restore Last [j] of head j
*/
01401         i = Last [j] ;
01402         Last [j] = EMPTY ;
01403     }
01404
01405     /* if separate Hhead array is used: *
01406     i = Hhead [hash] ;
01407     Hhead [hash] = EMPTY ;
01408     */
01409
01410     ASSERT (i >= EMPTY && i < n) ;
01411     AMD_DEBUG2 (("----i "ID" hash "ID"\n", i, hash)) ;
01412
01413     while (i != EMPTY && Next [i] != EMPTY)
01414     {
01415
01416         /* -----
01417         * this bucket has one or more variables following i.
01418         * scan all of them to see if i can absorb any entries
01419         * that follow i in hash bucket. Scatter i into w.
01420         * -----
*/
01421
01422         ln = Len [i] ;
01423         eln = Elen [i] ;
01424         ASSERT (ln >= 0 && eln >= 0) ;
01425         ASSERT (Pe [i] >= 0 && Pe [i] < iwlen) ;
01426
01427         /* do not flag the first element in
the list (me) */
01428
01429         for (p = Pe [i] + 1 ; p <= Pe [i] + ln - 1 ; p++)
01430         {
01431             ASSERT (Iw [p] >= 0 && Iw [p] < n) ;
01432             W [Iw [p]] = wflg ;
01433         }
01434
01435         /* -----
01436         /* scan every other entry j following i in bucket */
01437         /* -----
*/
01438
01439         jlast = i ;
01440         j = Next [i] ;
01441         ASSERT (j >= EMPTY && j < n) ;
01442
01443         while (j != EMPTY)
01444         {
01445             /* -----
01446             /* check if j and i have identical nonzero pattern */
01447             /* -----
*/
01448
01449             AMD_DEBUG3 (("compare i "ID" and j "ID"\n", i, j)) ;
01450
01451             /* check if i and j have the same Len and Elen */
01452             ASSERT (Len [j] >= 0 && Elen [j] >= 0) ;
01453             ASSERT (Pe [j] >= 0 && Pe [j] < iwlen) ;
01454             ok = (Len [j] == ln) && (Elen [j] == eln) ;
01455
01456             /* skip the first element
in the list (me) */
01457
01458             for (p = Pe [j] + 1 ; ok && p <= Pe [j] + ln - 1 ;
p++)
01459             {
01460                 ASSERT (Iw [p] >= 0 && Iw [p] < n) ;
01461                 if (W [Iw [p]] != wflg) ok = 0 ;
01462             }
01463
01464             if (ok)
01465             {
01466                 /* -----
01467                 /* found it! j can be absorbed into i */
01468                 /* -----
*/
01469
01470                 AMD_DEBUG1 (("found it! j "ID" => i
ID"\n", j, i));
01471
01472                 Pe [j] = FLIP (i) ;
01473
01474                 /* both Nv [i]

```

```

and Nv [j] are negated since they */
01472                                     /* are in Lme, and the absolute values of
each */
01473                                     /* are the number of variables in i and
j: */
01474                                     Nv [i] += Nv [j] ;
01475                                     Nv [j] = 0 ;
01476                                     Elen [j] = EMPTY ;
01477
01478                                     /* delete j
from hash bucket */
01479                                     ASSERT (j != Next [j]) ;
01480                                     j = Next [j] ;
01481                                     Next [jlast] = j ;
01482                                     }
01483                                     else
01484                                     {
01485                                     /* j cannot be absorbed into i */
01486                                     jlast = j ;
01487                                     ASSERT (j != Next [j]) ;
01488                                     j = Next [j] ;
01489                                     }
01490                                     ASSERT (j >= EMPTY && j < n) ;
01491                                     }
01492
01493                                     /* -----
01494                                     * no more variables can be absorbed into i
01495                                     * go to next i in bucket and clear flag array
01496                                     * -----
*/
01497
01498                                     wflg++ ;
01499                                     i = Next [i] ;
01500                                     ASSERT (i >= EMPTY && i < n) ;
01501                                     }
01502                                     }
01503
01504
01505                                     AMD_DEBUG2 (("detect done\n")) ;
01506
01507                                     /* =====
*/
01508                                     /* RESTORE DEGREE LISTS AND REMOVE NONPRINCIPAL SUPERVARIABLES FROM ELEMENT */
01509                                     /* =====
*/
01510
01511                                     p = pme1 ;
01512                                     nleft = n - nel ;
01513                                     for (pme = pme1 ; pme <= pme2 ; pme++)
01514                                     {
01515                                     i = Iw [pme] ;
01516                                     ASSERT (i >= 0 && i < n) ;
01517                                     nvi = -Nv [i] ;
01518                                     AMD_DEBUG3 (("Restore i "ID" "ID"\n", i, nvi)) ;
01519                                     if (nvi > 0)
01520                                     {
01521                                     /* i is a principal variable in Lme */
01522                                     /* restore Nv [i] to signify that i is principal */
01523                                     Nv [i] = nvi ;
01524
01525                                     /* ----- */
01526                                     /* compute the external degree (add size of current element) */
01527                                     /* ----- */
01528
01529                                     deg = Degree [i] + degme - nvi ;
01530                                     deg = MIN (deg, nleft - nvi) ;
01531                                     ASSERT (IMPLIES (aggressive, deg > 0) && deg >= 0 && deg < n) ;
01532
01533                                     /* ----- */
01534                                     /* place the supervariable at the head of the degree list */
01535                                     /* ----- */
01536
01537                                     inext = Head [deg] ;
01538                                     ASSERT (inext >= EMPTY && inext < n) ;
01539                                     if (inext != EMPTY) Last [inext] = i ;
01540                                     Next [i] = inext ;
01541                                     Last [i] = EMPTY ;
01542                                     Head [deg] = i ;
01543
01544                                     /* ----- */
01545                                     /* save the new degree, and find the minimum degree */
01546                                     /* ----- */
01547
01548                                     mindeg = MIN (mindeg, deg) ;
01549                                     Degree [i] = deg ;
01550
01551                                     /* ----- */

```

```

01552                                     /* place the supervariable in the element pattern */
01553                                     /* ----- */
01554
01555                                     Iw [p++] = i ;
01556                                     }
01557     }
01558     AMD_DEBUG2 (("restore done\n")) ;
01559
01560     /* ===== */
01561     /* FINALIZE THE NEW ELEMENT */
01562     /* ===== */
01563
01564     AMD_DEBUG2 (("ME = "ID" DONE\n", me)) ;
01565     Nv [me] = nv piv ;
01566
01567     /* save the length of the list for the new element me */
01568     Len [me] = p - pme1 ;
01569     if (Len [me] == 0)
01570     {
01571         /* there is nothing left of the current pivot element */
01572         /* it is a root of the assembly tree */
01573         Pe [me] = EMPTY ;
01574         W [me] = 0 ;
01575     }
01576
01577     if (elenme != 0)
01578     {
01579         /* element was not constructed in place: deallocate part of */
01580         /* it since newly nonprincipal variables may have been removed */
01581         pfree = p ;
01582     }
01583
01584     /* The new element has nv piv pivots and the size of the contribution
01585     * block for a multifrontal method is degme-by-degme, not including
01586     * the "dense" rows/columns. If the "dense" rows/columns are included,
01587     * the frontal matrix is no larger than
01588     * (degme+ndense)-by-(degme+ndense).
01589     */
01590
01591     if (ABIPInfo != (abip_float *) ABIP_NULL)
01592     {
01593         f = nv piv ;
01594         r = degme + ndense ;
01595         dmax = MAX (dmax, f + r) ;
01596
01597         /* number of nonzeros in L (excluding the diagonal) */
01598         lnzme = f*r + (f-1)*f/2 ;
01599         lnz += lnzme ;
01600
01601         /* number of divide operations for LDL' and for LU */
01602         ndiv += lnzme ;
01603
01604         /* number of multiply-subtract pairs for LU */
01605         s = f*r*r + r*(f-1)*f + (f-1)*f*(2*f-1)/6 ;
01606         nms_lu += s ;
01607
01608         /* number of multiply-subtract pairs for LDL' */
01609         nms_ldl += (s + lnzme)/2 ;
01610     }
01611
01612     #ifndef NDEBUG
01613
01614     AMD_DEBUG2 (("finalize done nel "ID" n "ID"\n      ::::\n", nel, n)) ;
01615     for (pme = Pe [me] ; pme <= Pe [me] + Len [me] - 1 ; pme++)
01616     {
01617         AMD_DEBUG3 ((" "ID", Iw [pme])) ;
01618     }
01619     AMD_DEBUG3 ((" \n")) ;
01620
01621     #endif
01622 }
01623
01624 /* ===== */
01625 /* DONE SELECTING PIVOTS */
01626 /* ===== */
01627
01628 if (ABIPInfo != (abip_float *) ABIP_NULL)
01629 {
01630
01631     /* count the work to factorize the ndense-by-ndense submatrix */
01632     f = ndense ;
01633     dmax = MAX (dmax, (abip_float) ndense) ;
01634
01635     /* number of nonzeros in L (excluding the diagonal) */
01636     lnzme = (f-1)*f/2 ;

```

```

01637         lnz += lnzme ;
01638
01639         /* number of divide operations for LDL' and for LU */
01640         ndiv += lnzme ;
01641
01642         /* number of multiply-subtract pairs for LU */
01643         s = (f-1)*f*(2*f-1)/6 ;
01644         nms_lu += s ;
01645
01646         /* number of multiply-subtract pairs for LDL' */
01647         nms_ldl += (s + lnzme)/2 ;
01648
01649         /* number of nz's in L (excl. diagonal) */
01650         ABIPInfo [AMD_LNZ] = lnz ;
01651
01652         /* number of divide ops for LU and LDL' */
01653         ABIPInfo [AMD_NDIV] = ndiv ;
01654
01655         /* number of multiply-subtract pairs for LDL' */
01656         ABIPInfo [AMD_NMULTSUBS_LDL] = nms_ldl ;
01657
01658         /* number of multiply-subtract pairs for LU */
01659         ABIPInfo [AMD_NMULTSUBS_LU] = nms_lu ;
01660
01661         /* number of "dense" rows/columns */
01662         ABIPInfo [AMD_NDENSE] = ndense ;
01663
01664         /* largest front is dmax-by-dmax */
01665         ABIPInfo [AMD_DMAX] = dmax ;
01666
01667         /* number of garbage collections in AMD */
01668         ABIPInfo [AMD_NCMPA] = ncmpa ;
01669
01670         /* successful ordering */
01671         ABIPInfo [AMD_STATUS] = AMD_OK ;
01672     }
01673
01674     /* ===== */
01675     /* POST-ORDERING */
01676     /* ===== */
01677
01678     /* -----
01679     * Variables at this point:
01680     *
01681     * Pe: holds the elimination tree. The parent of j is FLIP (Pe [j]),
01682     * or EMPTY if j is a root. The tree holds both elements and
01683     * non-principal (unordered) variables absorbed into them.
01684     * Dense variables are non-principal and unordered.
01685     *
01686     * Elen: holds the size of each element, including the diagonal part.
01687     * FLIP (Elen [e]) > 0 if e is an element. For unordered
01688     * variables i, Elen [i] is EMPTY.
01689     *
01690     * Nv: Nv [e] > 0 is the number of pivots represented by the element e.
01691     * For unordered variables i, Nv [i] is zero.
01692     *
01693     * Contents no longer needed:
01694     * W, Iw, Len, Degree, Head, Next, Last.
01695     *
01696     * The matrix itself has been destroyed.
01697     *
01698     * n: the size of the matrix.
01699     * No other scalars needed (pfree, iwlen, etc.)
01700     * ----- */
01701
01702     /* restore Pe */
01703     for (i = 0 ; i < n ; i++)
01704     {
01705         Pe [i] = FLIP (Pe [i]) ;
01706     }
01707
01708     /* restore Elen, for output information, and for postordering */
01709     for (i = 0 ; i < n ; i++)
01710     {
01711         Elen [i] = FLIP (Elen [i]) ;
01712     }
01713
01714     /* Now the parent of j is Pe [j], or EMPTY if j is a root. Elen [e] > 0
01715     * is the size of element e. Elen [i] is EMPTY for unordered variable i. */
01716
01717     #ifndef NDEBUG
01718
01719     AMD_DEBUG2 ((" \nTree:\n")) ;
01720     for (i = 0 ; i < n ; i++)
01721     {
01722         AMD_DEBUG2 ((" "ID" parent: "ID" ", i, Pe [i])) ;
01723         ASSERT (Pe [i] >= EMPTY && Pe [i] < n) ;

```

```

01724         if (Nv [i] > 0)
01725         {
01726             /* this is an element */
01727             e = i ;
01728             AMD_DEBUG2 ((" element, size is "ID"\n", Elen [i])) ;
01729             ASSERT (Elen [e] > 0) ;
01730         }
01731         AMD_DEBUG2 ((" \n")) ;
01732     }
01733     AMD_DEBUG2 ((" \nelements:\n")) ;
01734
01735     for (e = 0 ; e < n ; e++)
01736     {
01737         if (Nv [e] > 0)
01738         {
01739             AMD_DEBUG3 (("Element e= "ID" size "ID" nv "ID" \n", e, Elen [e], Nv [e]))
01740         }
01741     }
01742 }
01743
01744 AMD_DEBUG2 ((" \nvariables:\n")) ;
01745 for (i = 0 ; i < n ; i++)
01746 {
01747     Int cnt ;
01748     if (Nv [i] == 0)
01749     {
01750         AMD_DEBUG3 (("i unordered: "ID"\n", i)) ;
01751         j = Pe [i] ;
01752         cnt = 0 ;
01753         AMD_DEBUG3 ((" j: "ID"\n", j)) ;
01754         if (j == EMPTY)
01755         {
01756             AMD_DEBUG3 ((" i is a dense variable\n")) ;
01757         }
01758         else
01759         {
01760             ASSERT (j >= 0 && j < n) ;
01761             while (Nv [j] == 0)
01762             {
01763                 AMD_DEBUG3 ((" j : "ID"\n", j)) ;
01764                 j = Pe [j] ;
01765                 AMD_DEBUG3 ((" j:: "ID"\n", j)) ;
01766                 cnt++ ;
01767                 if (cnt > n) break ;
01768             }
01769             e = j ;
01770             AMD_DEBUG3 ((" got to e: "ID"\n", e)) ;
01771         }
01772     }
01773 }
01774
01775 #endif
01776
01777 /* ===== */
01778 /* compress the paths of the variables */
01779 /* ===== */
01780
01781 for (i = 0 ; i < n ; i++)
01782 {
01783     if (Nv [i] == 0)
01784     {
01785
01786         /* -----
01787         * i is an un-ordered row. Traverse the tree from i until
01788         * reaching an element, e. The element, e, was the principal
01789         * supervariable of i and all nodes in the path from i to when e
01790         * was selected as pivot.
01791         * ----- */
01792
01793         AMD_DEBUG1 (("Path compression, i unordered: "ID"\n", i)) ;
01794         j = Pe [i] ;
01795         ASSERT (j >= EMPTY && j < n) ;
01796         AMD_DEBUG3 ((" j: "ID"\n", j)) ;
01797
01798         if (j == EMPTY)
01799         {
01800             /* Skip a dense variable. It has no parent. */
01801             AMD_DEBUG3 ((" i is a dense variable\n")) ;
01802             continue ;
01803         }
01804
01805         /* while (j is a variable) */
01806         while (Nv [j] == 0)
01807         {
01808             AMD_DEBUG3 ((" j : "ID"\n", j)) ;
01809             j = Pe [j] ;

```

```

01810             AMD_DEBUG3 (("      j:: "ID"\n", j)) ;
01811             ASSERT (j >= 0 && j < n) ;
01812         }
01813
01814         /* got to an element e */
01815         e = j ;
01816         AMD_DEBUG3 (("got to e: "ID"\n", e)) ;
01817
01818         /* -----
01819         * traverse the path again from i to e, and compress the path
01820         * (all nodes point to e). Path compression allows this code to
01821         * compute in O(n) time.
01822         * ----- */
01823
01824         j = i ;
01825
01826         /* while (j is a variable) */
01827         while (Nv [j] == 0)
01828         {
01829             jnext = Pe [j] ;
01830             AMD_DEBUG3 (("j "ID" jnext "ID"\n", j, jnext)) ;
01831             Pe [j] = e ;
01832             j = jnext ;
01833             ASSERT (j >= 0 && j < n) ;
01834         }
01835     }
01836 }
01837
01838 /* ===== */
01839 /* postorder the assembly tree */
01840 /* ===== */
01841
01842 AMD_postorder (n, Pe, Nv, Elen,
01843               W, /* output order */
01844               Head, Next, Last) ; /* workspace */
01845
01846 /* ===== */
01847 /* compute output permutation and inverse permutation */
01848 /* ===== */
01849
01850 /* W [e] = k means that element e is the kth element in the new
01851 * order. e is in the range 0 to n-1, and k is in the range 0 to
01852 * the number of elements. Use Head for inverse order. */
01853
01854 for (k = 0 ; k < n ; k++)
01855 {
01856     Head [k] = EMPTY ;
01857     Next [k] = EMPTY ;
01858 }
01859
01860 for (e = 0 ; e < n ; e++)
01861 {
01862     k = W [e] ;
01863     ASSERT ((k == EMPTY) == (Nv [e] == 0)) ;
01864     if (k != EMPTY)
01865     {
01866         ASSERT (k >= 0 && k < n) ;
01867         Head [k] = e ;
01868     }
01869 }
01870
01871 /* construct output inverse permutation in Next,
01872 * and permutation in Last */
01873 nel = 0 ;
01874
01875 for (k = 0 ; k < n ; k++)
01876 {
01877     e = Head [k] ;
01878     if (e == EMPTY) break ;
01879     ASSERT (e >= 0 && e < n && Nv [e] > 0) ;
01880     Next [e] = nel ;
01881     nel += Nv [e] ;
01882 }
01883 ASSERT (nel == n - ndense) ;
01884
01885 /* order non-principal variables (dense, & those merged into supervar's) */
01886 for (i = 0 ; i < n ; i++)
01887 {
01888     if (Nv [i] == 0)
01889     {
01890         e = Pe [i] ;
01891         ASSERT (e >= EMPTY && e < n) ;
01892
01893         if (e != EMPTY)
01894         {
01895             /* This is an unordered variable that was merged
01896              * into element e via supernode detection or mass

```

```

01897                                     * elimination of i when e became the pivot element.
01898                                     * Place i in order just before e. */
01899                                     ASSERT (Next [i] == EMPTY && Nv [e] > 0) ;
01900                                     Next [i] = Next [e] ;
01901                                     Next [e]++ ;
01902                                     }
01903                                     else
01904                                     {
01905                                         /* This is a dense unordered variable, with no parent.
01906                                         * Place it last in the output order. */
01907                                         Next [i] = nel++ ;
01908                                     }
01909                                     }
01910                                 }
01911
01912                                 ASSERT (nel == n) ;
01913
01914                                 AMD_DEBUG2 ((" \n\nPerm:\n")) ;
01915
01916                                 for (i = 0 ; i < n ; i++)
01917                                 {
01918                                     k = Next [i] ;
01919                                     ASSERT (k >= 0 && k < n) ;
01920                                     Last [k] = i ;
01921                                     AMD_DEBUG2 (("      perm ["ID"] = "ID"\n", k, i)) ;
01922                                 }
01923 }

```

## 4.7 amd/amd\_aat.c File Reference

```
#include "amd_internal.h"
```

### Functions

- GLOBAL size\_t AMD\_aat (Int n, const Int Ap[], const Int Ai[], Int Len[], Int Tp[], abip\_float ABIPInfo[])

### 4.7.1 Function Documentation

#### 4.7.1.1 AMD\_aat()

```

GLOBAL size_t AMD_aat (
    Int n,
    const Int Ap[],
    const Int Ai[],
    Int Len[],
    Int Tp[],
    abip_float ABIPInfo[] )

```

Definition at line 20 of file [amd\\_aat.c](#).

## 4.8 amd\_aat.c

[Go to the documentation of this file.](#)

```

00001 /* ===== */
00002 /* == AMD_aat == */
00003 /* ===== */
00004
00005 /* ----- */
00006 /* AMD, Copyright (c) Timothy A. Davis, */
00007 /* Patrick R. Amestoy, and Iain S. Duff. See ../README.txt for License. */
00008 /* email: DrTimothyAldenDavis@gmail.com */
00009 /* ----- */
00010
00011 /* AMD_aat: compute the symmetry of the pattern of A, and count the number of
00012 * nonzeros each column of A+A' (excluding the diagonal). Assumes the input
00013 * matrix has no errors, with sorted columns and no duplicates
00014 * (AMD_valid (n, n, Ap, Ai) must be AMD_OK, but this condition is not
00015 * checked).
00016 */
00017
00018 #include "amd_internal.h"
00019
00020 GLOBAL size_t AMD_aat /* returns nz in A+A' */
00021 (
00022     Int n,
00023     const Int Ap [ ],
00024     const Int Ai [ ],
00025     Int Len [ ], /* Len [j]: length of column j of A+A', excl diagonal*/
00026     Int Tp [ ], /* workspace of size n */
00027     abip_float ABIPInfo [ ]
00028 )
00029 {
00030     Int p1;
00031     Int p2;
00032     Int p;
00033     Int pj;
00034     Int pj2;
00035
00036     Int i;
00037     Int j;
00038     Int k;
00039
00040     Int nzdiag;
00041     Int nzboth;
00042     Int nz;
00043
00044     abip_float sym ;
00045     size_t nzaat ;
00046
00047 #ifndef NDEBUG
00048
00049     AMD_debug_init ("AMD AAT") ;
00050     for (k = 0 ; k < n ; k++) Tp [k] = EMPTY ;
00051     ASSERT (AMD_valid (n, n, Ap, Ai) == AMD_OK) ;
00052
00053 #endif
00054
00055     if (ABIPInfo != (abip_float *) ABIP_NULL)
00056     {
00057         /* clear the ABIPInfo array, if it exists */
00058         for (i = 0 ; i < AMD_INFO ; i++)
00059         {
00060             ABIPInfo [i] = EMPTY ;
00061         }
00062         ABIPInfo [AMD_STATUS] = AMD_OK ;
00063     }
00064
00065     for (k = 0 ; k < n ; k++)
00066     {
00067         Len [k] = 0 ;
00068     }
00069
00070     nzdiag = 0 ;
00071     nzboth = 0 ;
00072     nz = Ap [n] ;
00073
00074     for (k = 0 ; k < n ; k++)
00075     {
00076         p1 = Ap [k] ;
00077         p2 = Ap [k+1] ;
00078         AMD_DEBUG2 (("nAAT Column: "ID" p1: "ID" p2: "ID"\n", k, p1, p2)) ;
00079
00080         /* construct A+A' */
00081         for (p = p1 ; p < p2 ; )
00082     
```



```

00083         /* scan the upper triangular part of A */
00084         j = Ai [p] ;
00085
00086         if (j < k)
00087         {
00088             /* entry A (j,k) is in the strictly upper triangular part,
00089             * add both A (j,k) and A (k,j) to the matrix A+A' */
00090             Len [j]++ ;
00091             Len [k]++ ;
00092             AMD_DEBUG3 (("      upper ("ID","ID") ("ID","ID")\n", j,k, k,j));
00093             p++ ;
00094         }
00095         else if (j == k)
00096         {
00097             /* skip the diagonal */
00098             p++ ;
00099             nzdiag++ ;
00100             break ;
00101         }
00102         else /* j > k */
00103         {
00104             /* first entry below the diagonal */
00105             break ;
00106         }
00107
00108         /* scan lower triangular part of A, in column j until reaching
00109         * row k. Start where last scan left off. */
00110         ASSERT (Tp [j] != EMPTY) ;
00111         ASSERT (Ap [j] <= Tp [j] && Tp [j] <= Ap [j+1]) ;
00112         pj2 = Ap [j+1] ;
00113
00114         for (pj = Tp [j] ; pj < pj2 ; )
00115         {
00116             i = Ai [pj] ;
00117
00118             if (i < k)
00119             {
00120                 /* A (i,j) is only in the lower part, not in upper.
00121                 * add both A (i,j) and A (j,i) to the matrix A+A' */
00122                 Len [i]++ ;
00123                 Len [j]++ ;
00124                 AMD_DEBUG3 (("      lower ("ID","ID") ("ID","ID")\n", i,j, j,i)) ;
00125                 pj++ ;
00126             }
00127             else if (i == k)
00128             {
00129                 /* entry A (k,j) in lower part and A (j,k) in upper */
00130                 pj++ ;
00131                 nzboth++ ;
00132                 break ;
00133             }
00134             else /* i > k */
00135             {
00136                 /* consider this entry later, when k advances to i */
00137                 break ;
00138             }
00139             Tp [j] = pj ;
00140         }
00141     }
00142
00143     /* Tp [k] points to the entry just below the diagonal in column k */
00144     Tp [k] = p ;
00145 }
00146
00147 /* clean up, for remaining mismatched entries */
00148 for (j = 0 ; j < n ; j++)
00149 {
00150     for (pj = Tp [j] ; pj < Ap [j+1] ; pj++)
00151     {
00152         i = Ai [pj] ;
00153
00154         /* A (i,j) is only in the lower part, not in upper.
00155         * add both A (i,j) and A (j,i) to the matrix A+A' */
00156         Len [i]++ ;
00157         Len [j]++ ;
00158         AMD_DEBUG3 (("      lower cleanup ("ID","ID") ("ID","ID")\n", i,j, j,i)) ;
00159     }
00160 }
00161
00162 /* ----- */
00163 /* compute the symmetry of the nonzero pattern of A */
00164 /* ----- */
00165
00166 /* Given a matrix A, the symmetry of A is:
00167 *   B = tril (spones (A), -1) + triu (spones (A), 1) ;
00168 *   sym = nnz (B & B') / nnz (B) ;
00169 *   or 1 if nnz (B) is zero. */

```

```

00170
00171     if (nz == nzdiag)
00172     {
00173         sym = 1 ;
00174     }
00175     else
00176     {
00177         sym = (2 * (abip_float) nzboth) / ((abip_float) (nz - nzdiag)) ;
00178     }
00179
00180     nzaat = 0 ;
00181     for (k = 0 ; k < n ; k++)
00182     {
00183         nzaat += Len [k] ;
00184     }
00185
00186     AMD_DEBUG1 (("AMD nz in A+A', excluding diagonal (nzaat) = %g\n", (abip_float) nzaat)) ;
00187     AMD_DEBUG1 (("    nzboth: "ID" nz: "ID" nzdiag: "ID" symmetry: %g\n", nzboth, nz, nzdiag, sym))
;
00188
00189     if (ABIPInfo != (abip_float *) ABIP_NULL)
00190     {
00191         ABIPInfo [AMD_STATUS] = AMD_OK ;
00192         ABIPInfo [AMD_N] = n ;
00193         ABIPInfo [AMD_NZ] = nz ;
00194         ABIPInfo [AMD_SYMMETRY] = sym ;           /* symmetry of pattern of A */
00195         ABIPInfo [AMD_NZDIAG] = nzdiag ;         /* nonzeros on diagonal of A */
00196         ABIPInfo [AMD_NZ_A_PLUS_AT] = nzaat ;     /* nonzeros in A+A' */
00197     }
00198
00199     return (nzaat) ;
00200 }

```

## 4.9 amd/amd\_control.c File Reference

```
#include "amd_internal.h"
```

### Functions

- GLOBAL void AMD\_control (abip\_float Control[])

#### 4.9.1 Function Documentation

##### 4.9.1.1 AMD\_control()

```
GLOBAL void AMD_control (
    abip_float Control[] )
```

Definition at line 18 of file amd\_control.c.

## 4.10 amd\_control.c

[Go to the documentation of this file.](#)

```

00001 /* ===== */
00002 /* == AMD_control == */
00003 /* ===== */
00004
00005 /* ----- */
00006 /* AMD, Copyright (c) Timothy A. Davis, */
00007 /* Patrick R. Amestoy, and Iain S. Duff. See ../README.txt for License. */
00008 /* email: DrTimothyAldenDavis@gmail.com */
00009 /* ----- */
00010
00011 /* User-callable. Prints the control parameters for AMD. See amd.h
00012 * for details. If the Control array is not present, the defaults are
00013 * printed instead.
00014 */
00015
00016 #include "amd_internal.h"
00017
00018 GLOBAL void AMD_control
00019 (
00020     abip_float Control [ ]
00021 )
00022 {
00023     abip_float alpha ;
00024     Int aggressive ;
00025
00026     if (Control != (abip_float *) ABIP_NULL)
00027     {
00028         alpha = Control [AMD_DENSE] ;
00029         aggressive = Control [AMD_AGGRESSIVE] != 0 ;
00030     }
00031     else
00032     {
00033         alpha = AMD_DEFAULT_DENSE ;
00034         aggressive = AMD_DEFAULT_AGGRESSIVE ;
00035     }
00036
00037     PRINTF (("AMD version %d.%d.%d, %s: approximate minimum degree ordering\n"
00038 "dense row parameter: %g\n", AMD_MAIN_VERSION, AMD_SUB_VERSION, AMD_SUBSUB_VERSION,
AMD_DATE, alpha)) ;
00039
00040     if (alpha < 0)
00041     {
00042         PRINTF (("no rows treated as dense\n")) ;
00043     }
00044     else
00045     {
00046         PRINTF ((
00047 " (rows with more than max (%g * sqrt (n), 16) entries are\n"
00048 " considered \"dense\", and placed last in output permutation)\n",
alpha)) ;
00049     }
00050
00051     if (aggressive)
00052     {
00053         PRINTF (("aggressive absorption: yes\n")) ;
00054     }
00055     else
00056     {
00057         PRINTF (("aggressive absorption: no\n")) ;
00058     }
00059
00060     PRINTF (("size of AMD integer: %d\n", sizeof (Int))) ;
00061 }

```

## 4.11 amd/amd\_defaults.c File Reference

```
#include "amd_internal.h"
```

### Functions

- GLOBAL void `AMD_defaults` (`abip_float` Control[])

## 4.11.1 Function Documentation

### 4.11.1.1 AMD\_defaults()

```
GLOBAL void AMD_defaults (
    abip_float Control[] )
```

Definition at line 21 of file [amd\\_defaults.c](#).

## 4.12 amd\_defaults.c

[Go to the documentation of this file.](#)

```
00001 /* ===== */
00002 /* === AMD_defaults ===== */
00003 /* ===== */
00004
00005 /* ----- */
00006 /* AMD, Copyright (c) Timothy A. Davis, */
00007 /* Patrick R. Amestoy, and Iain S. Duff. See ../README.txt for License. */
00008 /* email: DrTimothyAldenDavis@gmail.com */
00009 /* ----- */
00010
00011 /* User-callable. Sets default control parameters for AMD. See amd.h
00012  * for details.
00013  */
00014
00015 #include "amd_internal.h"
00016
00017 /* ===== */
00018 /* === AMD defaults ===== */
00019 /* ===== */
00020
00021 GLOBAL void AMD_defaults
00022 (
00023     abip_float Control [ ]
00024 )
00025 {
00026     Int i;
00027
00028     if (Control != (abip_float *) ABIP_NULL)
00029     {
00030         for (i = 0 ; i < AMD_CONTROL ; i++)
00031         {
00032             Control [i] = 0 ;
00033         }
00034         Control [AMD_DENSE] = AMD_DEFAULT_DENSE ;
00035         Control [AMD_AGGRESSIVE] = AMD_DEFAULT_AGGRESSIVE ;
00036     }
00037 }
```

## 4.13 amd/amd\_dump.c File Reference

```
#include "amd_internal.h"
```

### Functions

- GLOBAL void [AMD\\_debug\\_init](#) (char \*s)
- GLOBAL void [AMD\\_dump](#) (Int n, Int Pe[], Int lw[], Int Len[], Int iwlen, Int pfree, Int Nv[], Int Next[], Int Last[], Int Head[], Int Elen[], Int Degree[], Int W[], Int nel)

## Variables

- `GLOBAL Int AMD_debug = -999`

### 4.13.1 Function Documentation

#### 4.13.1.1 AMD\_debug\_init()

```
GLOBAL void AMD_debug_init (  
    char * s )
```

Definition at line 29 of file `amd_dump.c`.

#### 4.13.1.2 AMD\_dump()

```
GLOBAL void AMD_dump (  
    Int n,  
    Int Pe[],  
    Int Iw[],  
    Int Len[],  
    Int iwlen,  
    Int pfree,  
    Int Nv[],  
    Int Next[],  
    Int Last[],  
    Int Head[],  
    Int Elen[],  
    Int Degree[],  
    Int W[],  
    Int nel )
```

Definition at line 58 of file `amd_dump.c`.

### 4.13.2 Variable Documentation

#### 4.13.2.1 AMD\_debug

```
GLOBAL Int AMD_debug = -999
```

Definition at line 21 of file `amd_dump.c`.

## 4.14 amd\_dump.c

[Go to the documentation of this file.](#)

```

00001 /* ===== */
00002 /* == AMD_dump == */
00003 /* ===== */
00004
00005 /* ----- */
00006 /* AMD, Copyright (c) Timothy A. Davis, */
00007 /* Patrick R. Amestoy, and Iain S. Duff. See ../README.txt for License. */
00008 /* email: DrTimothyAldenDavis@gmail.com */
00009 /* ----- */
00010
00011 /* Debugging routines for AMD. Not used if NDEBUG is not defined at compile-
00012 * time (the default). See comments in amd_internal.h on how to enable
00013 * debugging. Not user-callable.
00014 */
00015
00016 #include "amd_internal.h"
00017
00018 #ifndef NDEBUG
00019
00020 /* This global variable is present only when debugging */
00021 GLOBAL Int AMD_debug = -999 ; /* default is no debug printing */
00022
00023 /* ===== */
00024 /* == AMD_debug_init == */
00025 /* ===== */
00026
00027 /* Sets the debug print level, by reading the file debug.amd (if it exists) */
00028
00029 GLOBAL void AMD_debug_init ( char *s )
00030 {
00031     FILE *f ;
00032     f = fopen ("debug.amd", "r") ;
00033
00034     if (f == (FILE *) ABIP_NULL)
00035     {
00036         AMD_debug = -999 ;
00037     }
00038     else
00039     {
00040         fscanf (f, ID, &AMD_debug) ;
00041         fclose (f) ;
00042     }
00043
00044     if (AMD_debug >= 0)
00045     {
00046         printf ("%s: AMD_debug_init, D= "ID"\n", s, AMD_debug) ;
00047     }
00048 }
00049
00050 /* ===== */
00051 /* == AMD_dump == */
00052 /* ===== */
00053
00054 /* Dump AMD's data structure, except for the hash buckets. This routine
00055 * cannot be called when the hash buckets are non-empty.
00056 */
00057
00058 GLOBAL void AMD_dump (
00059     Int n, /* A is n-by-n */
00060     Int Pe [ ], /* pe [0..n-1]: index in iw of start of row i */
00061     Int Iw [ ], /* workspace of size iwlen, iwlen [0..pfree-1]
00062                * holds the matrix on input */
00063     Int Len [ ], /* len [0..n-1]: length for row i */
00064     Int iwlen, /* length of iw */
00065     Int pfree, /* iw [pfree ... iwlen-1] is empty on input */
00066     Int Nv [ ], /* nv [0..n-1] */
00067     Int Next [ ], /* next [0..n-1] */
00068     Int Last [ ], /* last [0..n-1] */
00069     Int Head [ ], /* head [0..n-1] */
00070     Int Elen [ ], /* size n */
00071     Int Degree [ ], /* size n */
00072     Int W [ ], /* size n */
00073     Int nel
00074 )
00075 {
00076     Int i;
00077     Int pe;
00078     Int elen;
00079     Int nv;
00080     Int len;
00081     Int e;
00082     Int p;

```

```

00083     Int k;
00084     Int j;
00085     Int deg;
00086     Int w;
00087     Int cnt;
00088     Int ilast;
00089
00090     if (AMD_debug < 0) return ;
00091     ASSERT (pfree <= iwlen) ;
00092     AMD_DEBUG3 (("AMD dump, pfree: "ID"\n", pfree)) ;
00093     for (i = 0 ; i < n ; i++)
00094     {
00095         pe = Pe [i] ;
00096         elen = Elen [i] ;
00097         nv = Nv [i] ;
00098         len = Len [i] ;
00099         w = W [i] ;
00100
00101         if (elen >= EMPTY)
00102         {
00103             if (nv == 0)
00104             {
00105                 AMD_DEBUG3 (("nI "ID": nonprincipal: ", i)) ;
00106                 ASSERT (elen == EMPTY) ;
00107
00108                 if (pe == EMPTY)
00109                 {
00110                     AMD_DEBUG3 (("dense node\n")) ;
00111                     ASSERT (w == 1) ;
00112                 }
00113                 else
00114                 {
00115                     ASSERT (pe < EMPTY) ;
00116                     AMD_DEBUG3 (("i "ID" -> parent "ID"\n", i, FLIP (Pe[i])));
00117                 }
00118             }
00119             else
00120             {
00121                 AMD_DEBUG3 (("nI "ID": active principal supervariable:\n", i));
00122                 AMD_DEBUG3 (("nv(i): "ID" Flag: %d\n", nv, (nv < 0))) ;
00123
00124                 ASSERT (elen >= 0) ;
00125                 ASSERT (nv > 0 && pe >= 0) ;
00126                 p = pe ;
00127                 AMD_DEBUG3 (("e/s: ")) ;
00128
00129                 if (elen == 0) AMD_DEBUG3 ((" : ")) ;
00130                 ASSERT (pe + len <= pfree) ;
00131
00132                 for (k = 0 ; k < len ; k++)
00133                 {
00134                     j = Iw [p] ;
00135                     AMD_DEBUG3 ((" "ID"", j)) ;
00136                     ASSERT (j >= 0 && j < n) ;
00137
00138                     if (k == elen-1) AMD_DEBUG3 ((" : ")) ;
00139                     p++ ;
00140                 }
00141
00142                 AMD_DEBUG3 (("n")) ;
00143             }
00144         }
00145         else
00146         {
00147             e = i ;
00148
00149             if (w == 0)
00150             {
00151                 AMD_DEBUG3 (("nE "ID": absorbed element: w "ID"\n", e, w)) ;
00152                 ASSERT (nv > 0 && pe < 0) ;
00153                 AMD_DEBUG3 (("e "ID" -> parent "ID"\n", e, FLIP (Pe [e])));
00154             }
00155             else
00156             {
00157                 AMD_DEBUG3 (("nE "ID": unabsorbed element: w "ID"\n", e, w)) ;
00158                 ASSERT (nv > 0 && pe >= 0) ;
00159                 p = pe ;
00160                 AMD_DEBUG3 ((" : ")) ;
00161                 ASSERT (pe + len <= pfree) ;
00162
00163                 for (k = 0 ; k < len ; k++)
00164                 {
00165                     j = Iw [p] ;
00166                     AMD_DEBUG3 ((" "ID"", j)) ;
00167                     ASSERT (j >= 0 && j < n) ;
00168                     p++ ;
00169                 }

```

```

00170
00171         AMD_DEBUG3 ((" \n")) ;
00172     }
00173 }
00174 }
00175
00176 /* this routine cannot be called when the hash buckets are non-empty */
00177 AMD_DEBUG3 ((" \nDegree lists: \n")) ;
00178 if (nel >= 0)
00179 {
00180     cnt = 0 ;
00181
00182     for (deg = 0 ; deg < n ; deg++)
00183     {
00184         if (Head [deg] == EMPTY) continue ;
00185         ilast = EMPTY ;
00186         AMD_DEBUG3 (("ID": \n", deg)) ;
00187
00188         for (i = Head [deg] ; i != EMPTY ; i = Next [i])
00189         {
00190             AMD_DEBUG3 (("      ID" : next "ID" last "ID" deg "ID" \n", i, Next [i], Last [i], Degree
[i])) ;
00191             ASSERT (i >= 0 && i < n && ilast == Last [i] && deg == Degree [i]) ;
00192             cnt += Nv [i] ;
00193             ilast = i ;
00194         }
00195
00196         AMD_DEBUG3 ((" \n")) ;
00197     }
00198
00199     ASSERT (cnt == n - nel) ;
00200 }
00201 }
00202
00203 #endif

```

## 4.15 amd/amd\_global.c File Reference

```

#include <stdlib.h>
#include "glbopts.h"

```

### Macros

- #define [ABIP\\_NULL](#) 0

### Variables

- void (\*)([amd\\_malloc](#))(size\_t) = malloc
- void (\*)([amd\\_free](#))(void \*) = free
- void (\*)([amd\\_realloc](#))(void \*, size\_t) = realloc
- void (\*)([amd\\_calloc](#))(size\_t, size\_t) = calloc
- int (\*)([amd\\_printf](#))(const char \*,...) = [ABIP\\_NULL](#)

### 4.15.1 Macro Definition Documentation

#### 4.15.1.1 ABIP\_NULL

```
#define ABIP_NULL 0
```

Definition at line 20 of file [amd\\_global.c](#).



## 4.15.2 Variable Documentation

### 4.15.2.1 amd\_calloc

```
void *(* amd_calloc) (size_t, size_t) (  
    size_t ,  
    size_t ) = calloc
```

Definition at line 54 of file [amd\\_global.c](#).

### 4.15.2.2 amd\_free

```
void(* amd_free) (void *) (  
    void * ) = free
```

Definition at line 52 of file [amd\\_global.c](#).

### 4.15.2.3 amd\_malloc

```
void *(* amd_malloc) (size_t) (  
    size_t ) = malloc
```

Definition at line 51 of file [amd\\_global.c](#).

### 4.15.2.4 amd\_printf

```
int(* amd_printf) (const char *,...) (  
    const char * ,  
    ... ) = ABIP_NULL
```

Definition at line 75 of file [amd\\_global.c](#).

### 4.15.2.5 amd\_realloc

```
void *(* amd_realloc) (void *, size_t) (  
    void * ,  
    size_t ) = realloc
```

Definition at line 53 of file [amd\\_global.c](#).

## 4.16 amd\_global.c

[Go to the documentation of this file.](#)

```

00001 /* ===== */
00002 /* == amd_global ===== */
00003 /* ===== */
00004
00005 /* ----- */
00006 /* AMD, Copyright (c) Timothy A. Davis, */
00007 /* Patrick R. Amestoy, and Iain S. Duff. See ../README.txt for License. */
00008 /* email: DrTimothyAldenDavis@gmail.com */
00009 /* ----- */
00010
00011 #include <stdlib.h>
00012 #include "glbopts.h"
00013
00014 #ifdef MATLAB_MEX_FILE
00015 #include "mex.h"
00016 #include "matrix.h"
00017 #endif
00018
00019 #ifndef ABIP_NULL
00020 #define ABIP_NULL 0
00021 #endif
00022
00023 /* ===== */
00024 /* == Default AMD memory manager ===== */
00025 /* ===== */
00026
00027 /* The user can redefine these global pointers at run-time to change the memory
00028 * manager used by AMD. AMD only uses malloc and free; realloc and calloc are
00029 * included for completeness, in case another package wants to use the same
00030 * memory manager as AMD.
00031 *
00032 * If compiling as a MATLAB mexFunction, the default memory manager is mxMalloc.
00033 * You can also compile AMD as a standard ANSI-C library and link a mexFunction
00034 * against it, and then redefine these pointers at run-time, in your
00035 * mexFunction.
00036 *
00037 * If -DNMALLOC is defined at compile-time, no memory manager is specified at
00038 * compile-time. You must then define these functions at run-time, before
00039 * calling AMD, for AMD to work properly.
00040 */
00041
00042 #ifndef NMALLOC
00043 #ifdef MATLAB_MEX_FILE
00044 /* MATLAB mexFunction: */
00045 void *(*amd_malloc) (size_t) = mxMalloc ;
00046 void (*amd_free) (void *) = mxFree ;
00047 void *(*amd_realloc) (void *, size_t) = mxRealloc ;
00048 void *(*amd_calloc) (size_t, size_t) = mxCalloc ;
00049 #else
00050 /* standard ANSI-C: */
00051 void *(*amd_malloc) (size_t) = malloc ;
00052 void (*amd_free) (void *) = free ;
00053 void *(*amd_realloc) (void *, size_t) = realloc ;
00054 void *(*amd_calloc) (size_t, size_t) = calloc ;
00055 #endif
00056 #else
00057 /* no memory manager defined at compile-time; you MUST define one at run-time */
00058 void *(*amd_malloc) (size_t) = ABIP_NULL ;
00059 void (*amd_free) (void *) = ABIP_NULL ;
00060 void *(*amd_realloc) (void *, size_t) = ABIP_NULL ;
00061 void *(*amd_calloc) (size_t, size_t) = ABIP_NULL ;
00062 #endif
00063
00064 /* ===== */
00065 /* == Default AMD printf routine ===== */
00066 /* ===== */
00067
00068 /* The user can redefine this global pointer at run-time to change the printf
00069 * routine used by AMD. If ABIP_NULL, no printing occurs.
00070 *
00071 * If -DNPRINT is defined at compile-time, stdio.h is not included. Printing
00072 * can then be enabled at run-time by setting amd_printf to a non-ABIP_NULL function.
00073 */
00074
00075 int (*amd_printf) (const char *, ...) = ABIP_NULL ;

```

## 4.17 amd/amd\_info.c File Reference

```
#include "amd_internal.h"
```

### Macros

- `#define PRI(format, x) { if (x >= 0) { PRINTF ((format, x)) ; }}`

### Functions

- `GLOBAL void AMD_info (abip_float ABIInfo[])`

### 4.17.1 Macro Definition Documentation

#### 4.17.1.1 PRI

```
#define PRI(  
    format,  
    x ) { if (x >= 0) { PRINTF ((format, x)) ; }}
```

Definition at line 17 of file [amd\\_info.c](#).

### 4.17.2 Function Documentation

#### 4.17.2.1 AMD\_info()

```
GLOBAL void AMD_info (  
    abip_float ABIInfo[ ] )
```

Definition at line 19 of file [amd\\_info.c](#).

## 4.18 amd\_info.c

[Go to the documentation of this file.](#)

```

00001 /* ===== */
00002 /* == AMD_info == */
00003 /* ===== */
00004
00005 /* ----- */
00006 /* AMD, Copyright (c) Timothy A. Davis, */
00007 /* Patrick R. Amestoy, and Iain S. Duff. See ../README.txt for License. */
00008 /* email: DrTimothyAldenDavis@gmail.com */
00009 /* ----- */
00010
00011 /* User-callable. Prints the output statistics for AMD. See amd.h
00012  * for details. If the ABIPInfo array is not present, nothing is printed.
00013  */
00014
00015 #include "amd_internal.h"
00016
00017 #define PRI(format,x) { if (x >= 0) { PRINTF ((format, x)) ; }}
00018
00019 GLOBAL void AMD_info
00020 (
00021     abip_float ABIPInfo [ ]
00022 )
00023 {
00024     abip_float n;
00025     abip_float ndiv;
00026     abip_float nmultsubs_ldl;
00027     abip_float nmultsubs_lu;
00028     abip_float lnz;
00029     abip_float lnzd;
00030
00031     PRINTF (("AMD version %d.%d.%d, %s, results:\n", AMD_MAIN_VERSION, AMD_SUB_VERSION,
AMD_SUBSUB_VERSION, AMD_DATE)) ;
00032
00033     if (!ABIPInfo)
00034     {
00035         return ;
00036     }
00037
00038     n = ABIPInfo [AMD_N] ;
00039     ndiv = ABIPInfo [AMD_NDIV] ;
00040     nmultsubs_ldl = ABIPInfo [AMD_NMULTSUBS_LDL] ;
00041     nmultsubs_lu = ABIPInfo [AMD_NMULTSUBS_LU] ;
00042     lnz = ABIPInfo [AMD_LNZ] ;
00043     lnzd = (n >= 0 && lnz >= 0) ? (n + lnz) : (-1) ;
00044
00045     /* AMD return status */
00046     PRINTF ((" status: ")) ;
00047     if (ABIPInfo [AMD_STATUS] == AMD_OK)
00048     {
00049         PRINTF (("OK\n")) ;
00050     }
00051     else if (ABIPInfo [AMD_STATUS] == AMD_OUT_OF_MEMORY)
00052     {
00053         PRINTF (("out of memory\n")) ;
00054     }
00055     else if (ABIPInfo [AMD_STATUS] == AMD_INVALID)
00056     {
00057         PRINTF (("invalid matrix\n")) ;
00058     }
00059     else if (ABIPInfo [AMD_STATUS] == AMD_OK_BUT_JUMBLED)
00060     {
00061         PRINTF (("OK, but jumbled\n")) ;
00062     }
00063     else
00064     {
00065         PRINTF (("unknown\n")) ;
00066     }
00067
00068     /* statistics about the input matrix */
00069     PRI (" n, dimension of A: %20g\n", n);
00070     PRI (" nz, number of nonzeros in A: %20g\n", ABIPInfo [AMD_NZ]);
00071
00072     ;
00073     PRI (" symmetry of A: %4f\n", ABIPInfo
[AMD_SYMMETRY]) ;
00074     PRI (" number of nonzeros on diagonal: %20g\n", ABIPInfo
[AMD_NZDIAG]) ;
00075     PRI (" nonzeros in pattern of A+A' (excl. diagonal): %20g\n", ABIPInfo
[AMD_NZ_A_PLUS_AT]) ;
00076     PRI (" # dense rows/columns of A+A': %20g\n", ABIPInfo
[AMD_NDENSE]) ;
00077
00078     /* statistics about AMD's behavior */

```

```

00077         PRI ("      memory used, in bytes:                                %.20g\n", ABIPInfo
[AMD_MEMORY]) ;
00078         PRI ("      # of memory compactions:                                %.20g\n", ABIPInfo
[AMD_NCMPA]) ;
00079
00080         /* statistics about the ordering quality */
00081         PRINTF ((" \n"
00082 "      The following approximate statistics are for a subsequent\n"
00083 "      factorization of A(P,P) + A(P,P)'. They are slight upper\n"
00084 "      bounds if there are no dense rows/columns in A+A', and become\n"
00085 "      looser if dense rows/columns exist.\n\n")) ;
00086
00087         PRI ("      nonzeros in L (excluding diagonal):                        %.20g\n", lnz) ;
00088         PRI ("      nonzeros in L (including diagonal):                        %.20g\n", lnzd) ;
00089         PRI ("      # divide operations for LDL' or LU:                          %.20g\n", ndiv) ;
00090         PRI ("      # multiply-subtract operations for LDL':                      %.20g\n", nmultsubs_ldl) ;
00091         PRI ("      # multiply-subtract operations for LU:                        %.20g\n", nmultsubs_lu) ;
00092         PRI ("      max nz. in any column of L (incl. diagonal):                %.20g\n", ABIPInfo
[AMD_DMAX]) ;
00093
00094         /* total flop counts for various factorizations */
00095
00096         if (n >= 0 && ndiv >= 0 && nmultsubs_ldl >= 0 && nmultsubs_lu >= 0)
00097         {
00098             PRINTF ((" \n"
00099 "      chol flop count for real A, sqrt counted as 1 flop: %.20g\n"
00100 "      LDL' flop count for real A:                          %.20g\n"
00101 "      LDL' flop count for complex A:                      %.20g\n"
00102 "      LU flop count for real A (with no pivoting):         %.20g\n"
00103 "      LU flop count for complex A (with no pivoting):      %.20g\n\n",
00104 n + ndiv + 2*nmultsubs_ldl,
00105 ndiv + 2*nmultsubs_ldl,
00106 9*ndiv + 8*nmultsubs_ldl,
00107 ndiv + 2*nmultsubs_lu,
00108 9*ndiv + 8*nmultsubs_lu)) ;
00109         }
00110 }

```

## 4.19 amd/amd\_internal.h File Reference

```

#include <stdlib.h>
#include <stdio.h>
#include <limits.h>
#include <math.h>
#include "amd.h"

```

### Macros

- #define **EMPTY** (-1)
- #define **FLIP**(i) (-(i)-2)
- #define **UNFLIP**(i) ((i < **EMPTY**) ? **FLIP** (i) : (i))
- #define **MAX**(a, b) (((a) > (b)) ? (a) : (b))
- #define **MIN**(a, b) (((a) < (b)) ? (a) : (b))
- #define **IMPLIES**(p, q) (!(p) || (q))
- #define **TRUE** (1)
- #define **FALSE** (0)
- #define **PRIVATE** static
- #define **GLOBAL**
- #define **EMPTY** (-1)
- #define **ABIP\_NULL** 0
- #define **SIZE\_T\_MAX** ((size\_t) (-1))
- #define **Int** int
- #define **ID** "%d"
- #define **Int\_MAX** INT\_MAX

- `#define AMD_order amd_order`
- `#define AMD_defaults amd_defaults`
- `#define AMD_control amd_control`
- `#define AMD_info amd_info`
- `#define AMD_1 amd_1`
- `#define AMD_2 amd_2`
- `#define AMD_valid amd_valid`
- `#define AMD_aat amd_aat`
- `#define AMD_postorder amd_postorder`
- `#define AMD_post_tree amd_post_tree`
- `#define AMD_dump amd_dump`
- `#define AMD_debug amd_debug`
- `#define AMD_debug_init amd_debug_init`
- `#define AMD_preprocess amd_preprocess`
- `#define PRINTF(params) { if (amd_printf != ABIP_NULL) (void) amd_printf params ; }`
- `#define ASSERT(expression)`
- `#define AMD_DEBUG0(params)`
- `#define AMD_DEBUG1(params)`
- `#define AMD_DEBUG2(params)`
- `#define AMD_DEBUG3(params)`
- `#define AMD_DEBUG4(params)`

## Functions

- `GLOBAL size_t AMD_aat (Int n, const Int Ap[], const Int Ai[], Int Len[], Int Tp[], abip_float ABIPInfo[])`
- `GLOBAL void AMD_1 (Int n, const Int Ap[], const Int Ai[], Int P[], Int Pinv[], Int Len[], Int slen, Int S[], abip_float Control[], abip_float ABIPInfo[])`
- `GLOBAL void AMD_postorder (Int nn, Int Parent[], Int Npiv[], Int Fsize[], Int Order[], Int Child[], Int Sibling[], Int Stack[])`
- `GLOBAL Int AMD_post_tree (Int root, Int k, Int Child[], const Int Sibling[], Int Order[], Int Stack[])`
- `GLOBAL void AMD_preprocess (Int n, const Int Ap[], const Int Ai[], Int Rp[], Int Ri[], Int W[], Int Flag[])`

### 4.19.1 Macro Definition Documentation

#### 4.19.1.1 ABIP\_NULL

```
#define ABIP_NULL 0
```

Definition at line 138 of file [amd\\_internal.h](#).

#### 4.19.1.2 AMD\_1

```
#define AMD_1 amd_1
```

Definition at line 187 of file [amd\\_internal.h](#).

#### 4.19.1.3 AMD\_2

```
#define AMD_2 amd_2
```

Definition at line 188 of file [amd\\_internal.h](#).

#### 4.19.1.4 AMD\_aat

```
#define AMD_aat amd_aat
```

Definition at line 190 of file [amd\\_internal.h](#).

#### 4.19.1.5 AMD\_control

```
#define AMD_control amd_control
```

Definition at line 185 of file [amd\\_internal.h](#).

#### 4.19.1.6 AMD\_debug

```
#define AMD_debug amd_debug
```

Definition at line 194 of file [amd\\_internal.h](#).

#### 4.19.1.7 AMD\_DEBUG0

```
#define AMD_DEBUG0(  
    params )
```

Definition at line 328 of file [amd\\_internal.h](#).

#### 4.19.1.8 AMD\_DEBUG1

```
#define AMD_DEBUG1(  
    params )
```

Definition at line 329 of file [amd\\_internal.h](#).

#### 4.19.1.9 AMD\_DEBUG2

```
#define AMD_DEBUG2(  
    params )
```

Definition at line 330 of file [amd\\_internal.h](#).

#### 4.19.1.10 AMD\_DEBUG3

```
#define AMD_DEBUG3(  
    params )
```

Definition at line 331 of file [amd\\_internal.h](#).

#### 4.19.1.11 AMD\_DEBUG4

```
#define AMD_DEBUG4(  
    params )
```

Definition at line 332 of file [amd\\_internal.h](#).

#### 4.19.1.12 AMD\_debug\_init

```
#define AMD_debug_init amd_debug_init
```

Definition at line 195 of file [amd\\_internal.h](#).

#### 4.19.1.13 AMD\_defaults

```
#define AMD_defaults amd\_defaults
```

Definition at line 184 of file [amd\\_internal.h](#).

#### 4.19.1.14 AMD\_dump

```
#define AMD_dump amd_dump
```

Definition at line 193 of file [amd\\_internal.h](#).



#### 4.19.1.15 AMD\_info

```
#define AMD_info amd_info
```

Definition at line 186 of file [amd\\_internal.h](#).

#### 4.19.1.16 AMD\_order

```
#define AMD_order amd_order
```

Definition at line 183 of file [amd\\_internal.h](#).

#### 4.19.1.17 AMD\_post\_tree

```
#define AMD_post_tree amd_post_tree
```

Definition at line 192 of file [amd\\_internal.h](#).

#### 4.19.1.18 AMD\_postorder

```
#define AMD_postorder amd_postorder
```

Definition at line 191 of file [amd\\_internal.h](#).

#### 4.19.1.19 AMD\_preprocess

```
#define AMD_preprocess amd_preprocess
```

Definition at line 196 of file [amd\\_internal.h](#).

#### 4.19.1.20 AMD\_valid

```
#define AMD_valid amd_valid
```

Definition at line 189 of file [amd\\_internal.h](#).

#### 4.19.1.21 ASSERT

```
#define ASSERT(  
    expression )
```

Definition at line 327 of file [amd\\_internal.h](#).

#### 4.19.1.22 EMPTY [1/2]

```
#define EMPTY (-1)
```

Definition at line 129 of file [amd\\_internal.h](#).

#### 4.19.1.23 EMPTY [2/2]

```
#define EMPTY (-1)
```

Definition at line 129 of file [amd\\_internal.h](#).

#### 4.19.1.24 FALSE

```
#define FALSE (0)
```

Definition at line 126 of file [amd\\_internal.h](#).

#### 4.19.1.25 FLIP

```
#define FLIP(  
    i )  (-(i)-2)
```

Definition at line 106 of file [amd\\_internal.h](#).

#### 4.19.1.26 GLOBAL

```
#define GLOBAL
```

Definition at line 128 of file [amd\\_internal.h](#).

#### 4.19.1.27 ID

```
#define ID "%d"
```

Definition at line 180 of file [amd\\_internal.h](#).

#### 4.19.1.28 IMPLIES

```
#define IMPLIES(  
    p,  
    q )  ( ! (p) || (q) )
```

Definition at line 114 of file [amd\\_internal.h](#).

#### 4.19.1.29 Int

```
#define Int int
```

Definition at line 179 of file [amd\\_internal.h](#).

#### 4.19.1.30 Int\_MAX

```
#define Int_MAX INT_MAX
```

Definition at line 181 of file [amd\\_internal.h](#).

#### 4.19.1.31 MAX

```
#define MAX(  
    a,  
    b )  ( (a) > (b) ) ?  (a) :  (b) )
```

Definition at line 110 of file [amd\\_internal.h](#).

#### 4.19.1.32 MIN

```
#define MIN(  
    a,  
    b )  ( (a) < (b) ) ?  (a) :  (b) )
```

Definition at line 111 of file [amd\\_internal.h](#).

#### 4.19.1.33 PRINTF

```
#define PRINTF(  
    params ) { if (amd_printf != ABIP_NULL) (void) amd_printf params ; }
```

Definition at line 205 of file [amd\\_internal.h](#).

#### 4.19.1.34 PRIVATE

```
#define PRIVATE static
```

Definition at line 127 of file [amd\\_internal.h](#).

#### 4.19.1.35 SIZE\_T\_MAX

```
#define SIZE_T_MAX ((size_t) (-1))
```

Definition at line 146 of file [amd\\_internal.h](#).

#### 4.19.1.36 TRUE

```
#define TRUE (1)
```

Definition at line 125 of file [amd\\_internal.h](#).

#### 4.19.1.37 UNFLIP

```
#define UNFLIP(  
    i ) ((i < EMPTY) ? FLIP (i) : (i))
```

Definition at line 107 of file [amd\\_internal.h](#).

### 4.19.2 Function Documentation

#### 4.19.2.1 AMD\_1()

```
GLOBAL void AMD_1 (
    Int n,
    const Int Ap[],
    const Int Ai[],
    Int P[],
    Int Pinv[],
    Int Len[],
    Int slen,
    Int S[],
    abip_float Control[],
    abip_float ABIPInfo[] )
```

Definition at line 29 of file [amd\\_1.c](#).

#### 4.19.2.2 AMD\_aat()

```
GLOBAL size_t AMD_aat (
    Int n,
    const Int Ap[],
    const Int Ai[],
    Int Len[],
    Int Tp[],
    abip_float ABIPInfo[] )
```

Definition at line 20 of file [amd\\_aat.c](#).

#### 4.19.2.3 AMD\_post\_tree()

```
GLOBAL Int AMD_post_tree (
    Int root,
    Int k,
    Int Child[],
    const Int Sibling[],
    Int Order[],
    Int Stack[] )
```

#### 4.19.2.4 AMD\_postorder()

```
GLOBAL void AMD_postorder (
    Int nn,
    Int Parent[],
    Int Npiv[],
    Int Fsize[],
    Int Order[],
    Int Child[],
    Int Sibling[],
    Int Stack[] )
```

Definition at line 15 of file [amd\\_postorder.c](#).

#### 4.19.2.5 AMD\_preprocess()

```
GLOBAL void AMD_preprocess (
    Int n,
    const Int Ap[],
    const Int Ai[],
    Int Rp[],
    Int Ri[],
    Int W[],
    Int Flag[] )
```

Definition at line 29 of file [amd\\_preprocess.c](#).

## 4.20 amd\_internal.h

[Go to the documentation of this file.](#)

```
00001 /* ===== */
00002 /* == amd_internal.h ===== */
00003 /* ===== */
00004
00005 /* ----- */
00006 /* AMD, Copyright (c) Timothy A. Davis, */
00007 /* Patrick R. Amestoy, and Iain S. Duff. See ../README.txt for License. */
00008 /* email: DrTimothyAldenDavis@gmail.com */
00009 /* ----- */
00010
00011 /* This file is for internal use in AMD itself, and does not normally need to
00012 * be included in user code (it is included in UMFPACK, however). All others
00013 * should use amd.h instead.
00014 */
00015
00016 /* ===== */
00017 /* == NDEBUG ===== */
00018 /* ===== */
00019
00020 /*
00021 * Turning on debugging takes some work (see below). If you do not edit this
00022 * file, then debugging is always turned off, regardless of whether or not
00023 * -DNDEBUG is specified in your compiler options.
00024 *
00025 * If AMD is being compiled as a mexFunction, then MATLAB_MEX_FILE is defined,
00026 * and mxAssert is used instead of assert. If debugging is not enabled, no
00027 * MATLAB include files or functions are used. Thus, the AMD library libamd.a
00028 * can be safely used in either a stand-alone C program or in another
00029 * mexFunction, without any change.
00030 */
00031
00032 /*
00033 AMD will be exceedingly slow when running in debug mode. The next three
00034 lines ensure that debugging is turned off.
00035 */
00036 #ifndef NDEBUG
00037 #define NDEBUG
00038 #endif
00039
00040 /*
00041 To enable debugging, uncomment the following line:
00042 #undef NDEBUG
00043 */
00044
00045 /* ----- */
00046 /* ANSI include files */
00047 /* ----- */
00048
00049 /* from stdlib.h: size_t, malloc, free, realloc, and calloc */
00050 #include <stdlib.h>
00051
00052 #if !defined(NPRINT) || !defined(NDEBUG)
00053 /* from stdio.h: printf. Not included if NPRINT is defined at compile time.
00054 * fopen and fscanf are used when debugging. */
00055 #include <stdio.h>
00056 #endif
00057
00058 /* from limits.h: INT_MAX and LONG_MAX */
00059 #include <limits.h>
```

```

00060
00061 /* from math.h: sqrt */
00062 #include <math.h>
00063
00064 /* ----- */
00065 /* MATLAB include files (only if being used in or via MATLAB) */
00066 /* ----- */
00067
00068 #ifdef MATLAB_MEX_FILE
00069 #include "matrix.h"
00070 #include "mex.h"
00071 #endif
00072
00073 /* ----- */
00074 /* basic definitions */
00075 /* ----- */
00076
00077 #ifdef FLIP
00078 #undef FLIP
00079 #endif
00080
00081 #ifdef MAX
00082 #undef MAX
00083 #endif
00084
00085 #ifdef MIN
00086 #undef MIN
00087 #endif
00088
00089 #ifdef EMPTY
00090 #undef EMPTY
00091 #endif
00092
00093 #ifdef GLOBAL
00094 #undef GLOBAL
00095 #endif
00096
00097 #ifdef PRIVATE
00098 #undef PRIVATE
00099 #endif
00100
00101 /* FLIP is a "negation about -1", and is used to mark an integer i that is
00102  * normally non-negative. FLIP (EMPTY) is EMPTY. FLIP of a number > EMPTY
00103  * is negative, and FLIP of a number < EMPTY is positive. FLIP (FLIP (i)) = i
00104  * for all integers i. UNFLIP (i) is >= EMPTY. */
00105 #define EMPTY (-1)
00106 #define FLIP(i) (-(i)-2)
00107 #define UNFLIP(i) ((i < EMPTY) ? FLIP (i) : (i))
00108
00109 /* for integer MAX/MIN, or for doubles when we don't care how NaN's behave: */
00110 #define MAX(a,b) ((a) > (b)) ? (a) : (b)
00111 #define MIN(a,b) ((a) < (b)) ? (a) : (b)
00112
00113 /* logical expression of p implies q: */
00114 #define IMPLIES(p,q) (!(p) || (q))
00115
00116 /* Note that the IBM RS 6000 xlc predefines TRUE and FALSE in <types.h>. */
00117 /* The Compaq Alpha also predefines TRUE and FALSE. */
00118 #ifdef TRUE
00119 #undef TRUE
00120 #endif
00121 #ifdef FALSE
00122 #undef FALSE
00123 #endif
00124
00125 #define TRUE (1)
00126 #define FALSE (0)
00127 #define PRIVATE static
00128 #define GLOBAL
00129 #define EMPTY (-1)
00130
00131 /* Note that Linux's gcc 2.96 defines NULL as ((void *) 0), but other */
00132 /* compilers (even gcc 2.95.2 on Solaris) define NULL as 0 or (0). We */
00133 /* need to use the ANSI standard value of 0. */
00134 #ifdef ABIP_NULL
00135 #undef ABIP_NULL
00136 #endif
00137
00138 #define ABIP_NULL 0
00139
00140 /* largest value of size_t */
00141 #ifndef SIZE_T_MAX
00142 #ifdef SIZE_MAX
00143 /* C99 only */
00144 #define SIZE_T_MAX SIZE_MAX
00145 #else
00146 #define SIZE_T_MAX ((size_t) (-1))

```

```

00147 #endif
00148 #endif
00149
00150 /* ----- */
00151 /* integer type for AMD: int or SuiteSparse_long */
00152 /* ----- */
00153
00154 #include "amd.h"
00155
00156 #if defined (DLONG) || defined (ZLONG)
00157
00158 #define Int SuiteSparse_long
00159 #define ID SuiteSparse_long_id
00160 #define Int_MAX SuiteSparse_long_max
00161
00162 #define AMD_order amd_l_order
00163 #define AMD_defaults amd_l_defaults
00164 #define AMD_control amd_l_control
00165 #define AMD_info amd_l_info
00166 #define AMD_1 amd_l1
00167 #define AMD_2 amd_l2
00168 #define AMD_valid amd_l_valid
00169 #define AMD_aat amd_l_aat
00170 #define AMD_postorder amd_l_postorder
00171 #define AMD_post_tree amd_l_post_tree
00172 #define AMD_dump amd_l_dump
00173 #define AMD_debug amd_l_debug
00174 #define AMD_debug_init amd_l_debug_init
00175 #define AMD_preprocess amd_l_preprocess
00176
00177 #else
00178
00179 #define Int int
00180 #define ID "%d"
00181 #define Int_MAX INT_MAX
00182
00183 #define AMD_order amd_order
00184 #define AMD_defaults amd_defaults
00185 #define AMD_control amd_control
00186 #define AMD_info amd_info
00187 #define AMD_1 amd_1
00188 #define AMD_2 amd_2
00189 #define AMD_valid amd_valid
00190 #define AMD_aat amd_aat
00191 #define AMD_postorder amd_postorder
00192 #define AMD_post_tree amd_post_tree
00193 #define AMD_dump amd_dump
00194 #define AMD_debug amd_debug
00195 #define AMD_debug_init amd_debug_init
00196 #define AMD_preprocess amd_preprocess
00197
00198 #endif
00199
00200 /* ===== */
00201 /* === PRINTF macro ===== */
00202 /* ===== */
00203
00204 /* All output goes through the PRINTF macro. */
00205 #define PRINTF(params) { if (amd_printf != ABIP_NULL) (void) amd_printf params ; }
00206
00207 /* ----- */
00208 /* AMD routine definitions (not user-callable) */
00209 /* ----- */
00210
00211 GLOBAL size_t AMD_aat
00212 (
00213     Int n,
00214     const Int Ap [ ],
00215     const Int Ai [ ],
00216     Int Len [ ],
00217     Int Tp [ ],
00218     abip_float ABIPInfo [ ]
00219 ) ;
00220
00221 GLOBAL void AMD_l
00222 (
00223     Int n,
00224     const Int Ap [ ],
00225     const Int Ai [ ],
00226     Int P [ ],
00227     Int Pinv [ ],
00228     Int Len [ ],
00229     Int slen,
00230     Int S [ ],
00231     abip_float Control [ ],
00232     abip_float ABIPInfo [ ]
00233 ) ;

```



```

00234
00235 GLOBAL void AMD_postorder
00236 (
00237     Int nn,
00238     Int Parent [ ],
00239     Int Npiv [ ],
00240     Int Fsize [ ],
00241     Int Order [ ],
00242     Int Child [ ],
00243     Int Sibling [ ],
00244     Int Stack [ ]
00245 ) ;
00246
00247 GLOBAL Int AMD_post_tree
00248 (
00249     Int root,
00250     Int k,
00251     Int Child [ ],
00252     const Int Sibling [ ],
00253     Int Order [ ],
00254     Int Stack [ ]
00255
00256     #ifndef NDEBUG
00257     , Int nn
00258     #endif
00259 ) ;
00260
00261 GLOBAL void AMD_preprocess
00262 (
00263     Int n,
00264     const Int Ap [ ],
00265     const Int Ai [ ],
00266     Int Rp [ ],
00267     Int Ri [ ],
00268     Int W [ ],
00269     Int Flag [ ]
00270 ) ;
00271
00272 /* ----- */
00273 /* debugging definitions */
00274 /* ----- */
00275
00276 #ifndef NDEBUG
00277
00278 /* from assert.h:  assert macro */
00279 #include <assert.h>
00280
00281 #ifndef EXTERN
00282 #define EXTERN extern
00283 #endif
00284
00285 EXTERN Int AMD_debug ;
00286
00287 GLOBAL void AMD_debug_init ( char *s ) ;
00288
00289 GLOBAL void AMD_dump
00290 (
00291     Int n,
00292     Int Pe [ ],
00293     Int Iw [ ],
00294     Int Len [ ],
00295     Int iwlen,
00296     Int pfree,
00297     Int Nv [ ],
00298     Int Next [ ],
00299     Int Last [ ],
00300     Int Head [ ],
00301     Int Elen [ ],
00302     Int Degree [ ],
00303     Int W [ ],
00304     Int nel
00305 ) ;
00306
00307 #ifdef ASSERT
00308 #undef ASSERT
00309 #endif
00310
00311 /* Use mxAssert if AMD is compiled into a mexFunction */
00312 #ifdef MATLAB_MEX_FILE
00313 #define ASSERT(expression) (mxAssert ((expression), ""))
00314 #else
00315 #define ASSERT(expression) (assert (expression))
00316 #endif
00317
00318 #define AMD_DEBUG0(params) { PRINTF (params) ; }
00319 #define AMD_DEBUG1(params) { if (AMD_debug >= 1) PRINTF (params) ; }
00320 #define AMD_DEBUG2(params) { if (AMD_debug >= 2) PRINTF (params) ; }

```

```

00321 #define AMD_DEBUG3(params) { if (AMD_debug >= 3) PRINTF (params) ; }
00322 #define AMD_DEBUG4(params) { if (AMD_debug >= 4) PRINTF (params) ; }
00323
00324 #else
00325
00326 /* no debugging */
00327 #define ASSERT(expression)
00328 #define AMD_DEBUG0(params)
00329 #define AMD_DEBUG1(params)
00330 #define AMD_DEBUG2(params)
00331 #define AMD_DEBUG3(params)
00332 #define AMD_DEBUG4(params)
00333
00334 #endif

```

## 4.21 amd/amd\_order.c File Reference

```
#include "amd_internal.h"
```

### Functions

- [GLOBAL Int AMD\\_order \(Int n, const Int Ap\[\], const Int Ai\[\], Int P\[\], abip\\_float Control\[\], abip\\_float ABIPInfo\[\]\)](#)

### 4.21.1 Function Documentation

#### 4.21.1.1 AMD\_order()

```

GLOBAL Int AMD_order (
    Int n,
    const Int Ap[],
    const Int Ai[],
    Int P[],
    abip_float Control[],
    abip_float ABIPInfo[] )

```

Definition at line 21 of file [amd\\_order.c](#).

## 4.22 amd\_order.c

[Go to the documentation of this file.](#)

```

00001 /* ===== */
00002 /* == AMD_order ===== */
00003 /* ===== */
00004
00005 /* ----- */
00006 /* AMD, Copyright (c) Timothy A. Davis, */
00007 /* Patrick R. Amestoy, and Iain S. Duff. See ../README.txt for License. */
00008 /* email: DrTimothyAldenDavis@gmail.com */
00009 /* ----- */
00010
00011 /* User-callable AMD minimum degree ordering routine. See amd.h for
00012 * documentation.
00013 */

```

```

00014
00015 #include "amd_internal.h"
00016
00017 /* ===== */
00018 /* === AMD_order ===== */
00019 /* ===== */
00020
00021 GLOBAL Int AMD_order
00022 (
00023     Int n,
00024     const Int Ap [ ],
00025     const Int Ai [ ],
00026     Int P [ ],
00027     abip_float Control [ ],
00028     abip_float ABIPInfo [ ]
00029 )
00030 {
00031     Int *Len;
00032     Int *S;
00033     Int nz;
00034     Int i;
00035     Int *Pinv;
00036     Int info;
00037     Int status;
00038
00039     Int *Rp;
00040     Int *Ri;
00041     Int *Cp;
00042     Int *Ci;
00043     Int ok;
00044
00045     size_t nzaat;
00046     size_t slen;
00047
00048     abip_float mem = 0 ;
00049
00050     #ifndef NDEBUG
00051     AMD_debug_init ("amd") ;
00052     #endif
00053
00054     /* clear the ABIPInfo array, if it exists */
00055     info = ABIPInfo != (abip_float *) ABIP_NULL ;
00056
00057     if (info)
00058     {
00059         for (i = 0 ; i < AMD_INFO ; i++)
00060         {
00061             ABIPInfo [i] = EMPTY ;
00062         }
00063         ABIPInfo [AMD_N] = n ;
00064         ABIPInfo [AMD_STATUS] = AMD_OK ;
00065     }
00066
00067     /* make sure inputs exist and n is >= 0 */
00068     if (Ai == (Int *) ABIP_NULL || Ap == (Int *) ABIP_NULL || P == (Int *) ABIP_NULL || n < 0)
00069     {
00070         if (info) ABIPInfo [AMD_STATUS] = AMD_INVALID ;
00071         return (AMD_INVALID) ; /* arguments are invalid */
00072     }
00073
00074     if (n == 0)
00075     {
00076         return (AMD_OK) ; /* n is 0 so there's nothing to do */
00077     }
00078
00079     nz = Ap [n] ;
00080
00081     if (info)
00082     {
00083         ABIPInfo [AMD_NZ] = nz ;
00084     }
00085
00086     if (nz < 0)
00087     {
00088         if (info) ABIPInfo [AMD_STATUS] = AMD_INVALID ;
00089         return (AMD_INVALID) ;
00090     }
00091
00092     /* check if n or nz will cause size_t overflow */
00093     if (((size_t) n) >= SIZE_T_MAX / sizeof (Int) || ((size_t) nz) >= SIZE_T_MAX / sizeof
(Int))
00094     {
00095         if (info) ABIPInfo [AMD_STATUS] = AMD_OUT_OF_MEMORY ;
00096         return (AMD_OUT_OF_MEMORY) ; /* problem too large */
00097     }
00098
00099     /* check the input matrix: AMD_OK, AMD_INVALID, or AMD_OK_BUT_JUMBLED */

```

```

00100     status = AMD_valid (n, n, Ap, Ai) ;
00101
00102     if (status == AMD_INVALID)
00103     {
00104         if (info) ABIPInfo [AMD_STATUS] = AMD_INVALID ;
00105         return (AMD_INVALID) ;          /* matrix is invalid */
00106     }
00107
00108     /* allocate two size-n integer workspaces */
00109     Len = amd_malloc (n * sizeof (Int)) ;
00110     Pinv = amd_malloc (n * sizeof (Int)) ;
00111     mem += n ;
00112     mem += n ;
00113
00114     if (!Len || !Pinv)
00115     {
00116         /* :: out of memory :: */
00117         amd_free (Len) ;
00118         amd_free (Pinv) ;
00119         if (info) ABIPInfo [AMD_STATUS] = AMD_OUT_OF_MEMORY ;
00120         return (AMD_OUT_OF_MEMORY) ;
00121     }
00122
00123     if (status == AMD_OK_BUT_JUMBLED)
00124     {
00125         /* sort the input matrix and remove duplicate entries */
00126         AMD_DEBUG1 (("Matrix is jumbled\n")) ;
00127         Rp = amd_malloc ((n+1) * sizeof (Int));
00128         Ri = amd_malloc (nz * sizeof (Int));
00129         mem += (n+1) ;
00130         mem += MAX (nz,1) ;
00131
00132         if (!Rp || !Ri)
00133         {
00134             /* :: out of memory :: */
00135             amd_free (Rp) ;
00136             amd_free (Ri) ;
00137             amd_free (Len) ;
00138             amd_free (Pinv) ;
00139
00140             if (info) ABIPInfo [AMD_STATUS] = AMD_OUT_OF_MEMORY ;
00141             return (AMD_OUT_OF_MEMORY) ;
00142         }
00143
00144         /* use Len and Pinv as workspace to create R = A' */
00145         AMD_preprocess (n, Ap, Ai, Rp, Ri, Len, Pinv) ;
00146         Cp = Rp ;
00147         Ci = Ri ;
00148     }
00149     else
00150     {
00151         /* order the input matrix as-is. No need to compute R = A' first */
00152         Rp = ABIP_NULL ;
00153         Ri = ABIP_NULL ;
00154         Cp = (Int *) Ap ;
00155         Ci = (Int *) Ai ;
00156     }
00157
00158     /* ----- */
00159     /* determine the symmetry and count off-diagonal nonzeros in A+A' */
00160     /* ----- */
00161
00162     nzaat = AMD_aat (n, Cp, Ci, Len, P, ABIPInfo) ;
00163     AMD_DEBUG1 (("nzaat: %g\n", (abip_float) nzaat)) ;
00164     ASSERT ((MAX (nz-n, 0) <= nzaat) && (nzaat <= 2 * (size_t) nz)) ;
00165
00166     /* ----- */
00167     /* allocate workspace for matrix, elbow room, and 6 size-n vectors */
00168     /* ----- */
00169
00170     S = ABIP_NULL ;
00171     slen = nzaat ;          /* space for matrix */
00172     ok = ((slen + nzaat/5) >= slen) ; /* check for size_t overflow */
00173     slen += nzaat/5 ;       /* add elbow room */
00174
00175     for (i = 0 ; ok && i < 7 ; i++)
00176     {
00177         ok = ((slen + n) > slen) ; /* check for size_t overflow */
00178         slen += n ;               /* size-n elbow room, 6 size-n work */
00179     }
00180
00181     mem += slen ;
00182     ok = ok && (slen < SIZE_T_MAX / sizeof (Int)) ; /* check for overflow */
00183     ok = ok && (slen < Int_MAX) ;                  /* S[i] for Int i must be
OK */
00184
00185     if (ok)

```

```

00186         {
00187             S = amd_malloc (slen * sizeof (Int)) ;
00188         }
00189
00190         AMD_DEBUG1 (("slen %g\n", (abip_float) slen)) ;
00191
00192         if (!S)
00193         {
00194             /* :: out of memory :: (or problem too large) */
00195             amd_free (Rp) ;
00196             amd_free (Ri) ;
00197             amd_free (Len) ;
00198             amd_free (Pinv) ;
00199             if (info) ABIPInfo [AMD_STATUS] = AMD_OUT_OF_MEMORY ;
00200             return (AMD_OUT_OF_MEMORY) ;
00201         }
00202
00203         if (info)
00204         {
00205             /* memory usage, in bytes. */
00206             ABIPInfo [AMD_MEMORY] = mem * sizeof (Int) ;
00207         }
00208
00209         /* ----- */
00210         /* order the matrix */
00211         /* ----- */
00212
00213         AMD_1 (n, Cp, Ci, P, Pinv, Len, slen, S, Control, ABIPInfo) ;
00214
00215         /* ----- */
00216         /* free the workspace */
00217         /* ----- */
00218
00219         amd_free (Rp) ;
00220         amd_free (Ri) ;
00221         amd_free (Len) ;
00222         amd_free (Pinv) ;
00223         amd_free (S) ;
00224
00225         if (info) ABIPInfo [AMD_STATUS] = status ;
00226         return (status) ; /* successful ordering */
00227     }

```

## 4.23 amd/amd\_post\_tree.c File Reference

```
#include "amd_internal.h"
```

### Functions

- **GLOBAL** `Int AMD_post_tree (Int root, Int k, Int Child[], const Int Sibling[], Int Order[], Int Stack[], Int nn)`

#### 4.23.1 Function Documentation

##### 4.23.1.1 AMD\_post\_tree()

```

GLOBAL Int AMD_post_tree (
    Int root,
    Int k,
    Int Child[],
    const Int Sibling[],
    Int Order[],
    Int Stack[],
    Int nn )

```

Definition at line 15 of file [amd\\_post\\_tree.c](#).

## 4.24 amd\_post\_tree.c

[Go to the documentation of this file.](#)

```

00001 /* ===== */
00002 /* === AMD_post_tree === */
00003 /* ===== */
00004
00005 /* ----- */
00006 /* AMD, Copyright (c) Timothy A. Davis, */
00007 /* Patrick R. Amestoy, and Iain S. Duff. See ../README.txt for License. */
00008 /* email: DrTimothyAldenDavis@gmail.com */
00009 /* ----- */
00010
00011 /* Post-ordering of a supernodal elimination tree. */
00012
00013 #include "amd_internal.h"
00014
00015 GLOBAL Int AMD_post_tree
00016 (
00017     Int root,          /* root of the tree */
00018     Int k,             /* start numbering at k */
00019     Int Child [ ],     /* input argument of size nn, undefined on
00020                          * output. Child [i] is the head of a link
00021                          * list of all nodes that are children of node
00022                          * i in the tree. */
00023     const Int Sibling [ ], /* input argument of size nn, not modified.
00024                          * If f is a node in the link list of the
00025                          * children of node i, then Sibling [f] is the
00026                          * next child of node i.
00027                          */
00028     Int Order [ ],     /* output order, of size nn. Order [i] = k
00029                          * if node i is the kth node of the reordered
00030                          * tree. */
00031     Int Stack [ ]      /* workspace of size nn */
00032 #ifndef NDEBUG
00033     , Int nn           /* nodes are in the range 0..nn-1. */
00034 #endif
00035 )
00036 {
00037     Int f;
00038     Int head;
00039     Int h;
00040     Int i;
00041
00042     #if 0
00043         /* ----- */
00044         /* recursive version (Stack [ ] is not used): */
00045         /* ----- */
00046
00047         /* this is simple, but can caouse stack overflow if nn is large */
00048         i = root ;
00049         for (f = Child [i] ; f != EMPTY ; f = Sibling [f])
00050         {
00051             k = AMD_post_tree (f, k, Child, Sibling, Order, Stack, nn) ;
00052         }
00053         Order [i] = k++ ;
00054         return (k) ;
00055     #endif
00056
00057     /* ----- */
00058     /* non-recursive version, using an explicit stack */
00059     /* ----- */
00060
00061     /* push root on the stack */
00062     head = 0 ;
00063     Stack [0] = root ;
00064
00065     while (head >= 0)
00066     {
00067         /* get head of stack */
00068         ASSERT (head < nn) ;
00069         i = Stack [head] ;
00070         AMD_DEBUG1 (("head of stack "ID" \n", i)) ;
00071         ASSERT (i >= 0 && i < nn) ;
00072
00073         if (Child [i] != EMPTY)
00074         {
00075             /* the children of i are not yet ordered */
00076             /* push each child onto the stack in reverse order */
00077             /* so that small ones at the head of the list get popped first */
00078             /* and the biggest one at the end of the list gets popped last */
00079             for (f = Child [i] ; f != EMPTY ; f = Sibling [f])
00080             {
00081                 head++ ;
00082                 ASSERT (head < nn) ;

```

```

00083         ASSERT (f >= 0 && f < nn) ;
00084     }
00085
00086     h = head ;
00087     ASSERT (head < nn) ;
00088
00089     for (f = Child [i] ; f != EMPTY ; f = Sibling [f])
00090     {
00091         ASSERT (h > 0) ;
00092         Stack [h--] = f ;
00093         AMD_DEBUG1 (("push "ID" on stack\n", f)) ;
00094         ASSERT (f >= 0 && f < nn) ;
00095     }
00096
00097     ASSERT (Stack [h] == i) ;
00098
00099     /* delete child list so that i gets ordered next time we see it */
00100     Child [i] = EMPTY ;
00101 }
00102 else
00103 {
00104     /* the children of i (if there were any) are already ordered */
00105     /* remove i from the stack and order it. Front i is kth front */
00106     head-- ;
00107     AMD_DEBUG1 (("pop "ID" order "ID"\n", i, k)) ;
00108     Order [i] = k++ ;
00109     ASSERT (k <= nn) ;
00110 }
00111
00112 #ifndef NDEBUG
00113 AMD_DEBUG1 (("Stack:")) ;
00114
00115 for (h = head ; h >= 0 ; h--)
00116 {
00117     Int j = Stack [h] ;
00118     AMD_DEBUG1 ((" "ID, j)) ;
00119     ASSERT (j >= 0 && j < nn) ;
00120 }
00121
00122 AMD_DEBUG1 (("Stack\n")) ;
00123 ASSERT (head < nn) ;
00124 #endif
00125 }
00126 return (k) ;
00127 }

```

## 4.25 amd/amd\_postorder.c File Reference

```
#include "amd_internal.h"
```

### Functions

- GLOBAL void [AMD\\_postorder](#) (Int nn, Int Parent[], Int Nv[], Int Fsize[], Int Order[], Int Child[], Int Sibling[], Int Stack[])

#### 4.25.1 Function Documentation

### 4.25.1.1 AMD\_postorder()

```
GLOBAL void AMD_postorder (
    Int nn,
    Int Parent[],
    Int Nv[],
    Int Fsize[],
    Int Order[],
    Int Child[],
    Int Sibling[],
    Int Stack[] )
```

Definition at line 15 of file [amd\\_postorder.c](#).

## 4.26 amd\_postorder.c

[Go to the documentation of this file.](#)

```
00001 /* ===== */
00002 /* == AMD_postorder ===== */
00003 /* ===== */
00004
00005 /* ----- */
00006 /* AMD, Copyright (c) Timothy A. Davis, */
00007 /* Patrick R. Amestoy, and Iain S. Duff. See ../README.txt for License. */
00008 /* email: DrTimothyAldenDavis@gmail.com */
00009 /* ----- */
00010
00011 /* Perform a postordering (via depth-first search) of an assembly tree. */
00012
00013 #include "amd_internal.h"
00014
00015 GLOBAL void AMD_postorder
00016 (
00017     /* inputs, not modified on output: */
00018     Int nn, /* nodes are in the range 0..nn-1 */
00019     Int Parent [], /* Parent [j] is the parent of j, or EMPTY if root */
00020     Int Nv [], /* Nv [j] > 0 number of pivots represented by node j,
00021     * or zero if j is not a node. */
00022     Int Fsize [], /* Fsize [j]: size of node j */
00023
00024     /* output, not defined on input: */
00025     Int Order [], /* output post-order */
00026
00027     /* workspaces of size nn: */
00028     Int Child [],
00029     Int Sibling [],
00030     Int Stack []
00031 )
00032 {
00033     Int i;
00034     Int j;
00035     Int k;
00036
00037     Int parent;
00038     Int frsize;
00039     Int f;
00040     Int fprev;
00041     Int maxfrsize;
00042     Int bigfprev;
00043     Int bigf;
00044     Int fnext;
00045
00046     for (j = 0 ; j < nn ; j++)
00047     {
00048         Child [j] = EMPTY ;
00049         Sibling [j] = EMPTY ;
00050     }
00051
00052     /* ----- */
00053     /* place the children in link lists - bigger elements tend to be last */
00054     /* ----- */
00055
00056     for (j = nn-1 ; j >= 0 ; j--)
00057     {
```



```

00058         if (Nv [j] > 0)
00059         {
00060             /* this is an element */
00061             parent = Parent [j] ;
00062             if (parent != EMPTY)
00063             {
00064                 /* place the element in link list of the children its parent */
00065                 /* bigger elements will tend to be at the end of the list */
00066                 Sibling [j] = Child [parent] ;
00067                 Child [parent] = j ;
00068             }
00069         }
00070     }
00071
00072     #ifndef NDEBUG
00073         Int nels;
00074         Int ff;
00075         Int nchild;
00076         AMD_DEBUG1 (("N\n\n===== AMD_postorder:\n"));
00077         nels = 0;
00078
00079         for (j = 0 ; j < nn ; j++)
00080         {
00081             if (Nv [j] > 0)
00082             {
00083                 AMD_DEBUG1 (( "ID" : nels "ID" npiv "ID" size "ID" parent "ID" maxfr "ID"\n", j, nels,
00084 Nv [j], Fsize [j], Parent [j], Fsize [j])) ;
00085
00086                 /* this is an element */
00087                 /* dump the link list of children */
00088                 nchild = 0 ;
00089                 AMD_DEBUG1 (("      Children: ")) ;
00090
00091                 for (ff = Child [j] ; ff != EMPTY ; ff = Sibling [ff])
00092                 {
00093                     AMD_DEBUG1 ((ID" ", ff)) ;
00094                     ASSERT (Parent [ff] == j) ;
00095                     nchild++ ;
00096                     ASSERT (nchild < nn) ;
00097                 }
00098
00099                 AMD_DEBUG1 (("N\n")) ;
00100                 parent = Parent [j] ;
00101
00102                 if (parent != EMPTY)
00103                 {
00104                     ASSERT (Nv [parent] > 0) ;
00105                 }
00106
00107                 nels++ ;
00108             }
00109         }
00110
00111         AMD_DEBUG1 (("N\nGo through the children of each node, and put\n" "the biggest child last in
each list:\n")) ;
00112
00113     #endif
00114
00115     /* ----- */
00116     /* place the largest child last in the list of children for each node */
00117     /* ----- */
00118
00119     for (i = 0 ; i < nn ; i++)
00120     {
00121         if (Nv [i] > 0 && Child [i] != EMPTY)
00122         {
00123             #ifndef NDEBUG
00124                 Int nchild ;
00125                 AMD_DEBUG1 (("Before partial sort, element "ID"\n", i)) ;
00126                 nchild = 0 ;
00127
00128                 for (f = Child [i] ; f != EMPTY ; f = Sibling [f])
00129                 {
00130                     ASSERT (f >= 0 && f < nn) ;
00131                     AMD_DEBUG1 (("      f: "ID" size: "ID"\n", f, Fsize [f])) ;
00132                     nchild++ ;
00133                     ASSERT (nchild <= nn) ;
00134                 }
00135             #endif
00136
00137             /* find the biggest element in the child list */
00138             fprev = EMPTY ;
00139             maxfrsize = EMPTY ;

```

```

00143         bigfprev = EMPTY ;
00144         bigf = EMPTY ;
00145
00146         for (f = Child [i] ; f != EMPTY ; f = Sibling [f])
00147         {
00148             ASSERT (f >= 0 && f < nn) ;
00149             frsize = Fsize [f] ;
00150
00151             if (frsize >= maxfrsize)
00152             {
00153                 /* this is the biggest seen so far */
00154                 maxfrsize = frsize ;
00155                 bigfprev = fprev ;
00156                 bigf = f ;
00157             }
00158
00159             fprev = f ;
00160         }
00161
00162         ASSERT (bigf != EMPTY) ;
00163
00164         fnext = Sibling [bigf] ;
00165
00166         AMD_DEBUG1 (("bigf "ID" maxfrsize "ID" bigfprev "ID" fnext "ID" fprev " ID"\n", bigf,
maxfrsize, bigfprev, fnext, fprev)) ;
00167
00168         if (fnext != EMPTY)
00169         {
00170             /* if fnext is EMPTY then bigf is already at the end of list */
00171
00172             if (bigfprev == EMPTY)
00173             {
00174                 /* delete bigf from the element of the list */
00175                 Child [i] = fnext ;
00176             }
00177             else
00178             {
00179                 /* delete bigf from the middle of the list */
00180                 Sibling [bigfprev] = fnext ;
00181             }
00182
00183             /* put bigf at the end of the list */
00184             Sibling [bigf] = EMPTY ;
00185             ASSERT (Child [i] != EMPTY) ;
00186             ASSERT (fprev != bigf) ;
00187             ASSERT (fprev != EMPTY) ;
00188             Sibling [fprev] = bigf ;
00189         }
00190
00191         #ifndef NDEBUG
00192
00193         AMD_DEBUG1 (("After partial sort, element "ID"\n", i)) ;
00194
00195         for (f = Child [i] ; f != EMPTY ; f = Sibling [f])
00196         {
00197             ASSERT (f >= 0 && f < nn) ;
00198             AMD_DEBUG1 ((" "ID" "ID"\n", f, Fsize [f])) ;
00199             ASSERT (Nv [f] > 0) ;
00200             nchild-- ;
00201         }
00202
00203         ASSERT (nchild == 0) ;
00204
00205         #endif
00206     }
00207 }
00208
00209 /* ----- */
00210 /* postorder the assembly tree */
00211 /* ----- */
00212
00213 for (i = 0 ; i < nn ; i++)
00214 {
00215     Order [i] = EMPTY ;
00216 }
00217
00218 k = 0 ;
00219
00220 for (i = 0 ; i < nn ; i++)
00221 {
00222     if (Parent [i] == EMPTY && Nv [i] > 0)
00223     {
00224         AMD_DEBUG1 (("Root of assembly tree "ID"\n", i)) ;
00225         k = AMD_post_tree (i, k, Child, Sibling, Order, Stack
00226             , nn
00227             #endif
00228

```

```

00229         ) ;
00230     }
00231 }
00232 }

```

## 4.27 amd/amd\_preprocess.c File Reference

```
#include "amd_internal.h"
```

### Functions

- GLOBAL void [AMD\\_preprocess](#) (Int n, const Int Ap[], const Int Ai[], Int Rp[], Int Ri[], Int W[], Int Flag[])

### 4.27.1 Function Documentation

#### 4.27.1.1 AMD\_preprocess()

```

GLOBAL void AMD_preprocess (
    Int n,
    const Int Ap[],
    const Int Ai[],
    Int Rp[],
    Int Ri[],
    Int W[],
    Int Flag[] )

```

Definition at line 29 of file [amd\\_preprocess.c](#).

## 4.28 amd\_preprocess.c

[Go to the documentation of this file.](#)

```

00001 /* ===== */
00002 /* == AMD_preprocess == */
00003 /* ===== */
00004
00005 /* ----- */
00006 /* AMD, Copyright (c) Timothy A. Davis, */
00007 /* Patrick R. Amestoy, and Iain S. Duff. See ../README.txt for License. */
00008 /* email: DrTimothyAldenDavis@gmail.com */
00009 /* ----- */
00010
00011 /* Sorts, removes duplicate entries, and transposes from the nonzero pattern of
00012 * a column-form matrix A, to obtain the matrix R. The input matrix can have
00013 * duplicate entries and/or unsorted columns (AMD_valid (n,Ap,Ai) must not be
00014 * AMD_INVALID).
00015 *
00016 * This input condition is NOT checked. This routine is not user-callable.
00017 */
00018
00019 #include "amd_internal.h"
00020
00021 /* ===== */
00022 /* == AMD_preprocess == */
00023 /* ===== */

```

```

00024
00025 /* AMD_preprocess does not check its input for errors or allocate workspace.
00026  * On input, the condition (AMD_valid (n,n,Ap,Ai) != AMD_INVALID) must hold.
00027  */
00028
00029 GLOBAL void AMD_preprocess
00030 (
00031     Int n,          /* input matrix: A is n-by-n */
00032     const Int Ap [ ], /* size n+1 */
00033     const Int Ai [ ], /* size nz = Ap [n] */
00034
00035     /* output matrix R: */
00036     Int Rp [ ],      /* size n+1 */
00037     Int Ri [ ],      /* size nz (or less, if duplicates present) */
00038
00039     Int W [ ],        /* workspace of size n */
00040     Int Flag [ ] /* workspace of size n */
00041 )
00042 {
00043
00044     /* ----- */
00045     /* local variables */
00046     /* ----- */
00047
00048     Int i;
00049     Int j;
00050     Int p;
00051     Int p2;
00052
00053     ASSERT (AMD_valid (n, n, Ap, Ai) != AMD_INVALID) ;
00054
00055     /* ----- */
00056     /* count the entries in each row of A (excluding duplicates) */
00057     /* ----- */
00058
00059     for (i = 0 ; i < n ; i++)
00060     {
00061         W [i] = 0 ; /* # of nonzeros in row i (excl duplicates) */
00062         Flag [i] = EMPTY ; /* Flag [i] = j if i appears in column j */
00063     }
00064
00065     for (j = 0 ; j < n ; j++)
00066     {
00067         p2 = Ap [j+1] ;
00068         for (p = Ap [j] ; p < p2 ; p++)
00069         {
00070             i = Ai [p] ;
00071             if (Flag [i] != j)
00072             {
00073                 /* row index i has not yet appeared in column j */
00074                 W [i]++ ; /* one more entry in row i */
00075                 Flag [i] = j ; /* flag row index i as appearing in col j*/
00076             }
00077         }
00078     }
00079
00080     /* ----- */
00081     /* compute the row pointers for R */
00082     /* ----- */
00083
00084     Rp [0] = 0 ;
00085     for (i = 0 ; i < n ; i++)
00086     {
00087         Rp [i+1] = Rp [i] + W [i] ;
00088     }
00089
00090     for (i = 0 ; i < n ; i++)
00091     {
00092         W [i] = Rp [i] ;
00093         Flag [i] = EMPTY ;
00094     }
00095
00096     /* ----- */
00097     /* construct the row form matrix R */
00098     /* ----- */
00099
00100     /* R = row form of pattern of A */
00101     for (j = 0 ; j < n ; j++)
00102     {
00103         p2 = Ap [j+1] ;
00104         for (p = Ap [j] ; p < p2 ; p++)
00105         {
00106             i = Ai [p] ;
00107             if (Flag [i] != j)
00108             {
00109                 /* row index i has not yet appeared in column j */
00110                 Ri [W [i]++] = j ; /* put col j in row i */

```

```

00111             Flag [i] = j ;           /* flag row index i as appearing in col j*/
00112         }
00113     }
00114 }
00115
00116 #ifndef NDEBUG
00117 ASSERT (AMD_valid (n, n, Rp, Ri) == AMD_OK) ;
00118
00119 for (j = 0 ; j < n ; j++)
00120 {
00121     ASSERT (W [j] == Rp [j+1]) ;
00122 }
00123
00124 #endif
00125 }

```

## 4.29 amd/amd\_valid.c File Reference

```
#include "amd_internal.h"
```

### Functions

- [GLOBAL Int AMD\\_valid](#) (Int n\_row, Int n\_col, const Int Ap[], const Int Ai[])

### 4.29.1 Function Documentation

#### 4.29.1.1 AMD\_valid()

```

GLOBAL Int AMD_valid (
    Int n_row,
    Int n_col,
    const Int Ap[],
    const Int Ai[] )

```

Definition at line 38 of file [amd\\_valid.c](#).

## 4.30 amd\_valid.c

[Go to the documentation of this file.](#)

```

00001 /* ===== */
00002 /* === AMD_valid ===== */
00003 /* ===== */
00004
00005 /* ----- */
00006 /* AMD, Copyright (c) Timothy A. Davis, */
00007 /* Patrick R. Amestoy, and Iain S. Duff. See ../README.txt for License. */
00008 /* email: DrTimothyAldenDavis@gmail.com */
00009 /* ----- */
00010
00011 /* Check if a column-form matrix is valid or not. The matrix A is
00012 * n_row-by-n_col. The row indices of entries in column j are in
00013 * Ai [Ap [j] ... Ap [j+1]-1]. Required conditions are:
00014 *
00015 * n_row >= 0
00016 * n_col >= 0

```

```

00017 * nz = Ap [n_col] >= 0          number of entries in the matrix
00018 * Ap [0] == 0
00019 * Ap [j] <= Ap [j+1] for all j in the range 0 to n_col.
00020 *   Ai [0 ... nz-1] must be in the range 0 to n_row-1.
00021 *
00022 * If any of the above conditions hold, AMD_INVALID is returned.  If the
00023 * following condition holds, AMD_OK_BUT_JUMBLED is returned (a warning,
00024 * not an error):
00025 *
00026 * row indices in Ai [Ap [j] ... Ap [j+1]-1] are not sorted in ascending
00027 * order, and/or duplicate entries exist.
00028 *
00029 * Otherwise, AMD_OK is returned.
00030 *
00031 * In v1.2 and earlier, this function returned TRUE if the matrix was valid
00032 * (now returns AMD_OK), or FALSE otherwise (now returns AMD_INVALID or
00033 * AMD_OK_BUT_JUMBLED).
00034 */
00035
00036 #include "amd_internal.h"
00037
00038 GLOBAL Int AMD_valid
00039 (
00040     /* inputs, not modified on output: */
00041     Int n_row,          /* A is n_row-by-n_col */
00042     Int n_col,
00043     const Int Ap [ ],   /* column pointers of A, of size n_col+1 */
00044     const Int Ai [ ]    /* row indices of A, of size nz = Ap [n_col] */
00045 )
00046 {
00047     Int nz;
00048     Int j;
00049     Int p1;
00050     Int p2;
00051     Int ilast;
00052     Int i;
00053     Int p;
00054     Int result = AMD_OK ;
00055
00056     if (n_row < 0 || n_col < 0 || Ap == ABIP_NULL || Ai == ABIP_NULL)
00057     {
00058         return (AMD_INVALID) ;
00059     }
00060
00061     nz = Ap [n_col] ;
00062     if (Ap [0] != 0 || nz < 0)
00063     {
00064         /* column pointers must start at Ap [0] = 0, and Ap [n] must be >= 0 */
00065         AMD_DEBUG0 (("column 0 pointer bad or nz < 0\n")) ;
00066         return (AMD_INVALID) ;
00067     }
00068
00069     for (j = 0 ; j < n_col ; j++)
00070     {
00071         p1 = Ap [j] ;
00072         p2 = Ap [j+1] ;
00073         AMD_DEBUG2 (("nColumn: "ID" p1: "ID" p2: "ID"\n", j, p1, p2)) ;
00074
00075         if (p1 > p2)
00076         {
00077             /* column pointers must be ascending */
00078             AMD_DEBUG0 (("column "ID" pointer bad\n", j)) ;
00079             return (AMD_INVALID) ;
00080         }
00081
00082         ilast = EMPTY ;
00083         for (p = p1 ; p < p2 ; p++)
00084         {
00085             i = Ai [p] ;
00086             AMD_DEBUG3 (("row: "ID"\n", i)) ;
00087
00088             if (i < 0 || i >= n_row)
00089             {
00090                 /* row index out of range */
00091                 AMD_DEBUG0 (("index out of range, col "ID" row "ID"\n", j, i));
00092                 return (AMD_INVALID) ;
00093             }
00094
00095             if (i <= ilast)
00096             {
00097                 /* row index unsorted, or duplicate entry present */
00098                 AMD_DEBUG1 (("index unsorted/dupl col "ID" row "ID"\n", j, i));
00099                 result = AMD_OK_BUT_JUMBLED ;
00100             }
00101
00102             ilast = i;
00103         }
00104     }

```

```
00104     }  
00105  
00106     return (result) ;  
00107 }
```

## 4.31 amd/SuiteSparse\_config.c File Reference

```
#include <math.h>  
#include <stdlib.h>  
#include <stdio.h>  
#include "SuiteSparse_config.h"
```

### Functions

- void [SuiteSparse\\_start](#) (void)
- void [SuiteSparse\\_finish](#) (void)
- void \* [SuiteSparse\\_malloc](#) (size\_t nitems, size\_t size\_of\_item)
- void \* [SuiteSparse\\_calloc](#) (size\_t nitems, size\_t size\_of\_item)
- void \* [SuiteSparse\\_realloc](#) (size\_t nitems\_new, size\_t nitems\_old, size\_t size\_of\_item, void \*p, int \*ok)
- void \* [SuiteSparse\\_free](#) (void \*p)
- void [SuiteSparse\\_tic](#) (abip\_float tic[2])
- abip\_float [SuiteSparse\\_toc](#) (abip\_float tic[2])
- abip\_float [SuiteSparse\\_time](#) (void)
- int [SuiteSparse\\_version](#) (int version[3])
- abip\_float [SuiteSparse\\_hypot](#) (abip\_float x, abip\_float y)
- int [SuiteSparse\\_divcomplex](#) (abip\_float ar, abip\_float ai, abip\_float br, abip\_float bi, abip\_float \*cr, abip\_float \*ci)

### Variables

- struct [SuiteSparse\\_config\\_struct](#) [SuiteSparse\\_config](#)

#### 4.31.1 Function Documentation

##### 4.31.1.1 SuiteSparse\_calloc()

```
void * SuiteSparse_calloc (  
    size_t nitems,  
    size_t size_of_item )
```

Definition at line 211 of file [SuiteSparse\\_config.c](#).

#### 4.31.1.2 SuiteSparse\_divcomplex()

```
int SuiteSparse_divcomplex (
    abip_float ar,
    abip_float ai,
    abip_float br,
    abip_float bi,
    abip_float * cr,
    abip_float * ci )
```

Definition at line 516 of file [SuiteSparse\\_config.c](#).

#### 4.31.1.3 SuiteSparse\_finish()

```
void SuiteSparse_finish (
    void )
```

Definition at line 173 of file [SuiteSparse\\_config.c](#).

#### 4.31.1.4 SuiteSparse\_free()

```
void * SuiteSparse_free (
    void * p )
```

Definition at line 311 of file [SuiteSparse\\_config.c](#).

#### 4.31.1.5 SuiteSparse\_hypot()

```
abip_float SuiteSparse_hypot (
    abip_float x,
    abip_float y )
```

Definition at line 464 of file [SuiteSparse\\_config.c](#).

#### 4.31.1.6 SuiteSparse\_malloc()

```
void * SuiteSparse_malloc (
    size_t nitems,
    size_t size_of_item )
```

Definition at line 182 of file [SuiteSparse\\_config.c](#).



#### 4.31.1.7 SuiteSparse\_realloc()

```
void * SuiteSparse_realloc (
    size_t nitems_new,
    size_t nitems_old,
    size_t size_of_item,
    void * p,
    int * ok )
```

Definition at line 247 of file [SuiteSparse\\_config.c](#).

#### 4.31.1.8 SuiteSparse\_start()

```
void SuiteSparse_start (
    void )
```

Definition at line 109 of file [SuiteSparse\\_config.c](#).

#### 4.31.1.9 SuiteSparse\_tic()

```
void SuiteSparse_tic (
    abip_float tic[2] )
```

Definition at line 373 of file [SuiteSparse\\_config.c](#).

#### 4.31.1.10 SuiteSparse\_time()

```
abip_float SuiteSparse_time (
    void )
```

Definition at line 414 of file [SuiteSparse\\_config.c](#).

#### 4.31.1.11 SuiteSparse\_toc()

```
abip_float SuiteSparse_toc (
    abip_float tic[2] )
```

Definition at line 397 of file [SuiteSparse\\_config.c](#).

#### 4.31.1.12 SuiteSparse\_version()

```
int SuiteSparse_version (
    int version[3] )
```

Definition at line 429 of file [SuiteSparse\\_config.c](#).

### 4.31.2 Variable Documentation

#### 4.31.2.1 SuiteSparse\_config

```
struct SuiteSparse_config_struct SuiteSparse_config
```

Definition at line 53 of file [SuiteSparse\\_config.c](#).

## 4.32 SuiteSparse\_config.c

[Go to the documentation of this file.](#)

```
00001 /* ===== */
00002 /* == SuiteSparse_config == */
00003 /* ===== */
00004
00005 /* SuiteSparse configuration : memory manager and printf functions. */
00006
00007 /* Copyright (c) 2013, Timothy A. Davis. No licensing restrictions
00008  * apply to this file or to the SuiteSparse_config directory.
00009  * Author: Timothy A. Davis.
00010  */
00011
00012 #include <math.h>
00013 #include <stdlib.h>
00014
00015 #ifndef NPRINT
00016 #include <stdio.h>
00017 #endif
00018
00019 #ifdef MATLAB_MEX_FILE
00020 #include "mex.h"
00021 #include "matrix.h"
00022 #endif
00023
00024 #ifdef ABIP_NULL
00025 #undef ABIP_NULL
00026 #define ABIP_NULL ((void *) 0)
00027 #endif
00028
00029 #include "SuiteSparse_config.h"
00030
00031 /* ----- */
00032 /* SuiteSparse_config : a global extern struct */
00033 /* ----- */
00034
00035 /* The SuiteSparse_config struct is available to all SuiteSparse functions and
00036  * to all applications that use those functions. It must be modified with
00037  * care, particularly in a multithreaded context. Normally, the application
00038  * will initialize this object once, via SuiteSparse_start, possibly followed
00039  * by application-specific modifications if the applications wants to use
00040  * alternative memory manager functions.
00041
00042  * The user can redefine these global pointers at run-time to change the
00043  * memory manager and printf function used by SuiteSparse.
00044
00045  * If -DNMALLOC is defined at compile-time, then no memory-manager is
00046  * specified. You must define them at run-time, after calling
00047  * SuiteSparse_start.
```

```

00048
00049     If -DPRINT is defined a compile time, then printf is disabled, and
00050     SuiteSparse will not use printf.
00051 */
00052
00053 struct SuiteSparse_config_struct SuiteSparse_config =
00054 {
00055     /* memory management functions */
00056     #ifndef NMALLOC
00057
00058         #ifdef MATLAB_MEX_FILE
00059             /* MATLAB mexFunction: */
00060             mxMalloc, mxCalloc, mxRealloc, mxFree,
00061
00062         #else
00063             /* standard ANSI C: */
00064             malloc, calloc, realloc, free,
00065         #endif
00066
00067     #else
00068         /* no memory manager defined; you must define one at run-time: */
00069         ABIP_NULL, ABIP_NULL, ABIP_NULL, ABIP_NULL,
00070     #endif
00071
00072     /* printf function */
00073     #ifndef NPRINT
00074
00075         #ifdef MATLAB_MEX_FILE
00076             /* MATLAB mexFunction: */
00077             mexPrintf,
00078
00079         #else
00080             /* standard ANSI C: */
00081             printf,
00082         #endif
00083
00084     #else
00085         /* printf is disabled */
00086         ABIP_NULL,
00087     #endif
00088
00089     SuiteSparse_hypot,
00090     SuiteSparse_divcomplex
00091 } ;
00092
00093 /* ----- */
00094 /* SuiteSparse_start */
00095 /* ----- */
00096
00097 /* All applications that use SuiteSparse should call SuiteSparse_start prior
00098 to using any SuiteSparse function. Only a single thread should call this
00099 function, in a multithreaded application. Currently, this function is
00100 optional, since all this function currently does is to set the four memory
00101 function pointers to ABIP_NULL (which tells SuiteSparse to use the default
00102 functions). In a multi-threaded application, only a single thread should
00103 call this function.
00104
00105 Future releases of SuiteSparse might enforce a requirement that
00106 SuiteSparse_start be called prior to calling any SuiteSparse function.
00107 */
00108
00109 void SuiteSparse_start ( void )
00110 {
00111     /* memory management functions */
00112     #ifndef NMALLOC
00113
00114         #ifdef MATLAB_MEX_FILE
00115             /* MATLAB mexFunction: */
00116             SuiteSparse_config.malloc_func = mxMalloc ;
00117             SuiteSparse_config.calloc_func = mxCalloc ;
00118             SuiteSparse_config.realloc_func = mxRealloc ;
00119             SuiteSparse_config.free_func = mxFree ;
00120
00121         #else
00122             /* standard ANSI C: */
00123             SuiteSparse_config.malloc_func = malloc ;
00124             SuiteSparse_config.calloc_func = calloc ;
00125             SuiteSparse_config.realloc_func = realloc ;
00126             SuiteSparse_config.free_func = free ;
00127         #endif
00128
00129     #else
00130         /* no memory manager defined; you must define one after calling
00131 SuiteSparse_start */
00132         SuiteSparse_config.malloc_func = ABIP_NULL ;
00133         SuiteSparse_config.calloc_func = ABIP_NULL ;
00134         SuiteSparse_config.realloc_func = ABIP_NULL ;

```

```

00134         SuiteSparse_config.free_func = ABIP_NULL ;
00135     #endif
00136
00137     /* printf function */
00138     #ifndef NPRINT
00139
00140     #ifdef MATLAB_MEX_FILE
00141         /* MATLAB mexFunction: */
00142         SuiteSparse_config.printf_func = mexPrintf ;
00143     #else
00144         /* standard ANSI C: */
00145         SuiteSparse_config.printf_func = printf ;
00146     #endif
00147
00148     #else
00149         /* printf is disabled */
00150         SuiteSparse_config.printf_func = ABIP_NULL ;
00151     #endif
00152
00153     /* math functions */
00154     SuiteSparse_config.hypot_func = SuiteSparse_hypot ;
00155     SuiteSparse_config.divcomplex_func = SuiteSparse_divcomplex ;
00156 }
00157
00158 /* ----- */
00159 /* SuiteSparse_finish */
00160 /* ----- */
00161
00162 /* This currently does nothing, but in the future, applications should call
00163 SuiteSparse_start before calling any SuiteSparse function, and then
00164 SuiteSparse_finish after calling the last SuiteSparse function, just before
00165 exiting. In a multithreaded application, only a single thread should call
00166 this function.
00167
00168 Future releases of SuiteSparse might use this function for any
00169 SuiteSparse-wide cleanup operations or finalization of statistics.
00170 */
00171
00172 void SuiteSparse_finish ( void )
00173 {
00174     /* do nothing */ ;
00175 }
00176
00177 /* ----- */
00178 /* SuiteSparse_malloc: malloc wrapper */
00179 /* ----- */
00180
00181 void *SuiteSparse_malloc      /* pointer to allocated block of memory */
00182 (
00183     size_t nitems,           /* number of items to malloc */
00184     size_t size_of_item     /* sizeof each item */
00185 )
00186 {
00187     void *p ;
00188     size_t size ;
00189     if (nitems < 1) nitems = 1 ;
00190     if (size_of_item < 1) size_of_item = 1 ;
00191     size = nitems * size_of_item ;
00192
00193     if (size != ((abip_float) nitems) * size_of_item)
00194     {
00195         /* size_t overflow */
00196         p = ABIP_NULL ;
00197     }
00198     else
00199     {
00200         p = (void *) (SuiteSparse_config.malloc_func) (size) ;
00201     }
00202     return (p) ;
00203 }
00204
00205 /* ----- */
00206 /* SuiteSparse_calloc: calloc wrapper */
00207 /* ----- */
00208
00209 void *SuiteSparse_calloc      /* pointer to allocated block of memory */
00210 (
00211     size_t nitems,           /* number of items to calloc */
00212     size_t size_of_item     /* sizeof each item */
00213 )
00214 {
00215     void *p ;
00216     size_t size ;
00217     if (nitems < 1) nitems = 1 ;
00218     if (size_of_item < 1) size_of_item = 1 ;

```

```

00221         size = nitems * size_of_item ;
00222
00223         if (size != ((abip_float) nitems) * size_of_item)
00224         {
00225             /* size_t overflow */
00226             p = ABIP_NULL ;
00227         }
00228         else
00229         {
00230             p = (void *) (SuiteSparse_config.calloc_func) (nitems, size_of_item) ;
00231         }
00232         return (p) ;
00233     }
00234
00235     /* ----- */
00236     /* SuiteSparse_realloc: realloc wrapper */
00237     /* ----- */
00238
00239     /* If p is non-NULL on input, it points to a previously allocated object of
00240     size nitems_old * size_of_item. The object is reallocated to be of size
00241     nitems_new * size_of_item. If p is NULL on input, then a new object of that
00242     size is allocated. On success, a pointer to the new object is returned,
00243     and ok is returned as 1. If the allocation fails, ok is set to 0 and a
00244     pointer to the old (unmodified) object is returned.
00245     */
00246
00247     void *SuiteSparse_realloc    /* pointer to reallocated block of memory, or to original block if the
    realloc failed. */
00248     (
00249         size_t nitems_new,      /* new number of items in the object */
00250         size_t nitems_old,      /* old number of items in the object */
00251         size_t size_of_item,    /* sizeof each item */
00252         void *p,                /* old object to reallocate */
00253         int *ok                 /* 1 if successful, 0 otherwise */
00254     )
00255     {
00256         size_t size ;
00257         if (nitems_old < 1) nitems_old = 1 ;
00258         if (nitems_new < 1) nitems_new = 1 ;
00259         if (size_of_item < 1) size_of_item = 1 ;
00260         size = nitems_new * size_of_item ;
00261
00262         if (size != ((abip_float) nitems_new) * size_of_item)
00263         {
00264             /* size_t overflow */
00265             (*ok) = 0 ;
00266         }
00267         else if (p == ABIP_NULL)
00268         {
00269             /* a fresh object is being allocated */
00270             p = SuiteSparse_malloc (nitems_new, size_of_item) ;
00271             (*ok) = (p != ABIP_NULL) ;
00272         }
00273         else if (nitems_old == nitems_new)
00274         {
00275             /* the object does not change; do nothing */
00276             (*ok) = 1 ;
00277         }
00278         else
00279         {
00280             /* change the size of the object from nitems_old to nitems_new */
00281             void *pnew ;
00282             pnew = (void *) (SuiteSparse_config.realloc_func) (p, size) ;
00283
00284             if (pnew == ABIP_NULL)
00285             {
00286                 if (nitems_new < nitems_old)
00287                 {
00288                     /* the attempt to reduce the size of the block failed,
00289                     but the old block is unchanged. So pretend to succeed. */
00290                     (*ok) = 1 ;
00291                 }
00292                 else
00293                 {
00294                     /* out of memory */
00295                     (*ok) = 0 ;
00296                 }
00297             }
00298             else
00299             {
00300                 /* success */
00301                 p = pnew ;
00302                 (*ok) = 1 ;
00303             }
00304         }
00305         return (p) ;
00306     }

```

```

00306
00307 /* ----- */
00308 /* SuiteSparse_free: free wrapper */
00309 /* ----- */
00310
00311 void *SuiteSparse_free      /* always returns ABIP_NULL */
00312 (
00313     void *p                /* block to free */
00314 )
00315 {
00316     if (p)
00317     {
00318         (SuiteSparse_config.free_func) (p) ;
00319     }
00320     return (ABIP_NULL) ;
00321 }
00322
00323
00324 /* ----- */
00325 /* SuiteSparse_tic: return current wall clock time */
00326 /* ----- */
00327
00328 /* Returns the number of seconds (tic [0]) and nanoseconds (tic [1]) since some
00329 * unspecified but fixed time in the past. If no timer is installed, zero is
00330 * returned. A scalar abip_float precision value for 'tic' could be used, but this
00331 * might cause loss of precision because clock_gettime returns the time from
00332 * some distant time in the past. Thus, an array of size 2 is used.
00333 *
00334 * The timer is enabled by default. To disable the timer, compile with
00335 * -DNTIMER. If enabled on a POSIX C 1993 system, the timer requires linking
00336 * with the -lrt library.
00337 *
00338 * example:
00339 *
00340 *     abip_float tic [2], r, s, t ;
00341 *     SuiteSparse_tic (tic) ;      // start the timer
00342 *     // do some work A
00343 *     t = SuiteSparse_toc (tic) ; // t is time for work A, in seconds
00344 *     // do some work B
00345 *     s = SuiteSparse_toc (tic) ; // s is time for work A and B, in seconds
00346 *     SuiteSparse_tic (tic) ;      // restart the timer
00347 *     // do some work C
00348 *     r = SuiteSparse_toc (tic) ; // s is time for work C, in seconds
00349 *
00350 * A abip_float array of size 2 is used so that this routine can be more easily
00351 * ported to non-POSIX systems. The caller does not rely on the POSIX
00352 * <time.h> include file.
00353 */
00354
00355 #ifdef SUITESPARSE_TIMER_ENABLED
00356
00357 #include <time.h>
00358
00359 void SuiteSparse_tic
00360 (
00361     abip_float tic [2]      /* output, contents undefined on input */
00362 )
00363 {
00364     /* POSIX C 1993 timer, requires -librt */
00365     struct timespec t ;
00366     clock_gettime (CLOCK_MONOTONIC, &t) ;
00367     tic [0] = (abip_float) (t.tv_sec) ;
00368     tic [1] = (abip_float) (t.tv_nsec) ;
00369 }
00370
00371 #else
00372
00373 void SuiteSparse_tic
00374 (
00375     abip_float tic [2]      /* output, contents undefined on input */
00376 )
00377 {
00378     /* no timer installed */
00379     tic [0] = 0 ;
00380     tic [1] = 0 ;
00381 }
00382
00383 #endif
00384
00385
00386 /* ----- */
00387 /* SuiteSparse_toc: return time since last tic */
00388 /* ----- */
00389
00390 /* Assuming SuiteSparse_tic is accurate to the nanosecond, this function is
00391 * accurate down to the nanosecond for 2^53 nanoseconds since the last call to
00392 * SuiteSparse_tic, which is sufficient for SuiteSparse (about 104 days). If

```

```

00393  * additional accuracy is required, the caller can use two calls to
00394  * SuiteSparse_tic and do the calculations differently.
00395  */
00396
00397  abip_float SuiteSparse_toc /* returns time in seconds since last tic */
00398  (
00399      abip_float tic [2] /* input, not modified from last call to SuiteSparse_tic */
00400  )
00401  {
00402      abip_float toc [2] ;
00403      SuiteSparse_tic (toc) ;
00404      return ((toc [0] - tic [0]) + 1e-9 * (toc [1] - tic [1])) ;
00405  }
00406
00407
00408  /* ----- */
00409  /* SuiteSparse_time: return current wallclock time in seconds */
00410  /* ----- */
00411
00412  /* This function might not be accurate down to the nanosecond. */
00413
00414  abip_float SuiteSparse_time /* returns current wall clock time in seconds */
00415  (
00416      void
00417  )
00418  {
00419      abip_float toc [2] ;
00420      SuiteSparse_tic (toc) ;
00421      return (toc [0] + 1e-9 * toc [1]) ;
00422  }
00423
00424
00425  /* ----- */
00426  /* SuiteSparse_version: return the current version of SuiteSparse */
00427  /* ----- */
00428
00429  int SuiteSparse_version
00430  (
00431      int version [3]
00432  )
00433  {
00434      if (version != ABIP_NULL)
00435      {
00436          version [0] = SUITESPARSE_MAIN_VERSION ;
00437          version [1] = SUITESPARSE_SUB_VERSION ;
00438          version [2] = SUITESPARSE_SUBSUB_VERSION ;
00439      }
00440      return (SUITESPARSE_VERSION) ;
00441  }
00442
00443  /* ----- */
00444  /* SuiteSparse_hypot */
00445  /* ----- */
00446
00447  /* There is an equivalent routine called hypot in <math.h>, which conforms
00448  * to ANSI C99. However, SuiteSparse does not assume that ANSI C99 is
00449  * available. You can use the ANSI C99 hypot routine with:
00450  *
00451  * #include <math.h>
00452  * #define SuiteSparse_config.hypot_func = hypot ;
00453  *
00454  * Default value of the SuiteSparse_config.hypot_func pointer is
00455  * SuiteSparse_hypot, defined below.
00456  *
00457  * s = hypot (x,y) computes s = sqrt (x*x + y*y) but does so more accurately.
00458  * The NaN cases for the abip_float relops x >= y and x+y == x are safely ignored.
00459  *
00460  * Source: Algorithm 312, "Absolute value and square root of a complex number,"
00461  * P. Friedland, Comm. ACM, vol 10, no 10, October 1967, page 665.
00462  */
00463
00464  abip_float SuiteSparse_hypot (abip_float x, abip_float y)
00465  {
00466      abip_float s;
00467      abip_float r ;
00468      x = fabs (x) ;
00469      y = fabs (y) ;
00470
00471      if (x >= y)
00472      {
00473          if (x + y == x)
00474          {
00475              s = x ;
00476          }
00477          else
00478          {
00479              r = y / x ;

```

```

00480             s = x * sqrt (1.0 + r*r) ;
00481         }
00482     }
00483     else
00484     {
00485         if (y + x == y)
00486         {
00487             s = y ;
00488         }
00489         else
00490         {
00491             r = x / y ;
00492             s = y * sqrt (1.0 + r*r) ;
00493         }
00494     }
00495     return (s) ;
00496 }
00497
00498 /* ----- */
00499 /* SuiteSparse_divcomplex */
00500 /* ----- */
00501
00502 /* c = a/b where c, a, and b are complex. The real and imaginary parts are
00503  * passed as separate arguments to this routine. The NaN case is ignored
00504  * for the abip_float relop br >= bi. Returns 1 if the denominator is zero,
00505  * 0 otherwise.
00506  *
00507  * This uses ACM Algo 116, by R. L. Smith, 1962, which tries to avoid
00508  * underflow and overflow.
00509  *
00510  * c can be the same variable as a or b.
00511  *
00512  * Default value of the SuiteSparse_config.divcomplex_func pointer is
00513  * SuiteSparse_divcomplex.
00514  */
00515
00516 int SuiteSparse_divcomplex
00517 (
00518     abip_float ar,
00519     abip_float ai,      /* real and imaginary parts of a */
00520     abip_float br,
00521     abip_float bi,      /* real and imaginary parts of b */
00522     abip_float *cr,
00523     abip_float *ci      /* real and imaginary parts of c */
00524 )
00525 {
00526     abip_float tr;
00527     abip_float ti;
00528     abip_float r;
00529     abip_float den;
00530
00531     if (fabs (br) >= fabs (bi))
00532     {
00533         r = bi / br ;
00534         den = br + r * bi ;
00535         tr = (ar + ai * r) / den ;
00536         ti = (ai - ar * r) / den ;
00537     }
00538     else
00539     {
00540         r = br / bi ;
00541         den = r * br + bi ;
00542         tr = (ar * r + ai) / den ;
00543         ti = (ai * r - ar) / den ;
00544     }
00545
00546     *cr = tr ;
00547     *ci = ti ;
00548     return (den == 0.) ;
00549 }

```

## 4.33 amd/SuiteSparse\_config.h File Reference

```

#include "glbopts.h"
#include "ctrlc.h"
#include <limits.h>
#include <stdlib.h>

```



## Classes

- struct [SuiteSparse\\_config\\_struct](#)

## Macros

- `#define SuiteSparse_long long`
- `#define SuiteSparse_long_max LONG_MAX`
- `#define SuiteSparse_long_idd "ld"`
- `#define SuiteSparse_long_id "%" SuiteSparse_long_idd`
- `#define SUITESPARSE_PRINTF(params) {if (SuiteSparse_config.printf_func != ABIP_NULL){(void) (SuiteSparse_config.printf_func) params;}}`
- `#define SUITESPARSE_HAS_VERSION_FUNCTION`
- `#define SUITESPARSE_DATE "Dec 28, 2017"`
- `#define SUITESPARSE_VER_CODE(main, sub) ((main) * 1000 + (sub))`
- `#define SUITESPARSE_MAIN_VERSION 5`
- `#define SUITESPARSE_SUB_VERSION 1`
- `#define SUITESPARSE_SUBSUB_VERSION 2`
- `#define SUITESPARSE_VERSION SUITESPARSE_VER_CODE(SUITESPARSE_MAIN_VERSION, SUITESPARSE_SUB_VERSION)`

## Functions

- void [SuiteSparse\\_start](#) (void)
- void [SuiteSparse\\_finish](#) (void)
- void \* [SuiteSparse\\_malloc](#) (size\_t nitems, size\_t size\_of\_item)
- void \* [SuiteSparse\\_calloc](#) (size\_t nitems, size\_t size\_of\_item)
- void \* [SuiteSparse\\_realloc](#) (size\_t nitems\_new, size\_t nitems\_old, size\_t size\_of\_item, void \*p, int \*ok)
- void \* [SuiteSparse\\_free](#) (void \*p)
- void [SuiteSparse\\_tic](#) (abip\_float tic[2])
- abip\_float [SuiteSparse\\_toc](#) (abip\_float tic[2])
- abip\_float [SuiteSparse\\_time](#) (void)
- abip\_float [SuiteSparse\\_hypot](#) (abip\_float x, abip\_float y)
- int [SuiteSparse\\_divcomplex](#) (abip\_float ar, abip\_float ai, abip\_float br, abip\_float bi, abip\_float \*cr, abip\_float \*ci)
- int [SuiteSparse\\_version](#) (int version[3])

## Variables

- struct [SuiteSparse\\_config\\_struct](#) [SuiteSparse\\_config](#)

### 4.33.1 Macro Definition Documentation

#### 4.33.1.1 SUITESPARSE\_DATE

```
#define SUITESPARSE_DATE "Dec 28, 2017"
```

Definition at line 237 of file [SuiteSparse\\_config.h](#).

#### 4.33.1.2 SUITESPARSE\_HAS\_VERSION\_FUNCTION

```
#define SUITESPARSE_HAS_VERSION_FUNCTION
```

Definition at line 235 of file [SuiteSparse\\_config.h](#).

#### 4.33.1.3 SuiteSparse\_long

```
#define SuiteSparse_long long
```

Definition at line 64 of file [SuiteSparse\\_config.h](#).

#### 4.33.1.4 SuiteSparse\_long\_id

```
#define SuiteSparse_long_id "%" SuiteSparse_long_idd
```

Definition at line 69 of file [SuiteSparse\\_config.h](#).

#### 4.33.1.5 SuiteSparse\_long\_idd

```
#define SuiteSparse_long_idd "ld"
```

Definition at line 66 of file [SuiteSparse\\_config.h](#).

#### 4.33.1.6 SuiteSparse\_long\_max

```
#define SuiteSparse_long_max LONG_MAX
```

Definition at line 65 of file [SuiteSparse\\_config.h](#).

#### 4.33.1.7 SUITESPARSE\_MAIN\_VERSION

```
#define SUITESPARSE_MAIN_VERSION 5
```

Definition at line 239 of file [SuiteSparse\\_config.h](#).

#### 4.33.1.8 SUITESPARSE\_PRINTF

```
#define SUITESPARSE_PRINTF(  
    params ) {if (SuiteSparse_config.printf_func != ABIP_NULL) {(void) (SuiteSparse_↵  
config.printf_func) params;}}
```

Definition at line 170 of file [SuiteSparse\\_config.h](#).

#### 4.33.1.9 SUITESPARSE\_SUB\_VERSION

```
#define SUITESPARSE_SUB_VERSION 1
```

Definition at line 240 of file [SuiteSparse\\_config.h](#).

#### 4.33.1.10 SUITESPARSE\_SUBSUB\_VERSION

```
#define SUITESPARSE_SUBSUB_VERSION 2
```

Definition at line 241 of file [SuiteSparse\\_config.h](#).

#### 4.33.1.11 SUITESPARSE\_VER\_CODE

```
#define SUITESPARSE_VER_CODE(  
    main,  
    sub ) ((main) * 1000 + (sub))
```

Definition at line 238 of file [SuiteSparse\\_config.h](#).

#### 4.33.1.12 SUITESPARSE\_VERSION

```
#define SUITESPARSE_VERSION SUITESPARSE_VER_CODE(SUITESPARSE_MAIN_VERSION, SUITESPARSE_SUB_VERSION)
```

Definition at line 242 of file [SuiteSparse\\_config.h](#).

### 4.33.2 Function Documentation

#### 4.33.2.1 SuiteSparse\_calloc()

```
void * SuiteSparse_calloc (
    size_t nitems,
    size_t size_of_item )
```

Definition at line 211 of file [SuiteSparse\\_config.c](#).

#### 4.33.2.2 SuiteSparse\_divcomplex()

```
int SuiteSparse_divcomplex (
    abip_float ar,
    abip_float ai,
    abip_float br,
    abip_float bi,
    abip_float * cr,
    abip_float * ci )
```

Definition at line 516 of file [SuiteSparse\\_config.c](#).

#### 4.33.2.3 SuiteSparse\_finish()

```
void SuiteSparse_finish (
    void )
```

Definition at line 173 of file [SuiteSparse\\_config.c](#).

#### 4.33.2.4 SuiteSparse\_free()

```
void * SuiteSparse_free (
    void * p )
```

Definition at line 311 of file [SuiteSparse\\_config.c](#).

#### 4.33.2.5 SuiteSparse\_hypot()

```
abip_float SuiteSparse_hypot (
    abip_float x,
    abip_float y )
```

Definition at line 464 of file [SuiteSparse\\_config.c](#).

#### 4.33.2.6 SuiteSparse\_malloc()

```
void * SuiteSparse_malloc (
    size_t nitems,
    size_t size_of_item )
```

Definition at line 182 of file [SuiteSparse\\_config.c](#).

#### 4.33.2.7 SuiteSparse\_realloc()

```
void * SuiteSparse_realloc (
    size_t nitems_new,
    size_t nitems_old,
    size_t size_of_item,
    void * p,
    int * ok )
```

Definition at line 247 of file [SuiteSparse\\_config.c](#).

#### 4.33.2.8 SuiteSparse\_start()

```
void SuiteSparse_start (
    void )
```

Definition at line 109 of file [SuiteSparse\\_config.c](#).

#### 4.33.2.9 SuiteSparse\_tic()

```
void SuiteSparse_tic (
    abip_float tic[2] )
```

Definition at line 373 of file [SuiteSparse\\_config.c](#).

#### 4.33.2.10 SuiteSparse\_time()

```
abip_float SuiteSparse_time (
    void )
```

Definition at line 414 of file [SuiteSparse\\_config.c](#).

#### 4.33.2.11 SuiteSparse\_toc()

```
abip_float SuiteSparse_toc (
    abip_float tic[2] )
```

Definition at line 397 of file [SuiteSparse\\_config.c](#).

#### 4.33.2.12 SuiteSparse\_version()

```
int SuiteSparse_version (
    int version[3] )
```

Definition at line 429 of file [SuiteSparse\\_config.c](#).

### 4.33.3 Variable Documentation

#### 4.33.3.1 SuiteSparse\_config

```
struct SuiteSparse_config_struct SuiteSparse_config [extern]
```

Definition at line 53 of file [SuiteSparse\\_config.c](#).

## 4.34 SuiteSparse\_config.h

[Go to the documentation of this file.](#)

```
00001 /* ===== */
00002 /* === SuiteSparse_config ===== */
00003 /* ===== */
00004
00005 /* Configuration file for SuiteSparse: a Suite of Sparse matrix packages
00006  * (AMD, COLAMD, CCOLAMD, CAMD, CHOLMOD, UMFPACK, CXSparse, and others).
00007  *
00008  * SuiteSparse_config.h provides the definition of the long integer. On most
00009  * systems, a C program can be compiled in LP64 mode, in which long's and
00010  * pointers are both 64-bits, and int's are 32-bits. Windows 64, however, uses
00011  * the LLP64 model, in which int's and long's are 32-bits, and long long's and
00012  * pointers are 64-bits.
00013  *
00014  * SuiteSparse packages that include long integer versions are
00015  * intended for the LP64 mode. However, as a workaround for Windows 64
00016  * (and perhaps other systems), the long integer can be redefined.
00017  *
00018  * If _WIN64 is defined, then the __int64 type is used instead of long.
00019  *
00020  * The long integer can also be defined at compile time. For example, this
00021  * could be added to SuiteSparse_config.mk:
00022  *
00023  * CFLAGS = -O -D'SuiteSparse_long=long long' \
00024  * -D'SuiteSparse_long_max=9223372036854775801' -D'SuiteSparse_long_idd="lld"
00025  *
00026  * This file defines SuiteSparse_long as either long (on all but _WIN64) or
00027  * __int64 on Windows 64. The intent is that a SuiteSparse_long is always a
00028  * 64-bit integer in a 64-bit code. ptrdiff_t might be a better choice than
00029  * long; it is always the same size as a pointer.
00030  *
00031  * This file also defines the SUITESPARSE_VERSION and related definitions.
```

```

00032  *
00033  * Copyright (c) 2012, Timothy A. Davis.  No licensing restrictions apply
00034  * to this file or to the SuiteSparse_config directory.
00035  * Author: Timothy A. Davis.
00036  */
00037
00038 #ifndef SUITESPARSE_CONFIG_H
00039 #define SUITESPARSE_CONFIG_H
00040
00041 #ifdef __cplusplus
00042 extern "C" {
00043 #endif
00044
00045 #include "glbopts.h"
00046 #include "ctrlc.h"
00047 #include <limits.h>
00048 #include <stdlib.h>
00049
00050 /* ===== */
00051 /* === SuiteSparse_long ===== */
00052 /* ===== */
00053
00054 #ifndef SuiteSparse_long
00055
00056 #ifdef _WIN64
00057
00058 #define SuiteSparse_long __int64
00059 #define SuiteSparse_long_max _I64_MAX
00060 #define SuiteSparse_long_idd "I64d"
00061
00062 #else
00063
00064 #define SuiteSparse_long long
00065 #define SuiteSparse_long_max LONG_MAX
00066 #define SuiteSparse_long_idd "ld"
00067
00068 #endif
00069 #define SuiteSparse_long_id "%" SuiteSparse_long_idd
00070 #endif
00071
00072 /* ===== */
00073 /* === SuiteSparse_config parameters and functions ===== */
00074 /* ===== */
00075
00076 /* SuiteSparse-wide parameters are placed in this struct.  It is meant to be
00077  * an extern, globally-accessible struct.  It is not meant to be updated
00078  * frequently by multiple threads.  Rather, if an application needs to modify
00079  * SuiteSparse_config, it should do it once at the beginning of the application,
00080  * before multiple threads are launched.
00081
00082  * The intent of these function pointers is that they not be used in your
00083  * application directly, except to assign them to the desired user-provided
00084  * functions.  Rather, you should use the
00085  */
00086
00087 struct SuiteSparse_config_struct
00088 {
00089     void *(*malloc_func) (size_t) ; /* pointer to malloc */
00090     void *(*calloc_func) (size_t, size_t) ; /* pointer to calloc */
00091     void *(*realloc_func) (void *, size_t) ; /* pointer to realloc */
00092     void *(*free_func) (void *) ; /* pointer to free */
00093     int (*printf_func) (const char *, ...) ; /* pointer to printf */
00094     abip_float (*hypot_func) (abip_float, abip_float) ; /* pointer to hypot */
00095     int (*divcomplex_func) (abip_float, abip_float, abip_float, abip_float *, abip_float
00096 *) ;
00097 };
00098
00099 extern struct SuiteSparse_config_struct SuiteSparse_config ;
00100
00101 void SuiteSparse_start ( void ) ; /* called to start SuiteSparse */
00102 void SuiteSparse_finish ( void ) ; /* called to finish SuiteSparse */
00103
00104 void *SuiteSparse_malloc /* pointer to allocated block of memory */
00105 (
00106     size_t nitems, /* number of items to malloc (>=1 is enforced) */
00107     size_t size_of_item /* sizeof each item */
00108 ) ;
00109
00110 void *SuiteSparse_calloc /* pointer to allocated block of memory */
00111 (
00112     size_t nitems, /* number of items to calloc (>=1 is enforced) */
00113     size_t size_of_item /* sizeof each item */
00114 ) ;
00115
00116 void *SuiteSparse_realloc /* pointer to reallocated block of memory, or
00117 to original block if the realloc failed. */

```

```

00118 (
00119     size_t nitems_new,      /* new number of items in the object */
00120     size_t nitems_old,      /* old number of items in the object */
00121     size_t size_of_item,    /* sizeof each item */
00122     void *p,                /* old object to reallocate */
00123     int *ok                  /* 1 if successful, 0 otherwise */
00124 );
00125
00126 void *SuiteSparse_free      /* always returns NULL */
00127 (
00128     void *p                  /* block to free */
00129 );
00130
00131 void SuiteSparse_tic        /* start the timer */
00132 (
00133     abip_float tic [2]       /* output, contents undefined on input */
00134 );
00135
00136 abip_float SuiteSparse_toc  /* return time in seconds since last tic */
00137 (
00138     abip_float tic [2]       /* input: from last call to SuiteSparse_tic */
00139 );
00140
00141 abip_float SuiteSparse_time /* returns current wall clock time in seconds */
00142 (
00143     void
00144 );
00145
00146 /* returns sqrt (x^2 + y^2), computed reliably */
00147 abip_float SuiteSparse_hypot (abip_float x, abip_float y) ;
00148
00149 /* complex division of c = a/b */
00150 int SuiteSparse_divcomplex
00151 (
00152     abip_float ar,
00153     abip_float ai,          /* real and imaginary parts of a */
00154     abip_float br,
00155     abip_float bi,          /* real and imaginary parts of b */
00156     abip_float *cr,
00157     abip_float *ci          /* real and imaginary parts of c */
00158 );
00159
00160 /* determine which timer to use, if any */
00161 #ifndef NTIMER
00162 #ifdef _POSIX_C_SOURCE
00163 #if _POSIX_C_SOURCE >= 199309L
00164 #define SUITESPARSE_TIMER_ENABLED
00165 #endif
00166 #endif
00167 #endif
00168
00169 /* SuiteSparse printf macro */
00170 #define SUITESPARSE_PRINTF(params) {if (SuiteSparse_config.printf_func != ABIP_NULL){(void)
(SuiteSparse_config.printf_func) params;}}
00171
00172 /* ===== */
00173 /* === SuiteSparse version ===== */
00174 /* ===== */
00175
00176 /* SuiteSparse is not a package itself, but a collection of packages, some of
00177 * which must be used together (UMFPACK requires AMD, CHOLMOD requires AMD,
00178 * COLAMD, CAMD, and COLAMD, etc). A version number is provided here for the
00179 * collection itself. The versions of packages within each version of
00180 * SuiteSparse are meant to work together. Combining one package from one
00181 * version of SuiteSparse, with another package from another version of
00182 * SuiteSparse, may or may not work.
00183 *
00184 * SuiteSparse contains the following packages:
00185 *
00186 * SuiteSparse_config version 5.1.2 (version always the same as SuiteSparse)
00187 * GraphBLAS          version 1.1.2
00188 * ssget              version 2.0.0
00189 * AMD                version 2.4.6
00190 * BTF                version 1.2.6
00191 * CAMD              version 2.4.6
00192 * COLAMD            version 2.9.6
00193 * CHOLMOD           version 3.0.11
00194 * COLAMD            version 2.9.6
00195 * CSparse           version 3.2.0
00196 * CXSparse          version 3.2.0
00197 * GPUQREngine       version 1.0.5
00198 * KLU               version 1.3.8
00199 * LDL               version 2.2.6
00200 * RBio              version 2.2.6
00201 * SPQR              version 2.0.8
00202 * SuiteSparse_GPU runtime version 1.0.5
00203 * UMFPACK           version 5.7.6

```



```

00204 * MATLAB_Tools    various packages & M-files
00205 * xerbla          version 1.0.3
00206 *
00207 * Other package dependencies:
00208 * BLAS            required by CHOLMOD and UMFPACK
00209 * LAPACK          required by CHOLMOD
00210 * METIS 5.1.0     required by CHOLMOD (optional) and KLU (optional)
00211 * CUBLAS, CUDART  NVIDIA libraries required by CHOLMOD and SPQR when
00212 *                they are compiled with GPU acceleration.
00213 */
00214
00215 int SuiteSparse_version    /* returns SUITESPARSE_VERSION */
00216 (
00217     /* output, not defined on input. Not used if NULL. Returns
00218     the three version codes in version [0..2]:
00219     version [0] is SUITESPARSE_MAIN_VERSION
00220     version [1] is SUITESPARSE_SUB_VERSION
00221     version [2] is SUITESPARSE_SUBSUB_VERSION
00222     */
00223     int version [3]
00224 ) ;
00225
00226 /* Versions prior to 4.2.0 do not have the above function. The following
00227 code fragment will work with any version of SuiteSparse:
00228
00229 #ifdef SUITESPARSE_HAS_VERSION_FUNCTION
00230 v = SuiteSparse_version (NULL) ;
00231 #else
00232 v = SUITESPARSE_VERSION ;
00233 #endif
00234 */
00235 #define SUITESPARSE_HAS_VERSION_FUNCTION
00236
00237 #define SUITESPARSE_DATE "Dec 28, 2017"
00238 #define SUITESPARSE_VER_CODE(main,sub) ((main) * 1000 + (sub))
00239 #define SUITESPARSE_MAIN_VERSION 5
00240 #define SUITESPARSE_SUB_VERSION 1
00241 #define SUITESPARSE_SUBSUB_VERSION 2
00242 #define SUITESPARSE_VERSION \
00243     SUITESPARSE_VER_CODE(SUITESPARSE_MAIN_VERSION,SUITESPARSE_SUB_VERSION)
00244
00245 #ifdef __cplusplus
00246 }
00247 #endif
00248 #endif

```

## 4.35 csparse/Include/cs.h File Reference

```

#include <stdlib.h>
#include <limits.h>
#include <math.h>
#include <stdio.h>
#include <stddef.h>
#include "glbopts.h"

```

### Classes

- struct [cs\\_sparse](#)
- struct [cs\\_symbolic](#)
- struct [cs\\_numeric](#)
- struct [cs\\_dmperm\\_results](#)

### Macros

- #define [CS\\_VER](#) 3 /\* CSparse Version \*/
- #define [CS\\_SUBVER](#) 2

- `#define CS_SUBSUB 0`
- `#define CS_DATE "Sept 12, 2017" /* CSparse release date */`
- `#define CS_COPYRIGHT "Copyright (c) Timothy A. Davis, 2006-2016"`
- `#define csi abip_int`
- `#define CS_MAX(a, b) (((a) > (b)) ? (a) : (b))`
- `#define CS_MIN(a, b) (((a) < (b)) ? (a) : (b))`
- `#define CS_FLIP(i) (-i)-2`
- `#define CS_UNFLIP(i) (((i) < 0) ? CS_FLIP(i) : (i))`
- `#define CS_MARKED(w, j) (w[j] < 0)`
- `#define CS_MARK(w, j) { w[j] = CS_FLIP(w[j]); }`
- `#define CS_CSC(A) (A && (A->nz == -1))`
- `#define CS_TRIPLET(A) (A && (A->nz >= 0))`

## Typedefs

- `typedef struct cs_sparse cs`
- `typedef struct cs_symbolic css`
- `typedef struct cs_numeric csn`
- `typedef struct cs_dmperm_results csd`

## Functions

- `cs * cs_add (const cs *A, const cs *B, double alpha, double beta)`
- `csi cs_cholsol (csi order, const cs *A, double *b)`
- `cs * cs_compress (const cs *T)`
- `csi cs_dupl (cs *A)`
- `csi cs_entry (cs *T, csi i, csi j, double x)`
- `csi cs_gaxpy (const cs *A, const double *x, double *y)`
- `cs * cs_load (FILE *f)`
- `csi cs_lusol (csi order, const cs *A, double *b, double tol)`
- `cs * cs_multiply (const cs *A, const cs *B)`
- `double cs_norm (const cs *A)`
- `csi cs_print (const cs *A, csi brief)`
- `csi cs_qrsol (csi order, const cs *A, double *b)`
- `cs * cs_transpose (const cs *A, csi values)`
- `void * cs_calloc (csi n, size_t size)`
- `void * cs_free (void *p)`
- `void * cs_realloc (void *p, csi n, size_t size, csi *ok)`
- `cs * cs_spalloc (csi m, csi n, csi nzmax, csi values, csi triplet)`
- `cs * cs_sprealloc (cs *A, csi nzmax)`
- `void * cs_malloc (csi n, size_t size)`
- `csi * cs_amd (csi order, const cs *A)`
- `csn * cs_chol (const cs *A, const css *S)`
- `csd * cs_dmperm (const cs *A, csi seed)`
- `csi cs_droptol (cs *A, double tol)`
- `csi cs_dropzeros (cs *A)`
- `csi cs_happly (const cs *V, csi i, double beta, double *x)`
- `csi cs_ipvec (const csi *p, const double *b, double *x, csi n)`
- `csi cs_lsolve (const cs *L, double *x)`
- `csi cs_ltsolve (const cs *L, double *x)`
- `csn * cs_lu (const cs *A, const css *S, double tol)`
- `cs * cs_permute (const cs *A, const csi *pinv, const csi *q, csi values)`

- `csi * cs_pinv` (const `csi` \*p, `csi` n)
- `csi cs_pvec` (const `csi` \*p, const double \*b, double \*x, `csi` n)
- `csn * cs_qr` (const `cs` \*A, const `css` \*S)
- `css * cs_schol` (`csi` order, const `cs` \*A)
- `css * cs_sqr` (`csi` order, const `cs` \*A, `csi` qr)
- `cs * cs_symperm` (const `cs` \*A, const `csi` \*pinv, `csi` values)
- `csi cs_updown` (`cs` \*L, `csi` sigma, const `cs` \*C, const `csi` \*parent)
- `csi cs_usolve` (const `cs` \*U, double \*x)
- `csi cs_utsolve` (const `cs` \*U, double \*x)
- `css * cs_sfree` (`css` \*S)
- `csn * cs_nfree` (`csn` \*N)
- `csd * cs_dfree` (`csd` \*D)
- `csi * cs_counts` (const `cs` \*A, const `csi` \*parent, const `csi` \*post, `csi` ata)
- `double cs_cumsum` (`csi` \*p, `csi` \*c, `csi` n)
- `csi cs_dfs` (`csi` j, `cs` \*G, `csi` top, `csi` \*xi, `csi` \*pstack, const `csi` \*pinv)
- `csi cs_ereach` (const `cs` \*A, `csi` k, const `csi` \*parent, `csi` \*s, `csi` \*w)
- `csi * cs_etree` (const `cs` \*A, `csi` ata)
- `csi cs_fkeep` (`cs` \*A, `csi` (\*fkeep)(`csi`, `csi`, double, void \*), void \*other)
- `double cs_house` (double \*x, double \*beta, `csi` n)
- `csi cs_leaf` (`csi` i, `csi` j, const `csi` \*first, `csi` \*maxfirst, `csi` \*prevleaf, `csi` \*ancestor, `csi` \*jleaf)
- `csi * cs_maxtrans` (const `cs` \*A, `csi` seed)
- `csi * cs_post` (const `csi` \*parent, `csi` n)
- `csi * cs_randperm` (`csi` n, `csi` seed)
- `csi cs_reach` (`cs` \*G, const `cs` \*B, `csi` k, `csi` \*xi, const `csi` \*pinv)
- `csi cs_scatter` (const `cs` \*A, `csi` j, double beta, `csi` \*w, double \*x, `csi` mark, `cs` \*C, `csi` nz)
- `csd * cs_scc` (`cs` \*A)
- `csi cs_spsolve` (`cs` \*G, const `cs` \*B, `csi` k, `csi` \*xi, double \*x, const `csi` \*pinv, `csi` lo)
- `csi cs_tdfs` (`csi` j, `csi` k, `csi` \*head, const `csi` \*next, `csi` \*post, `csi` \*stack)
- `csd * cs_dalloc` (`csi` m, `csi` n)
- `csd * cs_ddone` (`csd` \*D, `cs` \*C, void \*w, `csi` ok)
- `cs * cs_done` (`cs` \*C, void \*w, void \*x, `csi` ok)
- `csi * cs_idone` (`csi` \*p, `cs` \*C, void \*w, `csi` ok)
- `csn * cs_ndone` (`csn` \*N, `cs` \*C, void \*w, void \*x, `csi` ok)

## 4.35.1 Macro Definition Documentation

### 4.35.1.1 CS\_COPYRIGHT

```
#define CS_COPYRIGHT "Copyright (c) Timothy A. Davis, 2006-2016"
```

Definition at line 16 of file `cs.h`.

### 4.35.1.2 CS\_CSC

```
#define CS_CSC(
    A ) (A && (A->nz == -1))
```

Definition at line 152 of file `cs.h`.

#### 4.35.1.3 CS\_DATE

```
#define CS_DATE "Sept 12, 2017" /* CSparse release date */
```

Definition at line 15 of file [cs.h](#).

#### 4.35.1.4 CS\_FLIP

```
#define CS_FLIP(  
    i )  (-(i)-2)
```

Definition at line 148 of file [cs.h](#).

#### 4.35.1.5 CS\_MARK

```
#define CS_MARK(  
    w,  
    j ) { w [j] = CS_FLIP (w [j]) ; }
```

Definition at line 151 of file [cs.h](#).

#### 4.35.1.6 CS\_MARKED

```
#define CS_MARKED(  
    w,  
    j ) (w [j] < 0)
```

Definition at line 150 of file [cs.h](#).

#### 4.35.1.7 CS\_MAX

```
#define CS_MAX(  
    a,  
    b ) ((a) > (b)) ? (a) : (b)
```

Definition at line 146 of file [cs.h](#).

#### 4.35.1.8 CS\_MIN

```
#define CS_MIN(  
    a,  
    b ) ((a) < (b)) ? (a) : (b)
```

Definition at line 147 of file [cs.h](#).

#### 4.35.1.9 CS\_SUBSUB

```
#define CS_SUBSUB 0
```

Definition at line 14 of file [cs.h](#).

#### 4.35.1.10 CS\_SUBVER

```
#define CS_SUBVER 2
```

Definition at line 13 of file [cs.h](#).

#### 4.35.1.11 CS\_TRIPLET

```
#define CS_TRIPLET(  
    A ) (A && (A->nz >= 0))
```

Definition at line 153 of file [cs.h](#).

#### 4.35.1.12 CS\_UNFLIP

```
#define CS_UNFLIP(  
    i ) ((i) < 0) ? CS_FLIP(i) : (i)
```

Definition at line 149 of file [cs.h](#).

#### 4.35.1.13 CS\_VER

```
#define CS_VER 3 /* CSparse Version */
```

Definition at line 12 of file [cs.h](#).

#### 4.35.1.14 csi

```
#define csi abip_int
```

Definition at line 24 of file [cs.h](#).

### 4.35.2 Typedef Documentation

#### 4.35.2.1 cs

```
typedef struct cs_sparse cs
```

#### 4.35.2.2 csd

```
typedef struct cs_dmperm_results csd
```

#### 4.35.2.3 csn

```
typedef struct cs_numeric csn
```

#### 4.35.2.4 css

```
typedef struct cs_symbolic css
```

### 4.35.3 Function Documentation

#### 4.35.3.1 cs\_add()

```
cs * cs_add (
    const cs * A,
    const cs * B,
    double alpha,
    double beta )
```

Definition at line 3 of file [cs\\_add.c](#).

#### 4.35.3.2 cs\_amd()

```
csi * cs_amd (
    csi order,
    const cs * A )
```

Definition at line 18 of file [cs\\_amd.c](#).

#### 4.35.3.3 cs\_calloc()

```
void * cs_calloc (
    csi n,
    size_t size )
```

Definition at line 16 of file [cs\\_malloc.c](#).

#### 4.35.3.4 cs\_chol()

```
csn * cs_chol (
    const cs * A,
    const css * S )
```

Definition at line 3 of file [cs\\_chol.c](#).

#### 4.35.3.5 cs\_cholsol()

```
csi cs_cholsol (
    csi order,
    const cs * A,
    double * b )
```

Definition at line 3 of file [cs\\_cholsol.c](#).

#### 4.35.3.6 cs\_compress()

```
cs * cs_compress (
    const cs * T )
```

Definition at line 3 of file [cs\\_compress.c](#).

#### 4.35.3.7 cs\_counts()

```
csi * cs_counts (
    const cs * A,
    const csi * parent,
    const csi * post,
    csi ata )
```

Definition at line 17 of file [cs\\_counts.c](#).

#### 4.35.3.8 cs\_cumsum()

```
double cs_cumsum (
    csi * p,
    csi * c,
    csi n )
```

Definition at line 3 of file [cs\\_cumsum.c](#).

#### 4.35.3.9 cs\_dalloc()

```
csd * cs_dalloc (
    csi m,
    csi n )
```

Definition at line 66 of file [cs\\_util.c](#).

#### 4.35.3.10 cs\_ddone()

```
csd * cs_ddone (
    csd * D,
    cs * C,
    void * w,
    csi ok )
```

Definition at line 115 of file [cs\\_util.c](#).

#### 4.35.3.11 cs\_dfree()

```
csd * cs_dfree (
    csd * D )
```

Definition at line 79 of file [cs\\_util.c](#).



#### 4.35.3.12 cs\_dfs()

```
csi cs_dfs (
    csi j,
    cs * G,
    csi top,
    csi * xi,
    csi * pstack,
    const csi * pinv )
```

Definition at line 3 of file [cs\\_dfs.c](#).

#### 4.35.3.13 cs\_dmperm()

```
csd * cs_dmperm (
    const cs * A,
    csi seed )
```

Definition at line 68 of file [cs\\_dmperm.c](#).

#### 4.35.3.14 cs\_done()

```
cs * cs_done (
    cs * C,
    void * w,
    void * x,
    csi ok )
```

Definition at line 90 of file [cs\\_util.c](#).

#### 4.35.3.15 cs\_droptol()

```
csi cs_droptol (
    cs * A,
    double tol )
```

Definition at line 6 of file [cs\\_droptol.c](#).

#### 4.35.3.16 cs\_dropzeros()

```
csi cs_dropzeros (
    cs * A )
```

Definition at line 6 of file [cs\\_dropzeros.c](#).

#### 4.35.3.17 cs\_dupl()

```
csi cs_dupl (
    cs * A )
```

Definition at line 3 of file [cs\\_dupl.c](#).

#### 4.35.3.18 cs\_entry()

```
csi cs_entry (
    cs * T,
    csi i,
    csi j,
    double x )
```

Definition at line 3 of file [cs\\_entry.c](#).

#### 4.35.3.19 cs\_ereach()

```
csi cs_ereach (
    const cs * A,
    csi k,
    const csi * parent,
    csi * s,
    csi * w )
```

Definition at line 3 of file [cs\\_ereach.c](#).

#### 4.35.3.20 cs\_etree()

```
csi * cs_etree (
    const cs * A,
    csi ata )
```

Definition at line 3 of file [cs\\_etree.c](#).

#### 4.35.3.21 cs\_fkeep()

```
csi cs_fkeep (
    cs * A,
    csi(*) (csi, csi, double, void *) fkeep,
    void * other )
```

Definition at line 3 of file [cs\\_fkeep.c](#).

#### 4.35.3.22 cs\_free()

```
void * cs_free (
    void * p )
```

Definition at line 22 of file [cs\\_malloc.c](#).

#### 4.35.3.23 cs\_gaxpy()

```
csi cs_gaxpy (
    const cs * A,
    const double * x,
    double * y )
```

Definition at line 3 of file [cs\\_gaxpy.c](#).

#### 4.35.3.24 cs\_happly()

```
csi cs_happly (
    const cs * V,
    csi i,
    double beta,
    double * x )
```

Definition at line 3 of file [cs\\_happly.c](#).

#### 4.35.3.25 cs\_house()

```
double cs_house (
    double * x,
    double * beta,
    csi n )
```

Definition at line 4 of file [cs\\_house.c](#).

#### 4.35.3.26 cs\_idone()

```
csi * cs_idone (
    csi * p,
    cs * C,
    void * w,
    csi ok )
```

Definition at line 98 of file [cs\\_util.c](#).

#### 4.35.3.27 cs\_ipvec()

```
csi cs_ipvec (
    const csi * p,
    const double * b,
    double * x,
    csi n )
```

Definition at line 3 of file [cs\\_ipvec.c](#).

#### 4.35.3.28 cs\_leaf()

```
csi cs_leaf (
    csi i,
    csi j,
    const csi * first,
    csi * maxfirst,
    csi * prevleaf,
    csi * ancestor,
    csi * jleaf )
```

Definition at line 3 of file [cs\\_leaf.c](#).

#### 4.35.3.29 cs\_load()

```
cs * cs_load (
    FILE * f )
```

Definition at line 3 of file [cs\\_load.c](#).

#### 4.35.3.30 cs\_lsolve()

```
csi cs_lsolve (
    const cs * L,
    double * x )
```

Definition at line 3 of file [cs\\_lsolve.c](#).

#### 4.35.3.31 cs\_ltsolve()

```
csi cs_ltsolve (
    const cs * L,
    double * x )
```

Definition at line 3 of file [cs\\_ltsolve.c](#).

#### 4.35.3.32 cs\_lu()

```
csn * cs_lu (
    const cs * A,
    const css * S,
    double tol )
```

Definition at line 3 of file [cs\\_lu.c](#).

#### 4.35.3.33 cs\_lusol()

```
csi cs_lusol (
    csi order,
    const cs * A,
    double * b,
    double tol )
```

Definition at line 3 of file [cs\\_lusol.c](#).

#### 4.35.3.34 cs\_malloc()

```
void * cs_malloc (
    csi n,
    size_t size )
```

Definition at line 10 of file [cs\\_malloc.c](#).

#### 4.35.3.35 cs\_maxtrans()

```
csi * cs_maxtrans (
    const cs * A,
    csi seed )
```

Definition at line 44 of file [cs\\_maxtrans.c](#).

#### 4.35.3.36 cs\_multiply()

```
cs * cs_multiply (
    const cs * A,
    const cs * B )
```

Definition at line 3 of file [cs\\_multiply.c](#).

#### 4.35.3.37 cs\_ndone()

```
csn * cs_ndone (
    csn * N,
    cs * C,
    void * w,
    void * x,
    csi ok )
```

Definition at line 106 of file [cs\\_util.c](#).

#### 4.35.3.38 cs\_nfree()

```
csn * cs_nfree (
    csn * N )
```

Definition at line 43 of file [cs\\_util.c](#).

#### 4.35.3.39 cs\_norm()

```
double cs_norm (
    const cs * A )
```

Definition at line 3 of file [cs\\_norm.c](#).

#### 4.35.3.40 cs\_permute()

```
cs * cs_permute (
    const cs * A,
    const csi * pinv,
    const csi * q,
    csi values )
```

Definition at line 3 of file [cs\\_permute.c](#).

#### 4.35.3.41 cs\_pinv()

```
csi * cs_pinv (
    const csi * p,
    csi n )
```

Definition at line 3 of file [cs\\_pinv.c](#).

#### 4.35.3.42 cs\_post()

```
csi * cs_post (
    const csi * parent,
    csi n )
```

Definition at line 3 of file [cs\\_post.c](#).

#### 4.35.3.43 cs\_print()

```
csi cs_print (
    const cs * A,
    csi brief )
```

Definition at line 3 of file [cs\\_print.c](#).

#### 4.35.3.44 cs\_pvec()

```
csi cs_pvec (
    const csi * p,
    const double * b,
    double * x,
    csi n )
```

Definition at line 3 of file [cs\\_pvec.c](#).

#### 4.35.3.45 cs\_qr()

```
csn * cs_qr (
    const cs * A,
    const css * S )
```

Definition at line 3 of file [cs\\_qr.c](#).

#### 4.35.3.46 cs\_qrsol()

```
csi cs_qrsol (
    csi order,
    const cs * A,
    double * b )
```

Definition at line 3 of file [cs\\_qrsol.c](#).

#### 4.35.3.47 cs\_randperm()

```
csi * cs_randperm (
    csi n,
    csi seed )
```

Definition at line 5 of file [cs\\_randperm.c](#).

#### 4.35.3.48 cs\_reach()

```
csi cs_reach (
    cs * G,
    const cs * B,
    csi k,
    csi * xi,
    const csi * pinv )
```

Definition at line 4 of file [cs\\_reach.c](#).

#### 4.35.3.49 cs\_realloc()

```
void * cs_realloc (
    void * p,
    csi n,
    size_t size,
    csi * ok )
```

Definition at line 29 of file [cs\\_malloc.c](#).

#### 4.35.3.50 cs\_scatter()

```
csi cs_scatter (
    const cs * A,
    csi j,
    double beta,
    csi * w,
    double * x,
    csi mark,
    cs * C,
    csi nz )
```

Definition at line 3 of file [cs\\_scatter.c](#).



#### 4.35.3.51 cs\_scc()

```
csd * cs_scc (
    cs * A )
```

Definition at line 3 of file [cs\\_scc.c](#).

#### 4.35.3.52 cs\_schol()

```
css * cs_schol (
    csi order,
    const cs * A )
```

Definition at line 3 of file [cs\\_schol.c](#).

#### 4.35.3.53 cs\_sfree()

```
css * cs_sfree (
    css * S )
```

Definition at line 54 of file [cs\\_util.c](#).

#### 4.35.3.54 cs\_spalloc()

```
cs * cs_spalloc (
    csi m,
    csi n,
    csi nzmax,
    csi values,
    csi triplet )
```

Definition at line 3 of file [cs\\_util.c](#).

#### 4.35.3.55 cs\_spfree()

```
cs * cs_spfree (
    cs * A )
```

Definition at line 33 of file [cs\\_util.c](#).

#### 4.35.3.56 `cs_sprealloc()`

```
csi cs_sprealloc (
    cs * A,
    csi nzmax )
```

Definition at line 18 of file `cs_util.c`.

#### 4.35.3.57 `cs_spsolve()`

```
csi cs_spsolve (
    cs * G,
    const cs * B,
    csi k,
    csi * xi,
    double * x,
    const csi * pinv,
    csi lo )
```

Definition at line 3 of file `cs_spsolve.c`.

#### 4.35.3.58 `cs_sqr()`

```
css * cs_sqr (
    csi order,
    const cs * A,
    csi qr )
```

Definition at line 60 of file `cs_sqr.c`.

#### 4.35.3.59 `cs_symperm()`

```
cs * cs_symperm (
    const cs * A,
    const csi * pinv,
    csi values )
```

Definition at line 3 of file `cs_symperm.c`.

#### 4.35.3.60 cs\_tdfs()

```
csi cs_tdfs (
    csi j,
    csi k,
    csi * head,
    const csi * next,
    csi * post,
    csi * stack )
```

Definition at line 3 of file [cs\\_tdfs.c](#).

#### 4.35.3.61 cs\_transpose()

```
cs * cs_transpose (
    const cs * A,
    csi values )
```

Definition at line 3 of file [cs\\_transpose.c](#).

#### 4.35.3.62 cs\_updown()

```
csi cs_updown (
    cs * L,
    csi sigma,
    const cs * C,
    const csi * parent )
```

Definition at line 3 of file [cs\\_updown.c](#).

#### 4.35.3.63 cs\_usolve()

```
csi cs_usolve (
    const cs * U,
    double * x )
```

Definition at line 3 of file [cs\\_usolve.c](#).

#### 4.35.3.64 cs\_utsolve()

```
csi cs_utsolve (
    const cs * U,
    double * x )
```

Definition at line 3 of file [cs\\_utsolve.c](#).

## 4.36 cs.h

[Go to the documentation of this file.](#)

```

00001 #ifndef _CS_H
00002 #define _CS_H
00003 #include <stdlib.h>
00004 #include <limits.h>
00005 #include <math.h>
00006 #include <stdio.h>
00007 #include <stddef.h>
00008 #include "glbopts.h"
00009 #ifdef MATLAB_MEX_FILE
00010 #include "mex.h"
00011 #endif
00012 #define CS_VER 3 /* CSparse Version */
00013 #define CS_SUBVER 2
00014 #define CS_SUBSUB 0
00015 #define CS_DATE "Sept 12, 2017" /* CSparse release date */
00016 #define CS_COPYRIGHT "Copyright (c) Timothy A. Davis, 2006-2016"
00017
00018 // #ifdef MATLAB_MEX_FILE
00019 // #undef csi
00020 // #define csi mwSignedIndex
00021 // #endif
00022 #ifndef csi
00023 // #define csi ptrdiff_t
00024 #define csi abip_int
00025 #endif
00026
00027 /* --- primary CSparse routines and data structures ----- */
00028 typedef struct cs_sparse /* matrix in compressed-column or triplet form */
00029 {
00030     csi nzmax ; /* maximum number of entries */
00031     csi m ; /* number of rows */
00032     csi n ; /* number of columns */
00033     csi *p ; /* column pointers (size n+1) or col indices (size nzmax) */
00034     csi *i ; /* row indices, size nzmax */
00035     double *x ; /* numerical values, size nzmax */
00036     csi nz ; /* # of entries in triplet matrix, -1 for compressed-col */
00037 } cs ;
00038
00039 cs *cs_add (const cs *A, const cs *B, double alpha, double beta) ;
00040 csi cs_cholsol (csi order, const cs *A, double *b) ;
00041 cs *cs_compress (const cs *T) ;
00042 csi cs_dupl (cs *A) ;
00043 csi cs_entry (cs *T, csi i, csi j, double x) ;
00044 csi cs_gaxpy (const cs *A, const double *x, double *y) ;
00045 cs *cs_load (FILE *f) ;
00046 csi cs_lusol (csi order, const cs *A, double *b, double tol) ;
00047 cs *cs_multiply (const cs *A, const cs *B) ;
00048 double cs_norm (const cs *A) ;
00049 csi cs_print (const cs *A, csi brief) ;
00050 csi cs_qrsol (csi order, const cs *A, double *b) ;
00051 cs *cs_transpose (const cs *A, csi values) ;
00052 /* utilities */
00053 void *cs_calloc (csi n, size_t size) ;
00054 void *cs_free (void *p) ;
00055 void *cs_realloc (void *p, csi n, size_t size, csi *ok) ;
00056 cs *cs_spalloc (csi m, csi n, csi nzmax, csi values, csi triplet) ;
00057 cs *cs_spfree (cs *A) ;
00058 csi cs_sprealloc (cs *A, csi nzmax) ;
00059 void *cs_malloc (csi n, size_t size) ;
00060
00061 /* --- secondary CSparse routines and data structures ----- */
00062 typedef struct cs_symbolic /* symbolic Cholesky, LU, or QR analysis */
00063 {
00064     csi *pinv ; /* inverse row perm. for QR, fill red. perm for Chol */
00065     csi *q ; /* fill-reducing column permutation for LU and QR */
00066     csi *parent ; /* elimination tree for Cholesky and QR */
00067     csi *cp ; /* column pointers for Cholesky, row counts for QR */
00068     csi *leftmost ; /* leftmost[i] = min(find(A(i,:))), for QR */
00069     csi m2 ; /* # of rows for QR, after adding fictitious rows */
00070     double lnz ; /* # entries in L for LU or Cholesky; in V for QR */
00071     double unz ; /* # entries in U for LU; in R for QR */
00072 } css ;
00073
00074 typedef struct cs_numeric /* numeric Cholesky, LU, or QR factorization */
00075 {
00076     cs *L ; /* L for LU and Cholesky, V for QR */
00077     cs *U ; /* U for LU, R for QR, not used for Cholesky */
00078     csi *pinv ; /* partial pivoting for LU */
00079     double *B ; /* beta [0..n-1] for QR */
00080 } csn ;
00081
00082 typedef struct cs_dmperm_results /* cs_dmperm or cs_scc output */

```

```

00083 {
00084     csi *p ;           /* size m, row permutation */
00085     csi *q ;           /* size n, column permutation */
00086     csi *r ;           /* size nb+1, block k is rows r[k] to r[k+1]-1 in A(p,q) */
00087     csi *s ;           /* size nb+1, block k is cols s[k] to s[k+1]-1 in A(p,q) */
00088     csi nb ;           /* # of blocks in fine dmperm decomposition */
00089     csi rr [5] ;       /* coarse row decomposition */
00090     csi cc [5] ;       /* coarse column decomposition */
00091 } csd ;
00092
00093 csi *cs_amd (csi order, const cs *A) ;
00094 csn *cs_chol (const cs *A, const css *S) ;
00095 csd *cs_dmperm (const cs *A, csi seed) ;
00096 csi cs_droptol (cs *A, double tol) ;
00097 csi cs_dropzeros (cs *A) ;
00098 csi cs_happly (const cs *V, csi i, double beta, double *x) ;
00099 csi cs_ipvec (const csi *p, const double *b, double *x, csi n) ;
00100 csi cs_lsolve (const cs *L, double *x) ;
00101 csi cs_ltsolve (const cs *L, double *x) ;
00102 csn *cs_lu (const cs *A, const css *S, double tol) ;
00103 cs *cs_permute (const cs *A, const csi *pinv, const csi *q, csi values) ;
00104 csi *cs_pinv (const csi *p, csi n) ;
00105 csi cs_pvec (const csi *p, const double *b, double *x, csi n) ;
00106 csn *cs_qr (const cs *A, const css *S) ;
00107 css *cs_schol (csi order, const cs *A) ;
00108 css *cs_sqr (csi order, const cs *A, csi qr) ;
00109 cs *cs_syperm (const cs *A, const csi *pinv, csi values) ;
00110 csi cs_updown (cs *L, csi sigma, const cs *C, const csi *parent) ;
00111 csi cs_usolve (const cs *U, double *x) ;
00112 csi cs_utsolve (const cs *U, double *x) ;
00113 /* utilities */
00114 css *cs_sfree (css *S) ;
00115 csn *cs_nfree (csn *N) ;
00116 csd *cs_dfree (csd *D) ;
00117
00118 /* --- tertiary CSparse routines ----- */
00119 csi *cs_counts (const cs *A, const csi *parent, const csi *post, csi ata) ;
00120 double cs_cumsum (csi *p, csi *C, csi n) ;
00121 csi cs_dfs (csi j, cs *G, csi top, csi *xi, csi *pstack, const csi *pinv) ;
00122 csi cs_ereach (const cs *A, csi k, const csi *parent, csi *s, csi *w) ;
00123 csi *cs_etree (const cs *A, csi ata) ;
00124 csi cs_fkeep (cs *A, csi (*fkeep) (csi, csi, double, void *), void *other) ;
00125 double cs_house (double *x, double *beta, csi n) ;
00126 csi cs_leaf (csi i, csi j, const csi *first, csi *maxfirst, csi *prevleaf,
00127             csi *ancestor, csi *jleaf) ;
00128 csi *cs_maxtrans (const cs *A, csi seed) ;
00129 csi *cs_post (const csi *parent, csi n) ;
00130 csi *cs_randperm (csi n, csi seed) ;
00131 csi cs_reach (cs *G, const cs *B, csi k, csi *xi, const csi *pinv) ;
00132 csi cs_scatter (const cs *A, csi j, double beta, csi *w, double *x, csi mark,
00133               cs *C, csi nz) ;
00134 csd *cs_scc (cs *A) ;
00135 csi cs_spsolve (cs *G, const cs *B, csi k, csi *xi, double *x,
00136               const csi *pinv, csi lo) ;
00137 csi cs_tdfs (csi j, csi k, csi *head, const csi *next, csi *post,
00138             csi *stack) ;
00139 /* utilities */
00140 csd *cs_dalloc (csi m, csi n) ;
00141 csd *cs_ddone (csd *D, cs *C, void *w, csi ok) ;
00142 cs *cs_done (cs *C, void *w, void *x, csi ok) ;
00143 csi *cs_idone (csi *p, cs *C, void *w, csi ok) ;
00144 csn *cs_ndone (csn *N, cs *C, void *w, void *x, csi ok) ;
00145
00146 #define CS_MAX(a,b) (((a) > (b)) ? (a) : (b))
00147 #define CS_MIN(a,b) (((a) < (b)) ? (a) : (b))
00148 #define CS_FLIP(i) (~(i)-2)
00149 #define CS_UNFLIP(i) (((i) < 0) ? CS_FLIP(i) : (i))
00150 #define CS_MARKED(w,j) (w [j] < 0)
00151 #define CS_MARK(w,j) { w [j] = CS_FLIP (w [j]) ; }
00152 #define CS_CSC(A) (A && (A->nz == -1))
00153 #define CS_TRIPLET(A) (A && (A->nz >= 0))
00154 #endif

```

## 4.37 csparse/Source/cs\_add.c File Reference

```
#include "cs.h"
```

## Functions

- `cs * cs_add` (const `cs` \*A, const `cs` \*B, double alpha, double beta)

### 4.37.1 Function Documentation

#### 4.37.1.1 cs\_add()

```
cs * cs_add (
    const cs * A,
    const cs * B,
    double alpha,
    double beta )
```

Definition at line 3 of file `cs_add.c`.

## 4.38 cs\_add.c

[Go to the documentation of this file.](#)

```
00001 #include "cs.h"
00002 /* C = alpha*A + beta*B */
00003 cs *cs_add (const cs *A, const cs *B, double alpha, double beta)
00004 {
00005     csi p, j, nz = 0, anz, *Cp, *Ci, *Bp, m, n, bnz, *w, values ;
00006     double *x, *Bx, *Cx ;
00007     cs *C ;
00008     if (!CS_CSC (A) || !CS_CSC (B)) return (NULL) ;           /* check inputs */
00009     if (A->m != B->m || A->n != B->n) return (NULL) ;
00010     m = A->m ; anz = A->p [A->n] ;
00011     n = B->n ; Bp = B->p ; Bx = B->x ; bnz = Bp [n] ;
00012     w = cs_calloc (m, sizeof (csi)) ;                          /* get workspace */
00013     values = (A->x != NULL) && (Bx != NULL) ;
00014     x = values ? cs_malloc (m, sizeof (double)) : NULL ;      /* get workspace */
00015     C = cs_spalloc (m, n, anz + bnz, values, 0) ;             /* allocate result*/
00016     if (!C || !w || (values && !x)) return (cs_done (C, w, x, 0)) ;
00017     Cp = C->p ; Ci = C->i ; Cx = C->x ;
00018     for (j = 0 ; j < n ; j++)
00019     {
00020         Cp [j] = nz ;                                          /* column j of C starts here */
00021         nz = cs_scatter (A, j, alpha, w, x, j+1, C, nz) ;    /* alpha*A(:,j)*/
00022         nz = cs_scatter (B, j, beta, w, x, j+1, C, nz) ;     /* beta*B(:,j) */
00023         if (values) for (p = Cp [j] ; p < nz ; p++) Cx [p] = x [Ci [p]] ;
00024     }
00025     Cp [n] = nz ;                                              /* finalize the last column of C */
00026     cs_sprealloc (C, 0) ;                                     /* remove extra space from C */
00027     return (cs_done (C, w, x, 1)) ;                          /* success; free workspace, return C */
00028 }
```

## 4.39 csparse/Source/cs\_amd.c File Reference

```
#include "cs.h"
```

## Functions

- `csi * cs_amd` (csi order, const `cs` \*A)

## 4.39.1 Function Documentation

### 4.39.1.1 cs\_amd()

```
csi * cs_amd (
    csi order,
    const cs * A )
```

Definition at line 18 of file cs\_amd.c.

## 4.40 cs\_amd.c

[Go to the documentation of this file.](#)

```
00001 #include "cs.h"
00002 /* clear w */
00003 static csi cs_wclear (csi mark, csi lemax, csi *w, csi n)
00004 {
00005     csi k ;
00006     if (mark < 2 || (mark + lemax < 0))
00007     {
00008         for (k = 0 ; k < n ; k++) if (w [k] != 0) w [k] = 1 ;
00009         mark = 2 ;
00010     }
00011     return (mark) ;      /* at this point, w [0..n-1] < mark holds */
00012 }
00013
00014 /* keep off-diagonal entries; drop diagonal entries */
00015 static csi cs_diag (csi i, csi j, double aij, void *other) { return (i != j) ; }
00016
00017 /* p = amd(A+A') if symmetric is true, or amd(A'A) otherwise */
00018 csi *cs_amd (csi order, const cs *A) /* order 0:natural, 1:Chol, 2:LU, 3:QR */
00019 {
00020     cs *C, *A2, *AT ;
00021     csi *Cp, *Ci, *last, *W, *len, *nv, *next, *P, *head, *elen, *degree, *w,
00022         *hhead, *ATp, *ATi, d, dk, dext, lemax = 0, e, elenk, eln, i, j, k, k1,
00023         k2, k3, jlast, ln, dense, nzmax, mindeg = 0, nvi, nvj, nvk, mark, wnvi,
00024         ok, cnz, nel = 0, p, p1, p2, p3, p4, pj, pk, pk1, pk2, pn, q, n, m, t ;
00025     csi h ;
00026     /* --- Construct matrix C ----- */
00027     if (!CS_CSC (A) || order <= 0 || order > 3) return (NULL) ; /* check */
00028     AT = cs_transpose (A, 0) ; /* compute A' */
00029     if (!AT) return (NULL) ;
00030     m = A->m ; n = A->n ;
00031     dense = CS_MAX (16, 10 * sqrt ((double) n)) ; /* find dense threshold */
00032     dense = CS_MIN (n-2, dense) ;
00033     if (order == 1 && n == m)
00034     {
00035         C = cs_add (A, AT, 0, 0) ; /* C = A+A' */
00036     }
00037     else if (order == 2)
00038     {
00039         ATp = AT->p ; /* drop dense columns from AT */
00040         ATi = AT->i ;
00041         for (p2 = 0, j = 0 ; j < m ; j++)
00042         {
00043             p = ATp [j] ; /* column j of AT starts here */
00044             ATp [j] = p2 ; /* new column j starts here */
00045             if (ATp [j+1] - p > dense) continue ; /* skip dense col j */
00046             for ( ; p < ATp [j+1] ; p++) ATi [p2++] = ATi [p] ;
00047         }
00048         ATp [m] = p2 ; /* finalize AT */
00049         A2 = cs_transpose (AT, 0) ; /* A2 = AT' */
00050         C = A2 ? cs_multiply (AT, A2) : NULL ; /* C=A'*A with no dense rows */
00051         cs_spfree (A2) ;
00052     }
00053     else
00054     {
00055         C = cs_multiply (AT, A) ; /* C=A'*A */
00056     }
00057     cs_spfree (AT) ;
00058     if (!C) return (NULL) ;
```

```

00059     cs_fkeep (C, &cs_diag, NULL) ;           /* drop diagonal entries */
00060     Cp = C->p ;
00061     cnz = Cp [n] ;
00062     P = cs_malloc (n+1, sizeof (csi)) ;      /* allocate result */
00063     W = cs_malloc (8*(n+1), sizeof (csi)) ;  /* get workspace */
00064     t = cnz + cnz/5 + 2*n ;                  /* add elbow room to C */
00065     if (!P || !W || !cs_sprealloc (C, t)) return (cs_idone (P, C, W, 0)) ;
00066     len = W ; nv = W + (n+1) ; next = W + 2*(n+1) ;
00067     head = W + 3*(n+1) ; elen = W + 4*(n+1) ; degree = W + 5*(n+1) ;
00068     w = W + 6*(n+1) ; hhead = W + 7*(n+1) ;
00069     last = P ;                               /* use P as workspace for last */
00070     /* --- Initialize quotient graph ----- */
00071     for (k = 0 ; k < n ; k++) len [k] = Cp [k+1] - Cp [k] ;
00072     len [n] = 0 ;
00073     nzmax = C->nzmax ;
00074     Ci = C->i ;
00075     for (i = 0 ; i <= n ; i++)
00076     {
00077         head [i] = -1 ;                      /* degree list i is empty */
00078         last [i] = -1 ;
00079         next [i] = -1 ;
00080         hhead [i] = -1 ;                    /* hash list i is empty */
00081         nv [i] = 1 ;                        /* node i is just one node */
00082         w [i] = 1 ;                         /* node i is alive */
00083         elen [i] = 0 ;                      /* Ek of node i is empty */
00084         degree [i] = len [i] ;              /* degree of node i */
00085     }
00086     mark = cs_wclear (0, 0, w, n) ;          /* clear w */
00087     elen [n] = -2 ;                          /* n is a dead element */
00088     Cp [n] = -1 ;                            /* n is a root of assembly tree */
00089     w [n] = 0 ;                              /* n is a dead element */
00090     /* --- Initialize degree lists ----- */
00091     for (i = 0 ; i < n ; i++)
00092     {
00093         d = degree [i] ;
00094         if (d == 0)                          /* node i is empty */
00095         {
00096             elen [i] = -2 ;                  /* element i is dead */
00097             nel++ ;
00098             Cp [i] = -1 ;                    /* i is a root of assembly tree */
00099             w [i] = 0 ;
00100         }
00101         else if (d > dense)                   /* node i is dense */
00102         {
00103             nv [i] = 0 ;                     /* absorb i into element n */
00104             elen [i] = -1 ;                  /* node i is dead */
00105             nel++ ;
00106             Cp [i] = CS_FLIP (n) ;
00107             nv [n]++ ;
00108         }
00109         else
00110         {
00111             if (head [d] != -1) last [head [d]] = i ;
00112             next [i] = head [d] ;            /* put node i in degree list d */
00113             head [d] = i ;
00114         }
00115     }
00116     while (nel < n)                          /* while (selecting pivots) do */
00117     {
00118         /* --- Select node of minimum approximate degree ----- */
00119         for (k = -1 ; mindeg < n && (k = head [mindeg]) == -1 ; mindeg++) ;
00120         if (next [k] != -1) last [next [k]] = -1 ;
00121         head [mindeg] = next [k] ;           /* remove k from degree list */
00122         elenk = elen [k] ;                   /* elenk = |Ek| */
00123         nvk = nv [k] ;                       /* # of nodes k represents */
00124         nel += nvk ;                         /* nv[k] nodes of A eliminated */
00125         /* --- Garbage collection ----- */
00126         if (elenk > 0 && cnz + mindeg >= nzmax)
00127         {
00128             for (j = 0 ; j < n ; j++)
00129             {
00130                 if ((p = Cp [j]) >= 0)      /* j is a live node or element */
00131                 {
00132                     Cp [j] = Ci [p] ;        /* save first entry of object */
00133                     Ci [p] = CS_FLIP (j) ;    /* first entry is now CS_FLIP(j) */
00134                 }
00135             }
00136             for (q = 0, p = 0 ; p < cnz ; ) /* scan all of memory */
00137             {
00138                 if ((j = CS_FLIP (Ci [p++])) >= 0) /* found object j */
00139                 {
00140                     Ci [q] = Cp [j] ;        /* restore first entry of object */
00141                     Cp [j] = q++ ;           /* new pointer to object j */
00142                     for (k3 = 0 ; k3 < len [j]-1 ; k3++) Ci [q++] = Ci [p++] ;
00143                 }
00144             }
00145             cnz = q ;                        /* Ci [cnz...nzmax-1] now free */

```



```

00146     }
00147     /* --- Construct new element ----- */
00148     dk = 0 ;
00149     nv [k] = -nvk ;                /* flag k as in Lk */
00150     p = Cp [k] ;
00151     pk1 = (elenk == 0) ? p : cnz ; /* do in place if elen[k] == 0 */
00152     pk2 = pk1 ;
00153     for (k1 = 1 ; k1 <= elenk + 1 ; k1++)
00154     {
00155         if (k1 > elenk)
00156         {
00157             e = k ;                /* search the nodes in k */
00158             pj = p ;                /* list of nodes starts at Ci[pj]*/
00159             ln = len [k] - elenk ; /* length of list of nodes in k */
00160         }
00161         else
00162         {
00163             e = Ci [p++] ;          /* search the nodes in e */
00164             pj = Cp [e] ;
00165             ln = len [e] ;          /* length of list of nodes in e */
00166         }
00167         for (k2 = 1 ; k2 <= ln ; k2++)
00168         {
00169             i = Ci [pj++] ;
00170             if ((nvi = nv [i]) <= 0) continue ; /* node i dead, or seen */
00171             dk += nvi ;              /* degree[Lk] += size of node i */
00172             nv [i] = -nvi ;          /* negate nv[i] to denote i in Lk*/
00173             Ci [pk2++] = i ;         /* place i in Lk */
00174             if (next [i] != -1) last [next [i]] = last [i] ;
00175             if (last [i] != -1)      /* remove i from degree list */
00176             {
00177                 next [last [i]] = next [i] ;
00178             }
00179             else
00180             {
00181                 head [degree [i]] = next [i] ;
00182             }
00183         }
00184         if (e != k)
00185         {
00186             Cp [e] = CS_FLIP (k) ; /* absorb e into k */
00187             w [e] = 0 ;            /* e is now a dead element */
00188         }
00189     }
00190     if (elenk != 0) cnz = pk2 ;     /* Ci [cnz...nzmax] is free */
00191     degree [k] = dk ;              /* external degree of k - |Lk|i */
00192     Cp [k] = pk1 ;                 /* element k is in Ci[pk1..pk2-1] */
00193     len [k] = pk2 - pk1 ;
00194     elen [k] = -2 ;                /* k is now an element */
00195     /* --- Find set differences ----- */
00196     mark = cs_wclear (mark, lemax, w, n) ; /* clear w if necessary */
00197     for (pk = pk1 ; pk < pk2 ; pk++) /* scan 1: find |Le\Lk| */
00198     {
00199         i = Ci [pk] ;
00200         if ((eln = elen [i]) <= 0) continue ; /* skip if elen[i] empty */
00201         nvi = -nv [i] ;             /* nv [i] was negated */
00202         wnvi = mark - nvi ;
00203         for (p = Cp [i] ; p <= Cp [i] + eln - 1 ; p++) /* scan Ei */
00204         {
00205             e = Ci [p] ;
00206             if (w [e] >= mark)
00207             {
00208                 w [e] -= nvi ;      /* decrement |Le\Lk| */
00209             }
00210             else if (w [e] != 0)    /* ensure e is a live element */
00211             {
00212                 w [e] = degree [e] + wnvi ; /* 1st time e seen in scan 1 */
00213             }
00214         }
00215     }
00216     /* --- Degree update ----- */
00217     for (pk = pk1 ; pk < pk2 ; pk++) /* scan2: degree update */
00218     {
00219         i = Ci [pk] ;              /* consider node i in Lk */
00220         p1 = Cp [i] ;
00221         p2 = p1 + elen [i] - 1 ;
00222         pn = p1 ;
00223         for (h = 0, d = 0, p = p1 ; p <= p2 ; p++) /* scan Ei */
00224         {
00225             e = Ci [p] ;
00226             if (w [e] != 0)         /* e is an unabsorbed element */
00227             {
00228                 dext = w [e] - mark ; /* dext = |Le\Lk| */
00229                 if (dext > 0)
00230                 {
00231                     d += dext ;      /* sum up the set differences */
00232                     Ci [pn++] = e ;  /* keep e in Ei */
00233                 }
00234             }
00235         }
00236     }

```

```

00233         h += e ;           /* compute the hash of node i */
00234     }
00235     else
00236     {
00237         Cp [e] = CS_FLIP (k) ; /* aggressive absorb. e->k */
00238         w [e] = 0 ;           /* e is a dead element */
00239     }
00240 }
00241
00242 elen [i] = pn - p1 + 1 ;     /* elen[i] = |Ei| */
00243 p3 = pn ;
00244 p4 = p1 + len [i] ;
00245 for (p = p2 + 1 ; p < p4 ; p++) /* prune edges in Ai */
00246 {
00247     j = Ci [p] ;
00248     if ((nvj = nv [j]) <= 0) continue ; /* node j dead or in Lk */
00249     d += nvj ;                 /* degree(i) += |j| */
00250     Ci [pn++] = j ;           /* place j in node list of i */
00251     h += j ;                 /* compute hash for node i */
00252 }
00253 if (d == 0)                   /* check for mass elimination */
00254 {
00255     Cp [i] = CS_FLIP (k) ;    /* absorb i into k */
00256     nvi = -nv [i] ;
00257     dk -= nvi ;               /* |Lk| -= |i| */
00258     nvk += nvi ;              /* |k| += nv[i] */
00259     nel += nvi ;
00260     nv [i] = 0 ;
00261     elen [i] = -1 ;           /* node i is dead */
00262 }
00263 else
00264 {
00265     degree [i] = CS_MIN (degree [i], d) ; /* update degree(i) */
00266     Ci [pn] = Ci [p3] ;       /* move first node to end */
00267     Ci [p3] = Ci [p1] ;       /* move 1st el. to end of Ei */
00268     Ci [p1] = k ;             /* add k as 1st element in of Ei */
00269     len [i] = pn - p1 + 1 ;    /* new len of adj. list of node i */
00270     h = ((h<0) ? (-h):h) % n ; /* finalize hash of i */
00271     next [i] = hhead [h] ;     /* place i in hash bucket */
00272     hhead [h] = i ;
00273     last [i] = h ;             /* save hash of i in last[i] */
00274 }
00275 } /* scan2 is done */
00276 degree [k] = dk ;             /* finalize |Lk| */
00277 lemax = CS_MAX (lemax, dk) ;
00278 mark = cs_wclear (mark+lemax, lemax, w, n) ; /* clear w */
00279 /* --- Supernode detection ----- */
00280 for (pk = pk1 ; pk < pk2 ; pk++)
00281 {
00282     i = Ci [pk] ;
00283     if (nv [i] >= 0) continue ; /* skip if i is dead */
00284     h = last [i] ;             /* scan hash bucket of node i */
00285     i = hhead [h] ;
00286     hhead [h] = -1 ;           /* hash bucket will be empty */
00287     for ( ; i != -1 && next [i] != -1 ; i = next [i], mark++)
00288     {
00289         ln = len [i] ;
00290         eln = elen [i] ;
00291         for (p = Cp [i]+1 ; p <= Cp [i] + ln-1 ; p++) w [Ci [p]] = mark;
00292         jlast = i ;
00293         for (j = next [i] ; j != -1 ; ) /* compare i with all j */
00294         {
00295             ok = (len [j] == ln) && (elen [j] == eln) ;
00296             for (p = Cp [j] + 1 ; ok && p <= Cp [j] + ln - 1 ; p++)
00297             {
00298                 if (w [Ci [p]] != mark) ok = 0 ; /* compare i and j */
00299             }
00300             if (ok) /* i and j are identical */
00301             {
00302                 Cp [j] = CS_FLIP (i) ; /* absorb j into i */
00303                 nv [i] += nv [j] ;
00304                 nv [j] = 0 ;
00305                 elen [j] = -1 ; /* node j is dead */
00306                 j = next [j] ; /* delete j from hash bucket */
00307                 next [jlast] = j ;
00308             }
00309             else
00310             {
00311                 jlast = j ; /* j and i are different */
00312                 j = next [j] ;
00313             }
00314         }
00315     }
00316 }
00317 /* --- Finalize new element----- */
00318 for (p = pk1, pk = pk1 ; pk < pk2 ; pk++) /* finalize Lk */
00319 {

```

```

00320         i = Ci [pk] ;
00321         if ((nvi = -nv [i]) <= 0) continue ; /* skip if i is dead */
00322         nv [i] = nvi ; /* restore nv[i] */
00323         d = degree [i] + dk - nvi ; /* compute external degree(i) */
00324         d = CS_MIN (d, n - nel - nvi) ;
00325         if (head [d] != -1) last [head [d]] = i ;
00326         next [i] = head [d] ; /* put i back in degree list */
00327         last [i] = -1 ;
00328         head [d] = i ;
00329         mindeg = CS_MIN (mindeg, d) ; /* find new minimum degree */
00330         degree [i] = d ;
00331         Ci [p++] = i ; /* place i in Lk */
00332     }
00333     nv [k] = nvk ; /* # nodes absorbed into k */
00334     if ((len [k] = p-pk1) == 0) /* length of adj list of element k */
00335     {
00336         Cp [k] = -1 ; /* k is a root of the tree */
00337         w [k] = 0 ; /* k is now a dead element */
00338     }
00339     if (elenk != 0) cnz = p ; /* free unused space in Lk */
00340 }
00341 /* --- Postordering ----- */
00342 for (i = 0 ; i < n ; i++) Cp [i] = CS_FLIP (Cp [i]) ; /* fix assembly tree */
00343 for (j = 0 ; j <= n ; j++) head [j] = -1 ;
00344 for (j = n ; j >= 0 ; j--) /* place unordered nodes in lists */
00345 {
00346     if (nv [j] > 0) continue ; /* skip if j is an element */
00347     next [j] = head [Cp [j]] ; /* place j in list of its parent */
00348     head [Cp [j]] = j ;
00349 }
00350 for (e = n ; e >= 0 ; e--) /* place elements in lists */
00351 {
00352     if (nv [e] <= 0) continue ; /* skip unless e is an element */
00353     if (Cp [e] != -1)
00354     {
00355         next [e] = head [Cp [e]] ; /* place e in list of its parent */
00356         head [Cp [e]] = e ;
00357     }
00358 }
00359 for (k = 0, i = 0 ; i <= n ; i++) /* postorder the assembly tree */
00360 {
00361     if (Cp [i] == -1) k = cs_tdfs (i, k, head, next, P, w) ;
00362 }
00363 return (cs_idone (P, C, W, 1)) ;
00364 }

```

## 4.41 csparse/Source/cs\_chol.c File Reference

```
#include "cs.h"
```

### Functions

- [csn \\* cs\\_chol](#) (const [cs](#) \*A, const [css](#) \*S)

#### 4.41.1 Function Documentation

##### 4.41.1.1 cs\_chol()

```

csn * cs_chol (
    const cs * A,
    const css * S )

```

Definition at line 3 of file [cs\\_chol.c](#).

## 4.42 cs\_chol.c

[Go to the documentation of this file.](#)

```

00001 #include "cs.h"
00002 /* L = chol (A, [pinv parent cp]), pinv is optional */
00003 csn *cs_chol (const cs *A, const css *S)
00004 {
00005     double d, lki, *Lx, *x, *Cx ;
00006     csi top, i, p, k, n, *Li, *Lp, *cp, *pinv, *s, *c, *parent, *Cp, *Ci ;
00007     cs *L, *C, *E ;
00008     csn *N ;
00009     if (!CS_CSC (A) || !S || !S->cp || !S->parent) return (NULL) ;
00010     n = A->n ;
00011     N = cs_calloc (1, sizeof (csn)) ; /* allocate result */
00012     c = cs_malloc (2*n, sizeof (csi)) ; /* get csi workspace */
00013     x = cs_malloc (n, sizeof (double)) ; /* get double workspace */
00014     cp = S->cp ; pinv = S->pinv ; parent = S->parent ;
00015     C = pinv ? cs_symperm (A, pinv, 1) : ((cs *) A) ;
00016     E = pinv ? C : NULL ; /* E is alias for A, or a copy E=A(p,p) */
00017     if (!N || !c || !x || !C) return (cs_ndone (N, E, c, x, 0)) ;
00018     s = c + n ;
00019     Cp = C->p ; Ci = C->i ; Cx = C->x ;
00020     N->L = L = cs_spalloc (n, n, cp [n], 1, 0) ; /* allocate result */
00021     if (!L) return (cs_ndone (N, E, c, x, 0)) ;
00022     Lp = L->p ; Li = L->i ; Lx = L->x ;
00023     for (k = 0 ; k < n ; k++) Lp [k] = c [k] = cp [k] ;
00024     for (k = 0 ; k < n ; k++) /* compute L(k,:) for L*L' = C */
00025     {
00026         /* --- Nonzero pattern of L(k,:) ----- */
00027         top = cs_ereach (C, k, parent, s, c) ; /* find pattern of L(k,:) */
00028         x [k] = 0 ; /* x (0:k) is now zero */
00029         for (p = Cp [k] ; p < Cp [k+1] ; p++) /* x = full(triu(C(:,k))) */
00030         {
00031             if (Ci [p] <= k) x [Ci [p]] = Cx [p] ;
00032         }
00033         d = x [k] ; /* d = C(k,k) */
00034         x [k] = 0 ; /* clear x for k+1st iteration */
00035         /* --- Triangular solve ----- */
00036         for ( ; top < n ; top++) /* solve L(0:k-1,0:k-1) * x = C(:,k) */
00037         {
00038             i = s [top] ; /* s [top..n-1] is pattern of L(k,:) */
00039             lki = x [i] / Lx [Lp [i]] ; /* L(k,i) = x (i) / L(i,i) */
00040             x [i] = 0 ; /* clear x for k+1st iteration */
00041             for (p = Lp [i] + 1 ; p < c [i] ; p++)
00042             {
00043                 x [Li [p]] -= Lx [p] * lki ;
00044             }
00045             d -= lki * lki ; /* d = d - L(k,i)*L(k,i) */
00046             p = c [i]++ ;
00047             Li [p] = k ; /* store L(k,i) in column i */
00048             Lx [p] = lki ;
00049         }
00050         /* --- Compute L(k,k) ----- */
00051         if (d <= 0) return (cs_ndone (N, E, c, x, 0)) ; /* not pos def */
00052         p = c [k]++ ;
00053         Li [p] = k ; /* store L(k,k) = sqrt (d) in column k */
00054         Lx [p] = sqrt (d) ;
00055     }
00056     Lp [n] = cp [n] ; /* finalize L */
00057     return (cs_ndone (N, E, c, x, 1)) ; /* success: free E,s,x; return N */
00058 }

```

## 4.43 csparse/Source/cs\_cholsol.c File Reference

```
#include "cs.h"
```

### Functions

- `csi cs_cholsol (csi order, const cs *A, double *b)`

## 4.43.1 Function Documentation

### 4.43.1.1 cs\_cholsol()

```
csi cs_cholsol (
    csi order,
    const cs * A,
    double * b )
```

Definition at line 3 of file [cs\\_cholsol.c](#).

## 4.44 cs\_cholsol.c

[Go to the documentation of this file.](#)

```
00001 #include "cs.h"
00002 /* x=A\b where A is symmetric positive definite; b overwritten with solution */
00003 csi cs_cholsol (csi order, const cs *A, double *b)
00004 {
00005     double *x ;
00006     css *S ;
00007     csn *N ;
00008     csi n, ok ;
00009     if (!CS_CSC (A) || !b) return (0) ;      /* check inputs */
00010     n = A->n ;
00011     S = cs_schol (order, A) ;                 /* ordering and symbolic analysis */
00012     N = cs_chol (A, S) ;                     /* numeric Cholesky factorization */
00013     x = cs_malloc (n, sizeof (double)) ;     /* get workspace */
00014     ok = (S && N && x) ;
00015     if (ok)
00016     {
00017         cs_ipvec (S->pinv, b, x, n) ;        /* x = P*b */
00018         cs_lsolve (N->L, x) ;                 /* x = L\x */
00019         cs_ltsolve (N->L, x) ;                /* x = L'\x */
00020         cs_pvec (S->pinv, x, b, n) ;          /* b = P'*x */
00021     }
00022     cs_free (x) ;
00023     cs_sfree (S) ;
00024     cs_nfree (N) ;
00025     return (ok) ;
00026 }
```

## 4.45 csparse/Source/cs\_compress.c File Reference

```
#include "cs.h"
```

### Functions

- `cs * cs_compress (const cs *T)`

### 4.45.1 Function Documentation

#### 4.45.1.1 cs\_compress()

```
cs * cs_compress (
    const cs * T )
```

Definition at line 3 of file [cs\\_compress.c](#).

### 4.46 cs\_compress.c

[Go to the documentation of this file.](#)

```
00001 #include "cs.h"
00002 /* C = compressed-column form of a triplet matrix T */
00003 cs *cs_compress (const cs *T)
00004 {
00005     csi m, n, nz, p, k, *Cp, *Ci, *w, *Ti, *Tj ;
00006     double *Cx, *Tx ;
00007     cs *C ;
00008     if (!CS_TRIPLET (T)) return (NULL) ; /* check inputs */
00009     m = T->m ; n = T->n ; Ti = T->i ; Tj = T->p ; Tx = T->x ; nz = T->nz ;
00010     C = cs_spalloc (m, n, nz, Tx != NULL, 0) ; /* allocate result */
00011     w = cs_calloc (n, sizeof (csi)) ; /* get workspace */
00012     if (!C || !w) return (cs_done (C, w, NULL, 0)) ; /* out of memory */
00013     Cp = C->p ; Ci = C->i ; Cx = C->x ;
00014     for (k = 0 ; k < nz ; k++) w [Tj [k]]++ ; /* column counts */
00015     cs_cumsum (Cp, w, n) ; /* column pointers */
00016     for (k = 0 ; k < nz ; k++)
00017     {
00018         Ci [p = w [Tj [k]]++] = Ti [k] ; /* A(i,j) is the pth entry in C */
00019         if (Cx) Cx [p] = Tx [k] ;
00020     }
00021     return (cs_done (C, w, NULL, 1)) ; /* success; free w and return C */
00022 }
```

### 4.47 csparse/Source/cs\_counts.c File Reference

```
#include "cs.h"
```

#### Macros

- #define [HEAD](#)(k, j) (ata ? head [k] : j)
- #define [NEXT](#)(J) (ata ? next [J] : -1)

#### Functions

- [csi](#) \* [cs\\_counts](#) (const [cs](#) \*A, const [csi](#) \*parent, const [csi](#) \*post, [csi](#) ata)

#### 4.47.1 Macro Definition Documentation

## 4.47.1.1 HEAD

```
#define HEAD(
    k,
    j ) (ata ? head [k] : j)
```

Definition at line 3 of file [cs\\_counts.c](#).

## 4.47.1.2 NEXT

```
#define NEXT(
    J ) (ata ? next [J] : -1)
```

Definition at line 4 of file [cs\\_counts.c](#).

## 4.47.2 Function Documentation

## 4.47.2.1 cs\_counts()

```
csi * cs_counts (
    const cs * A,
    const csi * parent,
    const csi * post,
    csi ata )
```

Definition at line 17 of file [cs\\_counts.c](#).

## 4.48 cs\_counts.c

[Go to the documentation of this file.](#)

```
00001 #include "cs.h"
00002 /* column counts of LL'=A or LL'=A'A, given parent & post ordering */
00003 #define HEAD(k,j) (ata ? head [k] : j)
00004 #define NEXT(J) (ata ? next [J] : -1)
00005 static void init_ata (cs *AT, const csi *post, csi *w, csi **head, csi **next)
00006 {
00007     csi i, k, p, m = AT->n, n = AT->m, *ATp = AT->p, *ATi = AT->i ;
00008     *head = w+4*n, *next = w+5*n+1 ;
00009     for (k = 0 ; k < n ; k++) w [post [k]] = k ; /* invert post */
00010     for (i = 0 ; i < m ; i++)
00011     {
00012         for (k = n, p = ATp[i] ; p < ATp[i+1] ; p++) k = CS_MIN (k, w [ATi[p]]);
00013         (*next) [i] = (*head) [k] ; /* place row i in linked list k */
00014         (*head) [k] = i ;
00015     }
00016 }
00017 csi *cs_counts (const cs *A, const csi *parent, const csi *post, csi ata)
00018 {
00019     csi i, j, k, n, m, J, s, p, q, jleaf, *ATp, *ATi, *maxfirst, *prevleaf,
00020     *ancestor, *head = NULL, *next = NULL, *colcount, *w, *first, *delta ;
00021     cs *AT ;
00022     if (!CS_CSC (A) || !parent || !post) return (NULL) ; /* check inputs */
00023     m = A->m ; n = A->n ;
00024     s = 4*n + (ata ? (n+m+1) : 0) ;
```

```

00025     delta = colcount = cs_malloc (n, sizeof (csi)) ;      /* allocate result */
00026     w = cs_malloc (s, sizeof (csi)) ;                      /* get workspace */
00027     AT = cs_transpose (A, 0) ;                              /* AT = A' */
00028     if (!AT || !colcount || !w) return (cs_idone (colcount, AT, w, 0)) ;
00029     ancestor = w ; maxfirst = w+n ; prevleaf = w+2*n ; first = w+3*n ;
00030     for (k = 0 ; k < s ; k++) w [k] = -1 ;                /* clear workspace w [0..s-1] */
00031     for (k = 0 ; k < n ; k++)                              /* find first [j] */
00032     {
00033         j = post [k] ;
00034         delta [j] = (first [j] == -1) ? 1 : 0 ; /* delta[j]=1 if j is a leaf */
00035         for ( ; j != -1 && first [j] == -1 ; j = parent [j]) first [j] = k ;
00036     }
00037     ATp = AT->p ; ATi = AT->i ;
00038     if (ata) init_ata (AT, post, w, &head, &next) ;
00039     for (i = 0 ; i < n ; i++) ancestor [i] = i ; /* each node in its own set */
00040     for (k = 0 ; k < n ; k++)
00041     {
00042         j = post [k] ; /* j is the kth node in postordered etree */
00043         if (parent [j] != -1) delta [parent [j]]-- ; /* j is not a root */
00044         for (J = HEAD (k, j) ; J != -1 ; J = NEXT (J)) /* J=j for LL'=A case */
00045         {
00046             for (p = ATp [J] ; p < ATp [J+1] ; p++)
00047             {
00048                 i = ATi [p] ;
00049                 q = cs_leaf (i, j, first, maxfirst, prevleaf, ancestor, &jleaf);
00050                 if (jleaf >= 1) delta [j]++ ; /* A(i,j) is in skeleton */
00051                 if (jleaf == 2) delta [q]-- ; /* account for overlap in q */
00052             }
00053         }
00054         if (parent [j] != -1) ancestor [j] = parent [j] ;
00055     }
00056     for (j = 0 ; j < n ; j++) /* sum up delta's of each child */
00057     {
00058         if (parent [j] != -1) colcount [parent [j]] += colcount [j] ;
00059     }
00060     return (cs_idone (colcount, AT, w, 1)) ; /* success: free workspace */
00061 }

```

## 4.49 csparse/Source/cs\_cumsum.c File Reference

```
#include "cs.h"
```

### Functions

- double `cs_cumsum` (`csi` \*p, `csi` \*c, `csi` n)

#### 4.49.1 Function Documentation

##### 4.49.1.1 cs\_cumsum()

```
double cs_cumsum (
    csi * p,
    csi * c,
    csi n )
```

Definition at line 3 of file `cs_cumsum.c`.



## 4.50 cs\_cumsum.c

[Go to the documentation of this file.](#)

```
00001 #include "cs.h"
00002 /* p [0..n] = cumulative sum of c [0..n-1], and then copy p [0..n-1] into c */
00003 double cs_cumsum (csi *p, csi *c, csi n)
00004 {
00005     csi i, nz = 0 ;
00006     double nz2 = 0 ;
00007     if (!p || !c) return (-1) ;      /* check inputs */
00008     for (i = 0 ; i < n ; i++)
00009     {
00010         p [i] = nz ;
00011         nz += c [i] ;
00012         nz2 += c [i] ;                /* also in double to avoid csi overflow */
00013         c [i] = p [i] ;              /* also copy p[0..n-1] back into c[0..n-1] */
00014     }
00015     p [n] = nz ;
00016     return (nz2) ;                  /* return sum (c [0..n-1]) */
00017 }
```

## 4.51 csparse/Source/cs\_dfs.c File Reference

```
#include "cs.h"
```

### Functions

- [csi cs\\_dfs](#) (csi j, cs \*G, csi top, csi \*xi, csi \*pstack, const csi \*pinv)

### 4.51.1 Function Documentation

#### 4.51.1.1 cs\_dfs()

```
csi cs_dfs (
    csi j,
    cs * G,
    csi top,
    csi * xi,
    csi * pstack,
    const csi * pinv )
```

Definition at line 3 of file [cs\\_dfs.c](#).

## 4.52 cs\_dfs.c

[Go to the documentation of this file.](#)

```

00001 #include "cs.h"
00002 /* depth-first-search of the graph of a matrix, starting at node j */
00003 csi cs_dfs (csi j, cs *G, csi top, csi *xi, csi *pstack, const csi *pinv)
00004 {
00005     csi i, p, p2, done, jnew, head = 0, *Gp, *Gi ;
00006     if (!CS_CSC (G) || !xi || !pstack) return (-1) ; /* check inputs */
00007     Gp = G->p ; Gi = G->i ;
00008     xi [0] = j ; /* initialize the recursion stack */
00009     while (head >= 0)
00010     {
00011         j = xi [head] ; /* get j from the top of the recursion stack */
00012         jnew = pinv ? (pinv [j]) : j ;
00013         if (!CS_MARKED (Gp, j))
00014         {
00015             CS_MARK (Gp, j) ; /* mark node j as visited */
00016             pstack [head] = (jnew < 0) ? 0 : CS_UNFLIP (Gp [jnew]) ;
00017         }
00018         done = 1 ; /* node j done if no unvisited neighbors */
00019         p2 = (jnew < 0) ? 0 : CS_UNFLIP (Gp [jnew+1]) ;
00020         for (p = pstack [head] ; p < p2 ; p++) /* examine all neighbors of j */
00021         {
00022             i = Gi [p] ; /* consider neighbor node i */
00023             if (CS_MARKED (Gp, i)) continue ; /* skip visited node i */
00024             pstack [head] = p ; /* pause depth-first search of node j */
00025             xi [++head] = i ; /* start dfs at node i */
00026             done = 0 ; /* node j is not done */
00027             break ; /* break, to start dfs (i) */
00028         }
00029         if (done) /* depth-first search at node j is done */
00030         {
00031             head-- ; /* remove j from the recursion stack */
00032             xi [--top] = j ; /* and place in the output stack */
00033         }
00034     }
00035     return (top) ;
00036 }

```

## 4.53 csparse/Source/cs\_dmperm.c File Reference

```
#include "cs.h"
```

### Functions

- `csd * cs_dmperm (const cs *A, csi seed)`

#### 4.53.1 Function Documentation

##### 4.53.1.1 cs\_dmperm()

```

csd * cs_dmperm (
    const cs * A,
    csi seed )

```

Definition at line 68 of file `cs_dmperm.c`.

## 4.54 cs\_dmperm.c

[Go to the documentation of this file.](#)

```

00001 #include "cs.h"
00002 /* breadth-first search for coarse decomposition (C0,C1,R1 or R0,R3,C3) */
00003 static csi cs_bfs (const cs *A, csi n, csi *wi, csi *wj, csi *queue,
00004     const csi *imatch, const csi *jmatch, csi mark)
00005 {
00006     csi *Ap, *Ai, head = 0, tail = 0, j, i, p, j2 ;
00007     cs *C ;
00008     for (j = 0 ; j < n ; j++)          /* place all unmatched nodes in queue */
00009     {
00010         if (imatch [j] >= 0) continue ; /* skip j if matched */
00011         wj [j] = 0 ;                   /* j in set C0 (R0 if transpose) */
00012         queue [tail++] = j ;           /* place unmatched col j in queue */
00013     }
00014     if (tail == 0) return (1) ;         /* quick return if no unmatched nodes */
00015     C = (mark == 1) ? ((cs *) A) : cs_transpose (A, 0) ;
00016     if (!C) return (0) ;               /* bfs of C=A' to find R3,C3 from R0 */
00017     Ap = C->p ; Ai = C->i ;
00018     while (head < tail)                /* while queue is not empty */
00019     {
00020         j = queue [head++] ;           /* get the head of the queue */
00021         for (p = Ap [j] ; p < Ap [j+1] ; p++)
00022         {
00023             i = Ai [p] ;
00024             if (wi [i] >= 0) continue ; /* skip if i is marked */
00025             wi [i] = mark ;            /* i in set R1 (C3 if transpose) */
00026             j2 = jmatch [i] ;          /* traverse alternating path to j2 */
00027             if (wj [j2] >= 0) continue ; /* skip j2 if it is marked */
00028             wj [j2] = mark ;          /* j2 in set C1 (R3 if transpose) */
00029             queue [tail++] = j2 ;      /* add j2 to queue */
00030         }
00031     }
00032     if (mark != 1) cs_sfree (C) ;      /* free A' if it was created */
00033     return (1) ;
00034 }
00035
00036 /* collect matched rows and columns into p and q */
00037 static void cs_matched (csi n, const csi *wj, const csi *imatch, csi *p, csi *q,
00038     csi *cc, csi *rr, csi set, csi mark)
00039 {
00040     csi kc = cc [set], j ;
00041     csi kr = rr [set-1] ;
00042     for (j = 0 ; j < n ; j++)
00043     {
00044         if (wj [j] != mark) continue ; /* skip if j is not in C set */
00045         p [kr++] = imatch [j] ;
00046         q [kc++] = j ;
00047     }
00048     cc [set+1] = kc ;
00049     rr [set] = kr ;
00050 }
00051
00052 /* collect unmatched rows into the permutation vector p */
00053 static void cs_unmatched (csi m, const csi *wi, csi *p, csi *rr, csi set)
00054 {
00055     csi i, kr = rr [set] ;
00056     for (i = 0 ; i < m ; i++) if (wi [i] == 0) p [kr++] = i ;
00057     rr [set+1] = kr ;
00058 }
00059
00060 /* return 1 if row i is in R2 */
00061 static csi cs_rprune (csi i, csi j, double aij, void *other)
00062 {
00063     csi *rr = (csi *) other ;
00064     return (i >= rr [1] && i < rr [2]) ;
00065 }
00066
00067 /* Given A, compute coarse and then fine dmperm */
00068 csd *cs_dmperm (const cs *A, csi seed)
00069 {
00070     csi m, n, i, j, k, cnz, nc, *jmatch, *imatch, *wi, *wj, *pinv, *Cp, *Ci,
00071         *ps, *rs, nbl, nb2, *p, *q, *cc, *rr, *r, *s, ok ;
00072     cs *C ;
00073     csd *D, *scc ;
00074     /* --- Maximum matching --- */
00075     if (!CS_CSC (A)) return (NULL) ;    /* check inputs */
00076     m = A->m ; n = A->n ;
00077     D = cs_dalloc (m, n) ;             /* allocate result */
00078     if (!D) return (NULL) ;
00079     p = D->p ; q = D->q ; r = D->r ; s = D->s ; cc = D->cc ; rr = D->rr ;
00080     jmatch = cs_maxtrans (A, seed) ;    /* max transversal */
00081     imatch = jmatch + m ;              /* imatch = inverse of jmatch */
00082     if (!jmatch) return (cs_ddone (D, NULL, jmatch, 0)) ;

```

```

00083  /* --- Coarse decomposition ----- */
00084  wi = r ; wj = s ; /* use r and s as workspace */
00085  for (j = 0 ; j < n ; j++) wj [j] = -1 ; /* unmark all cols for bfs */
00086  for (i = 0 ; i < m ; i++) wi [i] = -1 ; /* unmark all rows for bfs */
00087  cs_bfs (A, n, wi, wj, q, imatch, jmatch, 1) ; /* find C1, R1 from C0 */
00088  ok = cs_bfs (A, m, wj, wi, p, jmatch, imatch, 3) ; /* find R3, C3 from R0 */
00089  if (!ok) return (cs_ddone (D, NULL, jmatch, 0)) ;
00090  cs_unmatched (n, wj, q, cc, 0) ; /* unmatched set C0 */
00091  cs_matched (n, wj, imatch, p, q, cc, rr, 1, 1) ; /* set R1 and C1 */
00092  cs_matched (n, wj, imatch, p, q, cc, rr, 2, -1) ; /* set R2 and C2 */
00093  cs_matched (n, wj, imatch, p, q, cc, rr, 3, 3) ; /* set R3 and C3 */
00094  cs_unmatched (m, wi, p, rr, 3) ; /* unmatched set R0 */
00095  cs_free (jmatch) ;
00096  /* --- Fine decomposition ----- */
00097  pinv = cs_pinv (p, m) ; /* pinv=p' */
00098  if (!pinv) return (cs_ddone (D, NULL, NULL, 0)) ;
00099  C = cs_permute (A, pinv, q, 0) ; /* C=A(p,q) (it will hold A(R2,C2)) */
00100  cs_free (pinv) ;
00101  if (!C) return (cs_ddone (D, NULL, NULL, 0)) ;
00102  Cp = C->p ;
00103  nc = cc [3] - cc [2] ; /* delete cols C0, C1, and C3 from C */
00104  if (cc [2] > 0) for (j = cc [2] ; j <= cc [3] ; j++) Cp [j-cc[2]] = Cp [j] ;
00105  C->n = nc ;
00106  if (rr [2] - rr [1] < m) /* delete rows R0, R1, and R3 from C */
00107  {
00108      cs_fkeep (C, cs_rprune, rr) ;
00109      cnz = Cp [nc] ;
00110      Ci = C->i ;
00111      if (rr [1] > 0) for (k = 0 ; k < cnz ; k++) Ci [k] -= rr [1] ;
00112  }
00113  C->m = nc ;
00114  scc = cs_scc (C) ; /* find strongly connected components of C */
00115  if (!scc) return (cs_ddone (D, C, NULL, 0)) ;
00116  /* --- Combine coarse and fine decompositions ----- */
00117  ps = scc->p ; /* C(ps,ps) is the permuted matrix */
00118  rs = scc->r ; /* kth block is rs[k].rs[k+1]-1 */
00119  nb1 = scc->nb ; /* # of blocks of A(R2,C2) */
00120  for (k = 0 ; k < nc ; k++) wj [k] = q [ps [k] + cc [2]] ;
00121  for (k = 0 ; k < nc ; k++) q [k + cc [2]] = wj [k] ;
00122  for (k = 0 ; k < nc ; k++) wi [k] = p [ps [k] + rr [1]] ;
00123  for (k = 0 ; k < nc ; k++) p [k + rr [1]] = wi [k] ;
00124  nb2 = 0 ; /* create the fine block partitions */
00125  r [0] = s [0] = 0 ;
00126  if (cc [2] > 0) nb2++ ; /* leading coarse block A (R1, [C0 C1]) */
00127  for (k = 0 ; k < nb1 ; k++) /* coarse block A (R2,C2) */
00128  {
00129      r [nb2] = rs [k] + rr [1] ; /* A (R2,C2) splits into nb1 fine blocks */
00130      s [nb2] = rs [k] + cc [2] ;
00131      nb2++ ;
00132  }
00133  if (rr [2] < m)
00134  {
00135      r [nb2] = rr [2] ; /* trailing coarse block A ([R3 R0], C3) */
00136      s [nb2] = cc [3] ;
00137      nb2++ ;
00138  }
00139  r [nb2] = m ;
00140  s [nb2] = n ;
00141  D->nb = nb2 ;
00142  cs_dfree (scc) ;
00143  return (cs_ddone (D, C, NULL, 1)) ;
00144 }

```

## 4.55 csparse/Source/cs\_droptol.c File Reference

```
#include "cs.h"
```

### Functions

- [csi cs\\_droptol](#) (cs \*A, double tol)

#### 4.55.1 Function Documentation

## 4.55.1.1 cs\_droptol()

```
csi cs_droptol (
    cs * A,
    double tol )
```

Definition at line 6 of file [cs\\_droptol.c](#).

## 4.56 cs\_droptol.c

[Go to the documentation of this file.](#)

```
00001 #include "cs.h"
00002 static csi cs_tol (csi i, csi j, double aij, void *tol)
00003 {
00004     return (fabs (aij) > *((double *) tol)) ;
00005 }
00006 csi cs_droptol (cs *A, double tol)
00007 {
00008     return (cs_fkeep (A, &cs_tol, &tol)) ;    /* keep all large entries */
00009 }
```

## 4.57 csparse/Source/cs\_dropzeros.c File Reference

```
#include "cs.h"
```

## Functions

- [csi cs\\_dropzeros](#) (cs \*A)

## 4.57.1 Function Documentation

## 4.57.1.1 cs\_dropzeros()

```
csi cs_dropzeros (
    cs * A )
```

Definition at line 6 of file [cs\\_dropzeros.c](#).

## 4.58 cs\_dropzeros.c

[Go to the documentation of this file.](#)

```
00001 #include "cs.h"
00002 static csi cs_nonzero (csi i, csi j, double aij, void *other)
00003 {
00004     return (aij != 0) ;
00005 }
00006 csi cs_dropzeros (cs *A)
00007 {
00008     return (cs_fkeep (A, &cs_nonzero, NULL)) ;    /* keep all nonzero entries */
00009 }
```

## 4.59 csparse/Source/cs\_dupl.c File Reference

```
#include "cs.h"
```

### Functions

- [csi cs\\_dupl \(cs \\*A\)](#)

### 4.59.1 Function Documentation

#### 4.59.1.1 cs\_dupl()

```
csi cs_dupl (
    cs * A )
```

Definition at line 3 of file [cs\\_dupl.c](#).

## 4.60 cs\_dupl.c

[Go to the documentation of this file.](#)

```
00001 #include "cs.h"
00002 /* remove duplicate entries from A */
00003 csi cs_dupl (cs *A)
00004 {
00005     csi i, j, p, q, nz = 0, n, m, *Ap, *Ai, *w ;
00006     double *Ax ;
00007     if (!CS_CSC (A)) return (0) ; /* check inputs */
00008     m = A->m ; n = A->n ; Ap = A->p ; Ai = A->i ; Ax = A->x ;
00009     w = cs_malloc (m, sizeof (csi)) ; /* get workspace */
00010     if (!w) return (0) ; /* out of memory */
00011     for (i = 0 ; i < m ; i++) w [i] = -1 ; /* row i not yet seen */
00012     for (j = 0 ; j < n ; j++)
00013     {
00014         q = nz ; /* column j will start at q */
00015         for (p = Ap [j] ; p < Ap [j+1] ; p++)
00016         {
00017             i = Ai [p] ; /* A(i,j) is nonzero */
00018             if (w [i] >= q)
00019             {
00020                 Ax [w [i]] += Ax [p] ; /* A(i,j) is a duplicate */
00021             }
00022             else
00023             {
00024                 w [i] = nz ; /* record where row i occurs */
00025                 Ai [nz] = i ; /* keep A(i,j) */
00026                 Ax [nz++] = Ax [p] ;
00027             }
00028         }
00029         Ap [j] = q ; /* record start of column j */
00030     }
00031     Ap [n] = nz ; /* finalize A */
00032     cs_free (w) ; /* free workspace */
00033     return (cs_sprealloc (A, 0)) ; /* remove extra space from A */
00034 }
```

## 4.61 csparse/Source/cs\_entry.c File Reference

```
#include "cs.h"
```

### Functions

- [csi cs\\_entry](#) ([cs](#) \*T, [csi](#) i, [csi](#) j, double x)

### 4.61.1 Function Documentation

#### 4.61.1.1 cs\_entry()

```
csi cs_entry (
    cs * T,
    csi i,
    csi j,
    double x )
```

Definition at line 3 of file [cs\\_entry.c](#).

## 4.62 cs\_entry.c

[Go to the documentation of this file.](#)

```
00001 #include "cs.h"
00002 /* add an entry to a triplet matrix; return 1 if ok, 0 otherwise */
00003 csi cs_entry (cs *T, csi i, csi j, double x)
00004 {
00005     if (!CS_TRIPLET (T) || i < 0 || j < 0) return (0) ;    /* check inputs */
00006     if (T->nz >= T->nzmax && !cs_sprealloc (T, 2*(T->nzmax))) return (0) ;
00007     if (T->x) T->x [T->nz] = x ;
00008     T->i [T->nz] = i ;
00009     T->p [T->nz++] = j ;
00010     T->m = CS_MAX (T->m, i+1) ;
00011     T->n = CS_MAX (T->n, j+1) ;
00012     return (1) ;
00013 }
```

## 4.63 csparse/Source/cs\_ereach.c File Reference

```
#include "cs.h"
```

### Functions

- [csi cs\\_ereach](#) (const [cs](#) \*A, [csi](#) k, const [csi](#) \*parent, [csi](#) \*s, [csi](#) \*w)

## 4.63.1 Function Documentation

### 4.63.1.1 cs\_ereach()

```
csi cs_ereach (
    const cs * A,
    csi k,
    const csi * parent,
    csi * s,
    csi * w )
```

Definition at line 3 of file [cs\\_ereach.c](#).

## 4.64 cs\_ereach.c

[Go to the documentation of this file.](#)

```
00001 #include "cs.h"
00002 /* find nonzero pattern of Cholesky L(k,1:k-1) using etree and triu(A(:,k)) */
00003 csi cs_ereach (const cs *A, csi k, const csi *parent, csi *s, csi *w)
00004 {
00005     csi i, p, n, len, top, *Ap, *Ai ;
00006     if (!CS_CSC (A) || !parent || !s || !w) return (-1) ; /* check inputs */
00007     top = n = A->n ; Ap = A->p ; Ai = A->i ;
00008     CS_MARK (w, k) ; /* mark node k as visited */
00009     for (p = Ap [k] ; p < Ap [k+1] ; p++)
00010     {
00011         i = Ai [p] ; /* A(i,k) is nonzero */
00012         if (i > k) continue ; /* only use upper triangular part of A */
00013         for (len = 0 ; !CS_MARKED (w,i) ; i = parent [i]) /* traverse up etree*/
00014         {
00015             s [len++] = i ; /* L(k,i) is nonzero */
00016             CS_MARK (w, i) ; /* mark i as visited */
00017         }
00018         while (len > 0) s [--top] = s [--len] ; /* push path onto stack */
00019     }
00020     for (p = top ; p < n ; p++) CS_MARK (w, s [p]) ; /* unmark all nodes */
00021     CS_MARK (w, k) ; /* unmark node k */
00022     return (top) ; /* s [top..n-1] contains pattern of L(k,:)*/
00023 }
```

## 4.65 csparse/Source/cs\_etree.c File Reference

```
#include "cs.h"
```

### Functions

- [csi \\* cs\\_etree](#) (const [cs](#) \*A, [csi](#) ata)

### 4.65.1 Function Documentation



## 4.65.1.1 cs\_etree()

```
csi * cs_etree (
    const cs * A,
    csi ata )
```

Definition at line 3 of file [cs\\_etree.c](#).

## 4.66 cs\_etree.c

[Go to the documentation of this file.](#)

```
00001 #include "cs.h"
00002 /* compute the etree of A (using triu(A), or A'A without forming A'A */
00003 csi *cs_etree (const cs *A, csi ata)
00004 {
00005     csi i, k, p, m, n, inext, *Ap, *Ai, *w, *parent, *ancestor, *prev ;
00006     if (!CS_CSC (A)) return (NULL) ; /* check inputs */
00007     m = A->m ; n = A->n ; Ap = A->p ; Ai = A->i ;
00008     parent = cs_malloc (n, sizeof (csi)) ; /* allocate result */
00009     w = cs_malloc (n + (ata ? m : 0), sizeof (csi)) ; /* get workspace */
00010     if (!w || !parent) return (cs_idone (parent, NULL, w, 0)) ;
00011     ancestor = w ; prev = w + n ;
00012     if (ata) for (i = 0 ; i < m ; i++) prev [i] = -1 ;
00013     for (k = 0 ; k < n ; k++)
00014     {
00015         parent [k] = -1 ; /* node k has no parent yet */
00016         ancestor [k] = -1 ; /* nor does k have an ancestor */
00017         for (p = Ap [k] ; p < Ap [k+1] ; p++)
00018         {
00019             i = ata ? (prev [Ai [p]]) : (Ai [p]) ;
00020             for ( ; i != -1 && i < k ; i = inext) /* traverse from i to k */
00021             {
00022                 inext = ancestor [i] ; /* inext = ancestor of i */
00023                 ancestor [i] = k ; /* path compression */
00024                 if (inext == -1) parent [i] = k ; /* no anc., parent is k */
00025             }
00026             if (ata) prev [Ai [p]] = k ;
00027         }
00028     }
00029     return (cs_idone (parent, NULL, w, 1)) ;
00030 }
```

## 4.67 csparse/Source/cs\_fkeep.c File Reference

```
#include "cs.h"
```

## Functions

- [csi cs\\_fkeep](#) (cs \*A, csi(\*fkeep)(csi, csi, double, void \*), void \*other)

## 4.67.1 Function Documentation

#### 4.67.1.1 cs\_fkeep()

```
csi cs_fkeep (
    cs * A,
    csi(*) (csi, csi, double, void *) fkeep,
    void * other )
```

Definition at line 3 of file [cs\\_fkeep.c](#).

### 4.68 cs\_fkeep.c

[Go to the documentation of this file.](#)

```
00001 #include "cs.h"
00002 /* drop entries for which fkeep(A(i,j)) is false; return nz if OK, else -1 */
00003 csi cs_fkeep (cs *A, csi (*fkeep) (csi, csi, double, void *), void *other)
00004 {
00005     csi j, p, nz = 0, n, *Ap, *Ai ;
00006     double *Ax ;
00007     if (!CS_CSC (A) || !fkeep) return (-1) ; /* check inputs */
00008     n = A->n ; Ap = A->p ; Ai = A->i ; Ax = A->x ;
00009     for (j = 0 ; j < n ; j++)
00010     {
00011         p = Ap [j] ; /* get current location of col j */
00012         Ap [j] = nz ; /* record new location of col j */
00013         for ( ; p < Ap [j+1] ; p++)
00014         {
00015             if (fkeep (Ai [p], j, Ax ? Ax [p] : 1, other))
00016             {
00017                 if (Ax) Ax [nz] = Ax [p] ; /* keep A(i,j) */
00018                 Ai [nz++] = Ai [p] ;
00019             }
00020         }
00021     }
00022     Ap [n] = nz ; /* finalize A */
00023     cs_sprealloc (A, 0) ; /* remove extra space from A */
00024     return (nz) ;
00025 }
```

### 4.69 csparse/Source/cs\_gaxpy.c File Reference

```
#include "cs.h"
```

#### Functions

- [csi cs\\_gaxpy](#) (const cs \*A, const double \*x, double \*y)

#### 4.69.1 Function Documentation

##### 4.69.1.1 cs\_gaxpy()

```
csi cs_gaxpy (
    const cs * A,
    const double * x,
    double * y )
```

Definition at line 3 of file [cs\\_gaxpy.c](#).

## 4.70 cs\_gaxpy.c

[Go to the documentation of this file.](#)

```
00001 #include "cs.h"
00002 /* y = A*x+y */
00003 csi cs_gaxpy (const cs *A, const double *x, double *y)
00004 {
00005     csi p, j, n, *Ap, *Ai ;
00006     double *Ax ;
00007     if (!CS_CSC (A) || !x || !y) return (0) ;      /* check inputs */
00008     n = A->n ; Ap = A->p ; Ai = A->i ; Ax = A->x ;
00009     for (j = 0 ; j < n ; j++)
00010     {
00011         for (p = Ap [j] ; p < Ap [j+1] ; p++)
00012         {
00013             y [Ai [p]] += Ax [p] * x [j] ;
00014         }
00015     }
00016     return (1) ;
00017 }
```

## 4.71 csparse/Source/cs\_happly.c File Reference

```
#include "cs.h"
```

### Functions

- [csi cs\\_happly](#) (const cs \*V, csi i, double beta, double \*x)

### 4.71.1 Function Documentation

#### 4.71.1.1 cs\_happly()

```
csi cs_happly (
    const cs * V,
    csi i,
    double beta,
    double * x )
```

Definition at line 3 of file [cs\\_happly.c](#).

## 4.72 cs\_happly.c

[Go to the documentation of this file.](#)

```
00001 #include "cs.h"
00002 /* apply the ith Householder vector to x */
00003 csi cs_happly (const cs *V, csi i, double beta, double *x)
00004 {
00005     csi p, *Vp, *Vi ;
00006     double *Vx, tau = 0 ;
00007     if (!CS_CSC (V) || !x) return (0) ;      /* check inputs */
00008     Vp = V->p ; Vi = V->i ; Vx = V->x ;
00009     for (p = Vp [i] ; p < Vp [i+1] ; p++) /* tau = v'*x */
00010     {
00011         tau += Vx [p] * x [Vi [p]] ;
00012     }
00013     tau *= beta ;      /* tau = beta*(v'*x) */
00014     for (p = Vp [i] ; p < Vp [i+1] ; p++) /* x = x - v*tau */
00015     {
00016         x [Vi [p]] -= Vx [p] * tau ;
00017     }
00018     return (1) ;
00019 }
```

## 4.73 csparse/Source/cs\_house.c File Reference

```
#include "cs.h"
```

### Functions

- double [cs\\_house](#) (double \*x, double \*beta, [csi](#) n)

### 4.73.1 Function Documentation

#### 4.73.1.1 cs\_house()

```
double cs_house (
    double * x,
    double * beta,
    csi n )
```

Definition at line 4 of file [cs\\_house.c](#).

## 4.74 cs\_house.c

[Go to the documentation of this file.](#)

```
00001 #include "cs.h"
00002 /* create a Householder reflection [v,beta,s]=house(x), overwrite x with v,
00003  * where (I-beta*v*v')*x = s*e1. See Algo 5.1.1, Golub & Van Loan, 3rd ed. */
00004 double cs\_house (double *x, double *beta, csi n)
00005 {
00006     double s, sigma = 0 ;
00007     csi i ;
00008     if (!x || !beta) return (-1) ; /* check inputs */
00009     for (i = 1 ; i < n ; i++) sigma += x [i] * x [i] ;
00010     if (sigma == 0)
00011     {
00012         s = fabs (x [0]) ; /* s = |x(0)| */
00013         (*beta) = (x [0] <= 0) ? 2 : 0 ;
00014         x [0] = 1 ;
00015     }
00016     else
00017     {
00018         s = sqrt (x [0] * x [0] + sigma) ; /* s = norm (x) */
00019         x [0] = (x [0] <= 0) ? (x [0] - s) : (-sigma / (x [0] + s)) ;
00020         (*beta) = -1. / (s * x [0]) ;
00021     }
00022     return (s) ;
00023 }
```

## 4.75 csparse/Source/cs\_ipvec.c File Reference

```
#include "cs.h"
```

## Functions

- `csi cs_ipvec` (const `csi` \*p, const double \*b, double \*x, `csi` n)

### 4.75.1 Function Documentation

#### 4.75.1.1 cs\_ipvec()

```
csi cs_ipvec (
    const csi * p,
    const double * b,
    double * x,
    csi n )
```

Definition at line 3 of file `cs_ipvec.c`.

## 4.76 cs\_ipvec.c

[Go to the documentation of this file.](#)

```
00001 #include "cs.h"
00002 /* x(p) = b, for dense vectors x and b; p=NULL denotes identity */
00003 csi cs_ipvec (const csi *p, const double *b, double *x, csi n)
00004 {
00005     csi k ;
00006     if (!x || !b) return (0) ;
00007     for (k = 0 ; k < n ; k++) x [p ? p [k] : k] = b [k] ;
00008     return (1) ;
00009 }
```

## 4.77 csparse/Source/cs\_leaf.c File Reference

```
#include "cs.h"
```

## Functions

- `csi cs_leaf` (`csi` i, `csi` j, const `csi` \*first, `csi` \*maxfirst, `csi` \*prevleaf, `csi` \*ancestor, `csi` \*jleaf)

### 4.77.1 Function Documentation

#### 4.77.1.1 cs\_leaf()

```
csi cs_leaf (
    csi i,
    csi j,
    const csi * first,
    csi * maxfirst,
    csi * prevleaf,
    csi * ancestor,
    csi * jleaf )
```

Definition at line 3 of file [cs\\_leaf.c](#).

## 4.78 cs\_leaf.c

[Go to the documentation of this file.](#)

```
00001 #include "cs.h"
00002 /* consider A(i,j), node j in ith row subtree and return lca(jprev,j) */
00003 csi cs_leaf (csi i, csi j, const csi *first, csi *maxfirst, csi *prevleaf,
00004             csi *ancestor, csi *jleaf)
00005 {
00006     csi q, s, sparent, jprev ;
00007     if (!first || !maxfirst || !prevleaf || !ancestor || !jleaf) return (-1) ;
00008     *jleaf = 0 ;
00009     if (i <= j || first [j] <= maxfirst [i]) return (-1) ; /* j not a leaf */
00010     maxfirst [i] = first [j] ; /* update max first[j] seen so far */
00011     jprev = prevleaf [i] ; /* jprev = previous leaf of ith subtree */
00012     prevleaf [i] = j ;
00013     *jleaf = (jprev == -1) ? 1: 2 ; /* j is first or subsequent leaf */
00014     if (*jleaf == 1) return (i) ; /* if 1st leaf, q = root of ith subtree */
00015     for (q = jprev ; q != ancestor [q] ; q = ancestor [q]) ;
00016     for (s = jprev ; s != q ; s = sparent)
00017     {
00018         sparent = ancestor [s] ; /* path compression */
00019         ancestor [s] = q ;
00020     }
00021     return (q) ; /* q = least common ancestor (jprev,j) */
00022 }
```

## 4.79 csparse/Source/cs\_load.c File Reference

```
#include "cs.h"
```

### Functions

- [cs \\* cs\\_load](#) (FILE \*f)

#### 4.79.1 Function Documentation

##### 4.79.1.1 cs\_load()

```
cs * cs_load (
    FILE * f )
```

Definition at line 3 of file [cs\\_load.c](#).

## 4.80 cs\_load.c

[Go to the documentation of this file.](#)

```
00001 #include "cs.h"
00002 /* load a triplet matrix from a file */
00003 cs *cs_load (FILE *f)
00004 {
00005     double i, j ;    /* use double for integers to avoid csi conflicts */
00006     double x ;
00007     cs *T ;
00008     if (!f) return (NULL) ;                                /* check inputs */
00009     T = cs_spalloc (0, 0, 1, 1, 1) ;                        /* allocate result */
00010     while (fscanf (f, "%lg %lg %lg\n", &i, &j, &x) == 3)
00011     {
00012         if (!cs_entry (T, (csi) i, (csi) j, x)) return (cs_spfree (T)) ;
00013     }
00014     return (T) ;
00015 }
```

## 4.81 csparse/Source/cs\_Isolve.c File Reference

```
#include "cs.h"
```

### Functions

- [csi cs\\_Isolve](#) (const [cs](#) \*L, double \*x)

### 4.81.1 Function Documentation

#### 4.81.1.1 cs\_Isolve()

```
csi cs_Isolve (
    const cs * L,
    double * x )
```

Definition at line 3 of file [cs\\_Isolve.c](#).

## 4.82 cs\_Isolve.c

[Go to the documentation of this file.](#)

```
00001 #include "cs.h"
00002 /* solve Lx=b where x and b are dense.  x=b on input, solution on output. */
00003 csi cs_Isolve (const cs *L, double *x)
00004 {
00005     csi p, j, n, *Lp, *Li ;
00006     double *Lx ;
00007     if (!CS_CSC (L) || !x) return (0) ;                    /* check inputs */
00008     n = L->n ; Lp = L->p ; Li = L->i ; Lx = L->x ;
00009     for (j = 0 ; j < n ; j++)
00010     {
00011         x [j] /= Lx [Lp [j]] ;
00012         for (p = Lp [j]+1 ; p < Lp [j+1] ; p++)
00013         {
00014             x [Li [p]] -= Lx [p] * x [j] ;
00015         }
00016     }
00017     return (1) ;
00018 }
```

## 4.83 csparse/Source/cs\_Itsolve.c File Reference

```
#include "cs.h"
```

### Functions

- [csi cs\\_Itsolve](#) (const [cs](#) \*L, double \*x)

### 4.83.1 Function Documentation

#### 4.83.1.1 cs\_Itsolve()

```
csi cs_Itsolve (
    const cs * L,
    double * x )
```

Definition at line 3 of file [cs\\_Itsolve.c](#).

## 4.84 cs\_Itsolve.c

[Go to the documentation of this file.](#)

```
00001 #include "cs.h"
00002 /* solve L'x=b where x and b are dense.  x=b on input, solution on output. */
00003 csi cs_Itsolve (const cs *L, double *x)
00004 {
00005     csi p, j, n, *Lp, *Li ;
00006     double *Lx ;
00007     if (!CS_CSC (L) || !x) return (0) ; /* check inputs */
00008     n = L->n ; Lp = L->p ; Li = L->i ; Lx = L->x ;
00009     for (j = n-1 ; j >= 0 ; j--)
00010     {
00011         for (p = Lp [j]+1 ; p < Lp [j+1] ; p++)
00012         {
00013             x [j] -= Lx [p] * x [Li [p]] ;
00014         }
00015         x [j] /= Lx [Lp [j]] ;
00016     }
00017     return (1) ;
00018 }
```

## 4.85 csparse/Source/cs\_lu.c File Reference

```
#include "cs.h"
```

### Functions

- [csn](#) \* [cs\\_lu](#) (const [cs](#) \*A, const [css](#) \*S, double tol)



## 4.85.1 Function Documentation

### 4.85.1.1 cs\_lu()

```
csn * cs_lu (
    const cs * A,
    const css * S,
    double tol )
```

Definition at line 3 of file [cs\\_lu.c](#).

## 4.86 cs\_lu.c

[Go to the documentation of this file.](#)

```
00001 #include "cs.h"
00002 /* [L,U,pinv]=lu(A, [q lnz unz]). lnz and unz can be guess */
00003 csn *cs_lu (const cs *A, const css *S, double tol)
00004 {
00005     cs *L, *U ;
00006     csn *N ;
00007     double pivot, *Lx, *Ux, *x, a, t ;
00008     csi *Lp, *Li, *Up, *Ui, *pinv, *xi, *q, n, ipiv, k, top, p, i, col, lnz, unz;
00009     if (!CS_CSC (A) || !S) return (NULL) ; /* check inputs */
00010     n = A->n ;
00011     q = S->q ; lnz = S->lnz ; unz = S->unz ;
00012     x = cs_malloc (n, sizeof (double)) ; /* get double workspace */
00013     xi = cs_malloc (2*n, sizeof (csi)) ; /* get csi workspace */
00014     N = cs_calloc (1, sizeof (csn)) ; /* allocate result */
00015     if (!x || !xi || !N) return (cs_ndone (N, NULL, xi, x, 0)) ;
00016     N->L = L = cs_spalloc (n, n, lnz, 1, 0) ; /* allocate result L */
00017     N->U = U = cs_spalloc (n, n, unz, 1, 0) ; /* allocate result U */
00018     N->pinv = pinv = cs_malloc (n, sizeof (csi)) ; /* allocate result pinv */
00019     if (!L || !U || !pinv) return (cs_ndone (N, NULL, xi, x, 0)) ;
00020     Lp = L->p ; Up = U->p ;
00021     for (i = 0 ; i < n ; i++) x [i] = 0 ; /* clear workspace */
00022     for (i = 0 ; i < n ; i++) pinv [i] = -1 ; /* no rows pivotal yet */
00023     for (k = 0 ; k <= n ; k++) Lp [k] = 0 ; /* no cols of L yet */
00024     lnz = unz = 0 ;
00025     for (k = 0 ; k < n ; k++) /* compute L(:,k) and U(:,k) */
00026     {
00027         /* --- Triangular solve --- */
00028         Lp [k] = lnz ; /* L(:,k) starts here */
00029         Up [k] = unz ; /* U(:,k) starts here */
00030         if ((lnz + n > L->nzmax && !cs_sprealloc (L, 2*L->nzmax + n)) ||
00031             (unz + n > U->nzmax && !cs_sprealloc (U, 2*U->nzmax + n)))
00032         {
00033             return (cs_ndone (N, NULL, xi, x, 0)) ;
00034         }
00035         Li = L->i ; Lx = L->x ; Ui = U->i ; Ux = U->x ;
00036         col = q ? (q [k]) : k ;
00037         top = cs_spsolve (L, A, col, xi, x, pinv, 1) ; /* x = L\A(:,col) */
00038         /* --- Find pivot --- */
00039         ipiv = -1 ;
00040         a = -1 ;
00041         for (p = top ; p < n ; p++)
00042         {
00043             i = xi [p] ; /* x(i) is nonzero */
00044             if (pinv [i] < 0) /* row i is not yet pivotal */
00045             {
00046                 if ((t = fabs (x [i])) > a)
00047                 {
00048                     a = t ; /* largest pivot candidate so far */
00049                     ipiv = i ;
00050                 }
00051             }
00052             else /* x(i) is the entry U(pinv[i],k) */
00053             {
00054                 Ui [unz] = pinv [i] ;
00055                 Ux [unz++] = x [i] ;
00056             }
00057         }
00058     }
00059 }
```

```

00057     }
00058     if (ipiv == -1 || a <= 0) return (cs_ndone (N, NULL, xi, x, 0)) ;
00059     /* tol=1 for partial pivoting; tol<1 gives preference to diagonal */
00060     if (pinv [col] < 0 && fabs (x [col]) >= a*tol) ipiv = col ;
00061     /* --- Divide by pivot ----- */
00062     pivot = x [ipiv] ; /* the chosen pivot */
00063     Ui [unz] = k ; /* last entry in U(:,k) is U(k,k) */
00064     Ux [unz++] = pivot ;
00065     pinv [ipiv] = k ; /* ipiv is the kth pivot row */
00066     Li [lnz] = ipiv ; /* first entry in L(:,k) is L(k,k) = 1 */
00067     Lx [lnz++] = 1 ;
00068     for (p = top ; p < n ; p++) /* L(k+1:n,k) = x / pivot */
00069     {
00070         i = xi [p] ;
00071         if (pinv [i] < 0) /* x(i) is an entry in L(:,k) */
00072         {
00073             Li [lnz] = i ; /* save unpermuted row in L */
00074             Lx [lnz++] = x [i] / pivot ; /* scale pivot column */
00075         }
00076         x [i] = 0 ; /* x [0..n-1] = 0 for next k */
00077     }
00078 }
00079 /* --- Finalize L and U ----- */
00080 Lp [n] = lnz ;
00081 Up [n] = unz ;
00082 Li = L->i ; /* fix row indices of L for final pinv */
00083 for (p = 0 ; p < lnz ; p++) Li [p] = pinv [Li [p]] ;
00084 cs_sprealloc (L, 0) ; /* remove extra space from L and U */
00085 cs_sprealloc (U, 0) ;
00086 return (cs_ndone (N, NULL, xi, x, 1)) ; /* success */
00087 }

```

## 4.87 csparse/Source/cs\_lusol.c File Reference

```
#include "cs.h"
```

### Functions

- [csi cs\\_lusol](#) (csi order, const [cs](#) \*A, double \*b, double tol)

### 4.87.1 Function Documentation

#### 4.87.1.1 cs\_lusol()

```

csi cs_lusol (
    csi order,
    const cs * A,
    double * b,
    double tol )

```

Definition at line 3 of file [cs\\_lusol.c](#).

## 4.88 cs\_lusol.c

[Go to the documentation of this file.](#)

```
00001 #include "cs.h"
00002 /* x=A\b where A is unsymmetric; b overwritten with solution */
00003 csi cs_lusol (csi order, const cs *A, double *b, double tol)
00004 {
00005     double *x ;
00006     css *S ;
00007     csn *N ;
00008     csi n, ok ;
00009     if (!CS_CSC (A) || !b) return (0) ; /* check inputs */
00010     n = A->n ;
00011     S = cs_sqr (order, A, 0) ; /* ordering and symbolic analysis */
00012     N = cs_lu (A, S, tol) ; /* numeric LU factorization */
00013     x = cs_malloc (n, sizeof (double)) ; /* get workspace */
00014     ok = (S && N && x) ;
00015     if (ok)
00016     {
00017         cs_ipvec (N->pinv, b, x, n) ; /* x = b(p) */
00018         cs_lsolve (N->L, x) ; /* x = L\ x */
00019         cs_usolve (N->U, x) ; /* x = U\ x */
00020         cs_ipvec (S->q, x, b, n) ; /* b(q) = x */
00021     }
00022     cs_free (x) ;
00023     cs_sfree (S) ;
00024     cs_nfree (N) ;
00025     return (ok) ;
00026 }
```

## 4.89 csparse/Source/cs\_malloc.c File Reference

```
#include "cs.h"
```

### Functions

- void \* [cs\\_malloc](#) (csi n, size\_t size)
- void \* [cs\\_calloc](#) (csi n, size\_t size)
- void \* [cs\\_free](#) (void \*p)
- void \* [cs\\_realloc](#) (void \*p, csi n, size\_t size, csi \*ok)

### 4.89.1 Function Documentation

#### 4.89.1.1 cs\_calloc()

```
void * cs_calloc (
    csi n,
    size_t size )
```

Definition at line 16 of file [cs\\_malloc.c](#).

#### 4.89.1.2 cs\_free()

```
void * cs_free (
    void * p )
```

Definition at line 22 of file [cs\\_malloc.c](#).

#### 4.89.1.3 cs\_malloc()

```
void * cs_malloc (
    csi n,
    size_t size )
```

Definition at line 10 of file [cs\\_malloc.c](#).

#### 4.89.1.4 cs\_realloc()

```
void * cs_realloc (
    void * p,
    csi n,
    size_t size,
    csi * ok )
```

Definition at line 29 of file [cs\\_malloc.c](#).

### 4.90 cs\_malloc.c

[Go to the documentation of this file.](#)

```
00001 #include "cs.h"
00002 #ifdef MATLAB_MEX_FILE
00003 #define malloc mxMalloc
00004 #define free mxFree
00005 #define realloc mxRealloc
00006 #define calloc mxCalloc
00007 #endif
00008
00009 /* wrapper for malloc */
00010 void *cs_malloc (csi n, size_t size)
00011 {
00012     return (malloc (CS_MAX (n,1) * size)) ;
00013 }
00014
00015 /* wrapper for calloc */
00016 void *cs_calloc (csi n, size_t size)
00017 {
00018     return (calloc (CS_MAX (n,1), size)) ;
00019 }
00020
00021 /* wrapper for free */
00022 void *cs_free (void *p)
00023 {
00024     if (p) free (p) ;           /* free p if it is not already NULL */
00025     return (NULL) ;           /* return NULL to simplify the use of cs_free */
00026 }
00027
00028 /* wrapper for realloc */
00029 void *cs_realloc (void *p, csi n, size_t size, csi *ok)
00030 {
00031     void *pnew ;
00032     pnew = realloc (p, CS_MAX (n,1) * size) ; /* realloc the block */
00033     *ok = (pnew != NULL) ;                  /* realloc fails if pnew is NULL */
00034     return ((*ok) ? pnew : p) ;             /* return original p if failure */
00035 }
```

## 4.91 csparse/Source/cs\_maxtrans.c File Reference

```
#include "cs.h"
```

### Functions

- [csi \\* cs\\_maxtrans](#) (const [cs](#) \*A, [csi](#) seed)

### 4.91.1 Function Documentation

#### 4.91.1.1 cs\_maxtrans()

```
csi * cs_maxtrans (
    const cs * A,
    csi seed )
```

Definition at line 44 of file [cs\\_maxtrans.c](#).

## 4.92 cs\_maxtrans.c

[Go to the documentation of this file.](#)

```
00001 #include "cs.h"
00002 /* find an augmenting path starting at column k and extend the match if found */
00003 static void cs_augment (csi k, const cs *A, csi *jmatch, csi *cheap, csi *w,
00004     csi *js, csi *is, csi *ps)
00005 {
00006     csi found = 0, p, i = -1, *Ap = A->p, *Ai = A->i, head = 0, j ;
00007     js [0] = k ;
00008     while (head >= 0)
00009     {
00010         /* --- Start (or continue) depth-first-search at node j ----- */
00011         j = js [head] ;
00012         if (w [j] != k)
00013         {
00014             w [j] = k ;
00015             for (p = cheap [j] ; p < Ap [j+1] && !found ; p++)
00016             {
00017                 i = Ai [p] ;
00018                 found = (jmatch [i] == -1) ;
00019             }
00020             cheap [j] = p ;
00021             if (found)
00022             {
00023                 is [head] = i ;
00024                 break ;
00025             }
00026             ps [head] = Ap [j] ;
00027         }
00028         /* --- Depth-first-search of neighbors of j ----- */
00029         for (p = ps [head] ; p < Ap [j+1] ; p++)
00030         {
00031             i = Ai [p] ;
00032             if (w [jmatch [i]] == k) continue ;
00033             ps [head] = p + 1 ;
00034             is [head] = i ;
00035             js [++head] = jmatch [i] ;
00036             break ;
00037         }
00038         if (p == Ap [j+1]) head-- ;
00039     }
```

```

00040     if (found) for (p = head ; p >= 0 ; p--) jmatch [is [p]] = js [p] ;
00041 }
00042
00043 /* find a maximum transversal */
00044 csi *cs_maxtrans (const cs *A, csi seed) /*[jmatch [0..m-1]; imatch [0..n-1]]*/
00045 {
00046     csi i, j, k, n, m, p, n2 = 0, m2 = 0, *Ap, *jmatch, *w, *cheap, *js, *is,
00047         *ps, *Ai, *Cp, *jmatch, *imatch, *q ;
00048     cs *C ;
00049     if (!CS_CSC (A)) return (NULL) ; /* check inputs */
00050     n = A->n ; m = A->m ; Ap = A->p ; Ai = A->i ;
00051     w = jmatch = cs_calloc (m+n, sizeof (csi)) ; /* allocate result */
00052     if (!jmatch) return (NULL) ;
00053     for (k = 0, j = 0 ; j < n ; j++) /* count nonempty rows and columns */
00054     {
00055         n2 += (Ap [j] < Ap [j+1]) ;
00056         for (p = Ap [j] ; p < Ap [j+1] ; p++)
00057         {
00058             w [Ai [p]] = 1 ;
00059             k += (j == Ai [p]) ; /* count entries already on diagonal */
00060         }
00061     }
00062     if (k == CS_MIN (m,n)) /* quick return if diagonal zero-free */
00063     {
00064         jmatch = jmatch ; imatch = jmatch + m ;
00065         for (i = 0 ; i < k ; i++) jmatch [i] = i ;
00066         for ( ; i < m ; i++) jmatch [i] = -1 ;
00067         for (j = 0 ; j < k ; j++) imatch [j] = j ;
00068         for ( ; j < n ; j++) imatch [j] = -1 ;
00069         return (cs_idone (jmatch, NULL, NULL, 1)) ;
00070     }
00071     for (i = 0 ; i < m ; i++) m2 += w [i] ;
00072     C = (m2 < n2) ? cs_transpose (A,0) : ((cs *) A) ; /* transpose if needed */
00073     if (!C) return (cs_idone (jmatch, (m2 < n2) ? C : NULL, NULL, 0)) ;
00074     n = C->n ; m = C->m ; Cp = C->p ;
00075     jmatch = (m2 < n2) ? jmatch + n : jmatch ;
00076     imatch = (m2 < n2) ? jmatch : jmatch + m ;
00077     w = cs_malloc (5*n, sizeof (csi)) ; /* get workspace */
00078     if (!w) return (cs_idone (jmatch, (m2 < n2) ? C : NULL, w, 0)) ;
00079     cheap = w + n ; js = w + 2*n ; is = w + 3*n ; ps = w + 4*n ;
00080     for (j = 0 ; j < n ; j++) cheap [j] = Cp [j] ; /* for cheap assignment */
00081     for (j = 0 ; j < n ; j++) w [j] = -1 ; /* all columns unflagged */
00082     for (i = 0 ; i < m ; i++) jmatch [i] = -1 ; /* nothing matched yet */
00083     q = cs_randperm (n, seed) ; /* q = random permutation */
00084     for (k = 0 ; k < n ; k++) /* augment, starting at column q[k] */
00085     {
00086         cs_augment (q ? q [k] : k, C, jmatch, cheap, w, js, is, ps) ;
00087     }
00088     cs_free (q) ;
00089     for (j = 0 ; j < n ; j++) imatch [j] = -1 ; /* find row match */
00090     for (i = 0 ; i < m ; i++) if (jmatch [i] >= 0) imatch [jmatch [i]] = i ;
00091     return (cs_idone (jmatch, (m2 < n2) ? C : NULL, w, 1)) ;
00092 }

```

## 4.93 csparse/Source/cs\_multiply.c File Reference

```
#include "cs.h"
```

### Functions

- `cs * cs_multiply (const cs *A, const cs *B)`

#### 4.93.1 Function Documentation

## 4.93.1.1 cs\_multiply()

```
cs * cs_multiply (
    const cs * A,
    const cs * B )
```

Definition at line 3 of file [cs\\_multiply.c](#).

## 4.94 cs\_multiply.c

[Go to the documentation of this file.](#)

```
00001 #include "cs.h"
00002 /* C = A*B */
00003 cs *cs_multiply (const cs *A, const cs *B)
00004 {
00005     csi p, j, nz = 0, anz, *Cp, *Ci, *Bp, m, n, bnz, *w, values, *Bi ;
00006     double *x, *Bx, *Cx ;
00007     cs *C ;
00008     if (!CS_CSC (A) || !CS_CSC (B)) return (NULL) ; /* check inputs */
00009     if (A->n != B->m) return (NULL) ;
00010     m = A->m ; anz = A->p [A->n] ;
00011     n = B->n ; Bp = B->p ; Bi = B->i ; Bx = B->x ; bnz = Bp [n] ;
00012     w = cs_calloc (m, sizeof (csi)) ; /* get workspace */
00013     values = (A->x != NULL) && (Bx != NULL) ;
00014     x = values ? cs_malloc (m, sizeof (double)) : NULL ; /* get workspace */
00015     C = cs_salloc (m, n, anz + bnz, values, 0) ; /* allocate result */
00016     if (!C || !w || (values && !x)) return (cs_done (C, w, x, 0)) ;
00017     Cp = C->p ;
00018     for (j = 0 ; j < n ; j++)
00019     {
00020         if (nz + m > C->nzmax && !cs_sprealloc (C, 2*(C->nzmax)+m))
00021         {
00022             return (cs_done (C, w, x, 0)) ; /* out of memory */
00023         }
00024         Ci = C->i ; Cx = C->x ; /* C->i and C->x may be reallocated */
00025         Cp [j] = nz ; /* column j of C starts here */
00026         for (p = Bp [j] ; p < Bp [j+1] ; p++)
00027         {
00028             nz = cs_scatter (A, Bi [p], Bx ? Bx [p] : 1, w, x, j+1, C, nz) ;
00029         }
00030         if (values) for (p = Cp [j] ; p < nz ; p++) Cx [p] = x [Ci [p]] ;
00031     }
00032     Cp [n] = nz ; /* finalize the last column of C */
00033     cs_sprealloc (C, 0) ; /* remove extra space from C */
00034     return (cs_done (C, w, x, 1)) ; /* success; free workspace, return C */
00035 }
```

## 4.95 csparse/Source/cs\_norm.c File Reference

```
#include "cs.h"
```

## Functions

- double [cs\\_norm](#) (const cs \*A)

## 4.95.1 Function Documentation

#### 4.95.1.1 cs\_norm()

```
double cs_norm (
    const cs * A )
```

Definition at line 3 of file [cs\\_norm.c](#).

### 4.96 cs\_norm.c

[Go to the documentation of this file.](#)

```
00001 #include "cs.h"
00002 /* 1-norm of a sparse matrix = max (sum (abs (A))), largest column sum */
00003 double cs_norm (const cs *A)
00004 {
00005     csi p, j, n, *Ap ;
00006     double *Ax, norm = 0, s ;
00007     if (!CS_CSC (A) || !A->x) return (-1) ;           /* check inputs */
00008     n = A->n ; Ap = A->p ; Ax = A->x ;
00009     for (j = 0 ; j < n ; j++)
00010     {
00011         for (s = 0, p = Ap [j] ; p < Ap [j+1] ; p++) s += fabs (Ax [p]) ;
00012         norm = CS_MAX (norm, s) ;
00013     }
00014     return (norm) ;
00015 }
```

### 4.97 csparse/Source/cs\_permute.c File Reference

```
#include "cs.h"
```

#### Functions

- [cs \\* cs\\_permute](#) (const [cs](#) \*A, const [csi](#) \*pinv, const [csi](#) \*q, [csi](#) values)

#### 4.97.1 Function Documentation

##### 4.97.1.1 cs\_permute()

```
cs * cs_permute (
    const cs * A,
    const csi * pinv,
    const csi * q,
    csi values )
```

Definition at line 3 of file [cs\\_permute.c](#).



## 4.98 cs\_permute.c

[Go to the documentation of this file.](#)

```

00001 #include "cs.h"
00002 /* C = A(p,q) where p and q are permutations of 0..m-1 and 0..n-1. */
00003 cs *cs_permute (const cs *A, const csi *pinv, const csi *q, csi values)
00004 {
00005     csi t, j, k, nz = 0, m, n, *Ap, *Ai, *Cp, *Ci ;
00006     double *Cx, *Ax ;
00007     cs *C ;
00008     if (!CS_CSC (A)) return (NULL) ; /* check inputs */
00009     m = A->m ; n = A->n ; Ap = A->p ; Ai = A->i ; Ax = A->x ;
00010     C = cs_spalloc (m, n, Ap [n], values && Ax != NULL, 0) ; /* alloc result */
00011     if (!C) return (cs_done (C, NULL, NULL, 0)) ; /* out of memory */
00012     Cp = C->p ; Ci = C->i ; Cx = C->x ;
00013     for (k = 0 ; k < n ; k++)
00014     {
00015         Cp [k] = nz ; /* column k of C is column q[k] of A */
00016         j = q [k] : k ;
00017         for (t = Ap [j] ; t < Ap [j+1] ; t++)
00018         {
00019             if (Cx) Cx [nz] = Ax [t] ; /* row i of A is row pinv[i] of C */
00020             Ci [nz++] = pinv ? (pinv [Ai [t]]) : Ai [t] ;
00021         }
00022     }
00023     Cp [n] = nz ; /* finalize the last column of C */
00024     return (cs_done (C, NULL, NULL, 1)) ;
00025 }

```

## 4.99 csparse/Source/cs\_pinv.c File Reference

```
#include "cs.h"
```

### Functions

- [csi \\* cs\\_pinv \(csi const \\*p, csi n\)](#)

### 4.99.1 Function Documentation

#### 4.99.1.1 cs\_pinv()

```

csi * cs_pinv (
    csi const * p,
    csi n )

```

Definition at line 3 of file [cs\\_pinv.c](#).

## 4.100 cs\_pinv.c

[Go to the documentation of this file.](#)

```

00001 #include "cs.h"
00002 /* pinv = p', or p = pinv' */
00003 csi *cs_pinv (csi const *p, csi n)
00004 {
00005     csi k, *pinv ;
00006     if (!p) return (NULL) ; /* p = NULL denotes identity */
00007     pinv = cs_malloc (n, sizeof (csi)) ; /* allocate result */
00008     if (!pinv) return (NULL) ; /* out of memory */
00009     for (k = 0 ; k < n ; k++) pinv [p [k]] = k ; /* invert the permutation */
00010     return (pinv) ; /* return result */
00011 }

```

## 4.101 csparse/Source/cs\_post.c File Reference

```
#include "cs.h"
```

### Functions

- `csi * cs_post` (const `csi` \*parent, `csi` n)

### 4.101.1 Function Documentation

#### 4.101.1.1 cs\_post()

```
csi * cs_post (
    const csi * parent,
    csi n )
```

Definition at line 3 of file `cs_post.c`.

## 4.102 cs\_post.c

[Go to the documentation of this file.](#)

```
00001 #include "cs.h"
00002 /* post order a forest */
00003 csi *cs_post (const csi *parent, csi n)
00004 {
00005     csi j, k = 0, *post, *w, *head, *next, *stack ;
00006     if (!parent) return (NULL) ; /* check inputs */
00007     post = cs_malloc (n, sizeof (csi)) ; /* allocate result */
00008     w = cs_malloc (3*n, sizeof (csi)) ; /* get workspace */
00009     if (!w || !post) return (cs_idone (post, NULL, w, 0)) ;
00010     head = w ; next = w + n ; stack = w + 2*n ;
00011     for (j = 0 ; j < n ; j++) head [j] = -1 ; /* empty linked lists */
00012     for (j = n-1 ; j >= 0 ; j--) /* traverse nodes in reverse order*/
00013     {
00014         if (parent [j] == -1) continue ; /* j is a root */
00015         next [j] = head [parent [j]] ; /* add j to list of its parent */
00016         head [parent [j]] = j ;
00017     }
00018     for (j = 0 ; j < n ; j++)
00019     {
00020         if (parent [j] != -1) continue ; /* skip j if it is not a root */
00021         k = cs_tdfs (j, k, head, next, post, stack) ;
00022     }
00023     return (cs_idone (post, NULL, w, 1)) ; /* success; free w, return post */
00024 }
```

## 4.103 csparse/Source/cs\_print.c File Reference

```
#include "cs.h"
```

## Functions

- `csi cs_print` (const `cs` \*A, `csi` brief)

### 4.103.1 Function Documentation

#### 4.103.1.1 cs\_print()

```
csi cs_print (
    const cs * A,
    csi brief )
```

Definition at line 3 of file `cs_print.c`.

## 4.104 cs\_print.c

[Go to the documentation of this file.](#)

```
00001 #include "cs.h"
00002 /* print a sparse matrix; use %g for integers to avoid differences with csi */
00003 csi cs_print (const cs *A, csi brief)
00004 {
00005     csi p, j, m, n, nzmax, nz, *Ap, *Ai ;
00006     double *Ax ;
00007     if (!A) { printf ("(null)\n") ; return (0) ; }
00008     m = A->m ; n = A->n ; Ap = A->p ; Ai = A->i ; Ax = A->x ;
00009     nzmax = A->nzmax ; nz = A->nz ;
00010     printf ("CSparse Version %d.%d.%d, %s.  %s\n", CS_VER, CS_SUBVER,
00011           CS_SUBSUB, CS_DATE, CS_COPYRIGHT) ;
00012     if (nz < 0)
00013     {
00014         printf ("%g-by-%g, nzmax: %g nnz: %g, 1-norm: %g\n", (double) m,
00015               (double) n, (double) nzmax, (double) (Ap [n]), cs_norm (A)) ;
00016         for (j = 0 ; j < n ; j++)
00017         {
00018             printf ("    col %g : locations %g to %g\n", (double) j,
00019                   (double) (Ap [j]), (double) (Ap [j+1]-1)) ;
00020             for (p = Ap [j] ; p < Ap [j+1] ; p++)
00021             {
00022                 printf ("        %g : %g\n", (double) (Ai [p]), Ax ? Ax [p] : 1) ;
00023                 if (brief && p > 20) { printf ("    ...\n") ; return (1) ; }
00024             }
00025         }
00026     }
00027     else
00028     {
00029         printf ("triplet: %g-by-%g, nzmax: %g nnz: %g\n", (double) m,
00030               (double) n, (double) nzmax, (double) nz) ;
00031         for (p = 0 ; p < nz ; p++)
00032         {
00033             printf ("    %g %g : %g\n", (double) (Ai [p]), (double) (Ap [p]),
00034                   Ax ? Ax [p] : 1) ;
00035             if (brief && p > 20) { printf ("    ...\n") ; return (1) ; }
00036         }
00037     }
00038     return (1) ;
00039 }
```

## 4.105 csparse/Source/cs\_pvec.c File Reference

```
#include "cs.h"
```

## Functions

- `csi cs_pvec` (const `csi` \*p, const double \*b, double \*x, `csi` n)

### 4.105.1 Function Documentation

#### 4.105.1.1 cs\_pvec()

```
csi cs_pvec (
    const csi * p,
    const double * b,
    double * x,
    csi n )
```

Definition at line 3 of file `cs_pvec.c`.

## 4.106 cs\_pvec.c

[Go to the documentation of this file.](#)

```
00001 #include "cs.h"
00002 /* x = b(p), for dense vectors x and b; p=NULL denotes identity */
00003 csi cs_pvec (const csi *p, const double *b, double *x, csi n)
00004 {
00005     csi k ;
00006     if (!x || !b) return (0) ;
00007     for (k = 0 ; k < n ; k++) x [k] = b [p ? p [k] : k] ;
00008     return (1) ;
00009 }
```

## 4.107 csparse/Source/cs\_qr.c File Reference

```
#include "cs.h"
```

## Functions

- `csn * cs_qr` (const `cs` \*A, const `css` \*S)

### 4.107.1 Function Documentation

## 4.107.1.1 cs\_qr()

```
csn * cs_qr (
    const cs * A,
    const css * S )
```

Definition at line 3 of file [cs\\_qr.c](#).

## 4.108 cs\_qr.c

[Go to the documentation of this file.](#)

```
00001 #include "cs.h"
00002 /* sparse QR factorization [V,beta,pinv,R] = qr (A) */
00003 csn *cs_qr (const cs *A, const css *S)
00004 {
00005     double *Rx, *Vx, *Ax, *x, *Beta ;
00006     csi i, k, p, m, n, vnz, pl, top, m2, len, col, rnz, *s, *leftmost, *Ap, *Ai,
00007         *parent, *Rp, *Ri, *Vp, *Vi, *w, *pinv, *q ;
00008     cs *R, *V ;
00009     csn *N ;
00010     if (!CS_CSC (A) || !S) return (NULL) ;
00011     m = A->m ; n = A->n ; Ap = A->p ; Ai = A->i ; Ax = A->x ;
00012     q = S->q ; parent = S->parent ; pinv = S->pinv ; m2 = S->m2 ;
00013     vnz = S->lnz ; rnz = S->unz ; leftmost = S->leftmost ;
00014     w = cs_malloc (m2+n, sizeof (csi)) ; /* get csi workspace */
00015     x = cs_malloc (m2, sizeof (double)) ; /* get double workspace */
00016     N = cs_malloc (1, sizeof (csn)) ; /* allocate result */
00017     if (!w || !x || !N) return (cs_ndone (N, NULL, w, x, 0)) ;
00018     s = w + m2 ;
00019     for (k = 0 ; k < m2 ; k++) x [k] = 0 ; /* clear workspace x */
00020     N->L = V = cs_spalloc (m2, n, vnz, 1, 0) ; /* allocate result V */
00021     N->U = R = cs_spalloc (m2, n, rnz, 1, 0) ; /* allocate result R */
00022     N->B = Beta = cs_malloc (n, sizeof (double)) ; /* allocate result Beta */
00023     if (!R || !V || !Beta) return (cs_ndone (N, NULL, w, x, 0)) ;
00024     Rp = R->p ; Ri = R->i ; Rx = R->x ;
00025     Vp = V->p ; Vi = V->i ; Vx = V->x ;
00026     for (i = 0 ; i < m2 ; i++) w [i] = -1 ; /* clear w, to mark nodes */
00027     rnz = 0 ; vnz = 0 ;
00028     for (k = 0 ; k < n ; k++) /* compute V and R */
00029     {
00030         Rp [k] = rnz ; /* R(:,k) starts here */
00031         Vp [k] = pl = vnz ; /* V(:,k) starts here */
00032         w [k] = k ; /* add V(k,k) to pattern of V */
00033         Vi [vnz++] = k ;
00034         top = n ;
00035         col = q ? q [k] : k ;
00036         for (p = Ap [col] ; p < Ap [col+1] ; p++) /* find R(:,k) pattern */
00037         {
00038             i = leftmost [Ai [p]] ; /* i = min(find(A(i,q))) */
00039             for (len = 0 ; w [i] != k ; i = parent [i]) /* traverse up to k */
00040             {
00041                 s [len++] = i ;
00042                 w [i] = k ;
00043             }
00044             while (len > 0) s [--top] = s [--len] ; /* push path on stack */
00045             i = pinv [Ai [p]] ; /* i = permuted row of A(:,col) */
00046             x [i] = Ax [p] ; /* x (i) = A(:,col) */
00047             if (i > k && w [i] < k) /* pattern of V(:,k) = x (k+1:m) */
00048             {
00049                 Vi [vnz++] = i ; /* add i to pattern of V(:,k) */
00050                 w [i] = k ;
00051             }
00052         }
00053         for (p = top ; p < n ; p++) /* for each i in pattern of R(:,k) */
00054         {
00055             i = s [p] ; /* R(i,k) is nonzero */
00056             cs_happly (V, i, Beta [i], x) ; /* apply (V(i),Beta(i)) to x */
00057             Ri [rnz] = i ; /* R(i,k) = x(i) */
00058             Rx [rnz++] = x [i] ;
00059             x [i] = 0 ;
00060             if (parent [i] == k) vnz = cs_scatter (V, i, 0, w, NULL, k, V, vnz) ;
00061         }
00062         for (p = pl ; p < vnz ; p++) /* gather V(:,k) = x */
00063         {
00064             Vx [p] = x [Vi [p]] ;
00065             x [Vi [p]] = 0 ;
00066         }
```

```

00067         Ri [rnz] = k ;                               /* R(k,k) = norm (x) */
00068         Rx [rnz++] = cs_house (Vx+pl, Beta+k, vnz-pl) ; /* [v,beta]=house(x) */
00069     }
00070     Rp [n] = rnz ;                                     /* finalize R */
00071     Vp [n] = vnz ;                                     /* finalize V */
00072     return (cs_ndone (N, NULL, w, x, 1)) ; /* success */
00073 }

```

## 4.109 csparse/Source/cs\_qrsol.c File Reference

```
#include "cs.h"
```

### Functions

- [csi cs\\_qrsol](#) (csi order, const cs \*A, double \*b)

### 4.109.1 Function Documentation

#### 4.109.1.1 cs\_qrsol()

```

csi cs_qrsol (
    csi order,
    const cs * A,
    double * b )

```

Definition at line 3 of file [cs\\_qrsol.c](#).

## 4.110 cs\_qrsol.c

[Go to the documentation of this file.](#)

```

00001 #include "cs.h"
00002 /* x=A\b where A can be rectangular; b overwritten with solution */
00003 csi cs_qrsol (csi order, const cs *A, double *b)
00004 {
00005     double *x ;
00006     css *S ;
00007     csn *N ;
00008     cs *AT = NULL ;
00009     csi k, m, n, ok ;
00010     if (!CS_CSC (A) || !b) return (0) ; /* check inputs */
00011     n = A->n ;
00012     m = A->m ;
00013     if (m >= n)
00014     {
00015         S = cs_sqr (order, A, 1) ; /* ordering and symbolic analysis */
00016         N = cs_qr (A, S) ; /* numeric QR factorization */
00017         x = cs_calloc (S ? S->m2 : 1, sizeof (double)) ; /* get workspace */
00018         ok = (S && N && x) ;
00019         if (ok)
00020         {
00021             cs_ipvec (S->pinv, b, x, m) ; /* x(0:m-1) = b(p(0:m-1)) */
00022             for (k = 0 ; k < n ; k++) /* apply Householder refl. to x */
00023             {
00024                 cs_happly (N->L, k, N->B [k], x) ;
00025             }
00026             cs_usolve (N->U, x) ; /* x = R\ x */

```

```

00027         cs_ipvec (S->q, x, b, n) ;          /* b(q(0:n-1)) = x(0:n-1) */
00028     }
00029 }
00030 else
00031 {
00032     AT = cs_transpose (A, 1) ;                /* Ax=b is underdetermined */
00033     S = cs_sqr (order, AT, 1) ;               /* ordering and symbolic analysis */
00034     N = cs_qr (AT, S) ;                      /* numeric QR factorization of A' */
00035     x = cs_calloc (S ? S->m2 : 1, sizeof (double)) ; /* get workspace */
00036     ok = (AT && S && N && x) ;
00037     if (ok)
00038     {
00039         cs_pvec (S->q, b, x, m) ;             /* x(q(0:m-1)) = b(0:m-1) */
00040         cs_utsolve (N->U, x) ;                /* x = R'\x */
00041         for (k = m-1 ; k >= 0 ; k--)          /* apply Householder refl. to x */
00042         {
00043             cs_happly (N->L, k, N->B [k], x) ;
00044         }
00045         cs_pvec (S->pinv, x, b, n) ;          /* b(0:n-1) = x(p(0:n-1)) */
00046     }
00047 }
00048 cs_free (x) ;
00049 cs_sfree (S) ;
00050 cs_nfree (N) ;
00051 cs_spfree (AT) ;
00052 return (ok) ;
00053 }

```

## 4.111 csparse/Source/cs\_randperm.c File Reference

```
#include "cs.h"
```

### Functions

- `csi * cs_randperm (csi n, csi seed)`

### 4.111.1 Function Documentation

#### 4.111.1.1 cs\_randperm()

```

csi * cs_randperm (
    csi n,
    csi seed )

```

Definition at line 5 of file `cs_randperm.c`.

## 4.112 cs\_randperm.c

[Go to the documentation of this file.](#)

```
00001 #include "cs.h"
00002 /* return a random permutation vector, the identity perm, or p = n-1:-1:0.
00003  * seed = -1 means p = n-1:-1:0. seed = 0 means p = identity. otherwise
00004  * p = random permutation. */
00005 csi *cs_randperm (csi n, csi seed)
00006 {
00007     csi *p, k, j, t ;
00008     if (seed == 0) return (NULL) ; /* return p = NULL (identity) */
00009     p = cs_malloc (n, sizeof (csi)) ; /* allocate result */
00010     if (!p) return (NULL) ; /* out of memory */
00011     for (k = 0 ; k < n ; k++) p [k] = n-k-1 ;
00012     if (seed == -1) return (p) ; /* return reverse permutation */
00013     srand (seed) ; /* get new random number seed */
00014     for (k = 0 ; k < n ; k++)
00015     {
00016         j = k + (rand ( ) % (n-k)) ; /* j = rand integer in range k to n-1 */
00017         t = p [j] ; /* swap p[k] and p[j] */
00018         p [j] = p [k] ;
00019         p [k] = t ;
00020     }
00021     return (p) ;
00022 }
```

## 4.113 csparse/Source/cs\_reach.c File Reference

```
#include "cs.h"
```

### Functions

- [csi cs\\_reach](#) (cs \*G, const cs \*B, csi k, csi \*xi, const csi \*pinv)

### 4.113.1 Function Documentation

#### 4.113.1.1 cs\_reach()

```
csi cs_reach (
    cs * G,
    const cs * B,
    csi k,
    csi * xi,
    const csi * pinv )
```

Definition at line 4 of file [cs\\_reach.c](#).



## 4.114 cs\_reach.c

[Go to the documentation of this file.](#)

```

00001 #include "cs.h"
00002 /* xi [top...n-1] = nodes reachable from graph of G*P' via nodes in B(:,k).
00003  * xi [n...2n-1] used as workspace */
00004 csi cs_reach (cs *G, const cs *B, csi k, csi *xi, const csi *pinv)
00005 {
00006     csi p, n, top, *Bp, *Bi, *Gp ;
00007     if (!CS_CSC (G) || !CS_CSC (B) || !xi) return (-1) ;    /* check inputs */
00008     n = G->n ; Bp = B->p ; Bi = B->i ; Gp = G->p ;
00009     top = n ;
00010     for (p = Bp [k] ; p < Bp [k+1] ; p++)
00011     {
00012         if (!CS_MARKED (Gp, Bi [p]))    /* start a dfs at unmarked node i */
00013         {
00014             top = cs_dfs (Bi [p], G, top, xi, xi+n, pinv) ;
00015         }
00016     }
00017     for (p = top ; p < n ; p++) CS_MARK (Gp, xi [p]) ;    /* restore G */
00018     return (top) ;
00019 }

```

## 4.115 csparse/Source/cs\_scatter.c File Reference

```
#include "cs.h"
```

### Functions

- `csi cs_scatter` (const cs \*A, csi j, double beta, csi \*w, double \*x, csi mark, cs \*C, csi nz)

### 4.115.1 Function Documentation

#### 4.115.1.1 cs\_scatter()

```

csi cs_scatter (
    const cs * A,
    csi j,
    double beta,
    csi * w,
    double * x,
    csi mark,
    cs * C,
    csi nz )

```

Definition at line 3 of file [cs\\_scatter.c](#).

## 4.116 cs\_scatter.c

[Go to the documentation of this file.](#)

```
00001 #include "cs.h"
00002 /* x = x + beta * A(:,j), where x is a dense vector and A(:,j) is sparse */
00003 csi cs_scatter (const cs *A, csi j, double beta, csi *w, double *x, csi mark,
00004                cs *C, csi nz)
00005 {
00006     csi i, p, *Ap, *Ai, *Ci ;
00007     double *Ax ;
00008     if (!CS_CSC (A) || !w || !CS_CSC (C)) return (-1) ;    /* check inputs */
00009     Ap = A->p ; Ai = A->i ; Ax = A->x ; Ci = C->i ;
00010     for (p = Ap [j] ; p < Ap [j+1] ; p++)
00011     {
00012         i = Ai [p] ;                                /* A(i,j) is nonzero */
00013         if (w [i] < mark)
00014         {
00015             w [i] = mark ;                            /* i is new entry in column j */
00016             Ci [nz++] = i ;                            /* add i to pattern of C(:,j) */
00017             if (x) x [i] = beta * Ax [p] ;            /* x(i) = beta*A(i,j) */
00018         }
00019         else if (x) x [i] += beta * Ax [p] ;          /* i exists in C(:,j) already */
00020     }
00021     return (nz) ;
00022 }
```

## 4.117 csparse/Source/cs\_scc.c File Reference

```
#include "cs.h"
```

### Functions

- [csd](#) \* [cs\\_scc](#) ([cs](#) \*A)

### 4.117.1 Function Documentation

#### 4.117.1.1 cs\_scc()

```
csd * cs_scc (
    cs * A )
```

Definition at line 3 of file [cs\\_scc.c](#).

## 4.118 cs\_scc.c

Go to the documentation of this file.

```

00001 #include "cs.h"
00002 /* find the strongly connected components of a square matrix */
00003 csd *cs_scc (cs *A) /* matrix A temporarily modified, then restored */
00004 {
00005     csi n, i, k, b, nb = 0, top, *xi, *pstack, *p, *r, *Ap, *ATp, *rcopy, *Blk ;
00006     cs *AT ;
00007     csd *D ;
00008     if (!CS_CSC (A)) return (NULL) ; /* check inputs */
00009     n = A->n ; Ap = A->p ;
00010     D = cs_dalloc (n, 0) ; /* allocate result */
00011     AT = cs_transpose (A, 0) ; /* AT = A' */
00012     xi = cs_malloc (2*n+1, sizeof (csi)) ; /* get workspace */
00013     if (!D || !AT || !xi) return (cs_ddone (D, AT, xi, 0)) ;
00014     Blk = xi ; rcopy = pstack = xi + n ;
00015     p = D->p ; r = D->r ; ATp = AT->p ;
00016     top = n ;
00017     for (i = 0 ; i < n ; i++) /* first dfs(A) to find finish times (xi) */
00018     {
00019         if (!CS_MARKED (Ap, i)) top = cs_dfs (i, A, top, xi, pstack, NULL) ;
00020     }
00021     for (i = 0 ; i < n ; i++) CS_MARK (Ap, i) ; /* restore A; unmark all nodes */
00022     top = n ;
00023     nb = n ;
00024     for (k = 0 ; k < n ; k++) /* dfs(A') to find strongly connected comp */
00025     {
00026         i = xi [k] ; /* get i in reverse order of finish times */
00027         if (CS_MARKED (ATp, i)) continue ; /* skip node i if already ordered */
00028         r [nb--] = top ; /* node i is the start of a component in p */
00029         top = cs_dfs (i, AT, top, p, pstack, NULL) ;
00030     }
00031     r [nb] = 0 ; /* first block starts at zero; shift r up */
00032     for (k = nb ; k <= n ; k++) r [k-nb] = r [k] ;
00033     D->nb = nb = n-nb ; /* nb = # of strongly connected components */
00034     for (b = 0 ; b < nb ; b++) /* sort each block in natural order */
00035     {
00036         for (k = r [b] ; k < r [b+1] ; k++) Blk [p [k]] = b ;
00037     }
00038     for (b = 0 ; b <= nb ; b++) rcopy [b] = r [b] ;
00039     for (i = 0 ; i < n ; i++) p [rcopy [Blk [i]]++] = i ;
00040     return (cs_ddone (D, AT, xi, 1)) ;
00041 }

```

## 4.119 csparse/Source/cs\_schol.c File Reference

```
#include "cs.h"
```

### Functions

- `css * cs_schol (csi order, const cs *A)`

### 4.119.1 Function Documentation

#### 4.119.1.1 cs\_schol()

```

css * cs_schol (
    csi order,
    const cs * A )

```

Definition at line 3 of file `cs_schol.c`.

## 4.120 cs\_schol.c

[Go to the documentation of this file.](#)

```
00001 #include "cs.h"
00002 /* ordering and symbolic analysis for a Cholesky factorization */
00003 css *cs_schol (csi order, const cs *A)
00004 {
00005     csi n, *C, *post, *P ;
00006     cs *C ;
00007     css *S ;
00008     if (!CS_CSC (A)) return (NULL) ; /* check inputs */
00009     n = A->n ;
00010     S = cs_calloc (1, sizeof (css)) ; /* allocate result S */
00011     if (!S) return (NULL) ; /* out of memory */
00012     P = cs_amd (order, A) ; /* P = amd(A+A'), or natural */
00013     S->pinv = cs_pinv (P, n) ; /* find inverse permutation */
00014     cs_free (P) ;
00015     if (order && !S->pinv) return (cs_sfree (S)) ;
00016     C = cs_symperm (A, S->pinv, 0) ; /* C = spones(triu(A(P,P))) */
00017     S->parent = cs_etree (C, 0) ; /* find etree of C */
00018     post = cs_post (S->parent, n) ; /* postorder the etree */
00019     c = cs_counts (C, S->parent, post, 0) ; /* find column counts of chol(C) */
00020     cs_free (post) ;
00021     cs_spfree (C) ;
00022     S->cp = cs_malloc (n+1, sizeof (csi)) ; /* allocate result S->cp */
00023     S->unz = S->lnz = cs_cumsum (S->cp, c, n) ; /* find column pointers for L */
00024     cs_free (c) ;
00025     return ((S->lnz >= 0) ? S : cs_sfree (S)) ;
00026 }
```

## 4.121 csparse/Source/cs\_spsolve.c File Reference

```
#include "cs.h"
```

### Functions

- [csi cs\\_spsolve](#) (cs \*G, const cs \*B, csi k, csi \*xi, double \*x, const csi \*pinv, csi lo)

### 4.121.1 Function Documentation

#### 4.121.1.1 cs\_spsolve()

```
csi cs_spsolve (
    cs * G,
    const cs * B,
    csi k,
    csi * xi,
    double * x,
    const csi * pinv,
    csi lo )
```

Definition at line 3 of file [cs\\_spsolve.c](#).

## 4.122 cs\_spsolve.c

[Go to the documentation of this file.](#)

```

00001 #include "cs.h"
00002 /* solve Gx=b(:,k), where G is either upper (lo=0) or lower (lo=1) triangular */
00003 csi cs_spsolve (csi *G, const csi *B, csi k, csi *xi, double *x, const csi *pinv,
00004                csi lo)
00005 {
00006     csi j, J, p, q, px, top, n, *Gp, *Gi, *Bp, *Bi ;
00007     double *Gx, *Bx ;
00008     if (!CS_CSC (G) || !CS_CSC (B) || !xi || !x) return (-1) ;
00009     Gp = G->p ; Gi = G->i ; Gx = G->x ; n = G->n ;
00010     Bp = B->p ; Bi = B->i ; Bx = B->x ;
00011     top = cs_reach (G, B, k, xi, pinv) ; /* xi[top..n-1]=Reach(B(:,k)) */
00012     for (p = top ; p < n ; p++) x [xi [p]] = 0 ; /* clear x */
00013     for (p = Bp [k] ; p < Bp [k+1] ; p++) x [Bi [p]] = Bx [p] ; /* scatter B */
00014     for (px = top ; px < n ; px++)
00015     {
00016         j = xi [px] ; /* x(j) is nonzero */
00017         J = pinv ? (pinv [j]) : j ; /* j maps to col J of G */
00018         if (J < 0) continue ; /* column J is empty */
00019         x [j] /= Gx [lo ? (Gp [J]) : (Gp [J+1]-1)] ; /* x(j) /= G(j,j) */
00020         p = lo ? (Gp [J+1] : (Gp [J]) ; /* lo: L(j,j) 1st entry */
00021         q = lo ? (Gp [J+1] : (Gp [J+1]-1) ; /* up: U(j,j) last entry */
00022         for ( ; p < q ; p++)
00023         {
00024             x [Gi [p]] -= Gx [p] * x [j] ; /* x(i) -= G(i,j) * x(j) */
00025         }
00026     }
00027     return (top) ; /* return top of stack */
00028 }

```

## 4.123 csparse/Source/cs\_sqr.c File Reference

```
#include "cs.h"
```

### Functions

- [css \\* cs\\_sqr](#) (csi order, const csi \*A, csi qr)

### 4.123.1 Function Documentation

#### 4.123.1.1 cs\_sqr()

```

css * cs_sqr (
    csi order,
    const csi * A,
    csi qr )

```

Definition at line 60 of file [cs\\_sqr.c](#).

## 4.124 cs\_sqr.c

[Go to the documentation of this file.](#)

```

00001 #include "cs.h"
00002 /* compute nnz(V) = S->lnz, S->pinv, S->leftmost, S->m2 from A and S->parent */
00003 static csi cs_vcount (const cs *A, css *S)
00004 {
00005     csi i, k, p, pa, n = A->n, m = A->m, *Ap = A->p, *Ai = A->i, *next, *head,
00006     *tail, *nque, *pinv, *leftmost, *w, *parent = S->parent ;
00007     S->pinv = pinv = cs_malloc (m+n, sizeof (csi)) ; /* allocate pinv, */
00008     S->leftmost = leftmost = cs_malloc (m, sizeof (csi)) ; /* and leftmost */
00009     w = cs_malloc (m+3*n, sizeof (csi)) ; /* get workspace */
00010     if (!pinv || !w || !leftmost)
00011     {
00012         cs_free (w) ; /* pinv and leftmost freed later */
00013         return (0) ; /* out of memory */
00014     }
00015     next = w ; head = w + m ; tail = w + m + n ; nque = w + m + 2*n ;
00016     for (k = 0 ; k < n ; k++) head [k] = -1 ; /* queue k is empty */
00017     for (k = 0 ; k < n ; k++) tail [k] = -1 ;
00018     for (k = 0 ; k < n ; k++) nque [k] = 0 ;
00019     for (i = 0 ; i < m ; i++) leftmost [i] = -1 ;
00020     for (k = n-1 ; k >= 0 ; k--)
00021     {
00022         for (p = Ap [k] ; p < Ap [k+1] ; p++)
00023         {
00024             leftmost [Ai [p]] = k ; /* leftmost[i] = min(find(A(i,:)))* */
00025         }
00026     }
00027     for (i = m-1 ; i >= 0 ; i--) /* scan rows in reverse order */
00028     {
00029         pinv [i] = -1 ; /* row i is not yet ordered */
00030         k = leftmost [i] ;
00031         if (k == -1) continue ; /* row i is empty */
00032         if (nque [k]++ == 0) tail [k] = i ; /* first row in queue k */
00033         next [i] = head [k] ; /* put i at head of queue k */
00034         head [k] = i ;
00035     }
00036     S->lnz = 0 ;
00037     S->m2 = m ;
00038     for (k = 0 ; k < n ; k++) /* find row permutation and nnz(V) */
00039     {
00040         i = head [k] ; /* remove row i from queue k */
00041         S->lnz++ ; /* count V(k,k) as nonzero */
00042         if (i < 0) i = S->m2++ ; /* add a fictitious row */
00043         pinv [i] = k ; /* associate row i with V(:,k) */
00044         if (--nque [k] <= 0) continue ; /* skip if V(k+1:m,k) is empty */
00045         S->lnz += nque [k] ; /* nque [k] is nnz (V(k+1:m,k)) */
00046         if ((pa = parent [k]) != -1) /* move all rows to parent of k */
00047         {
00048             if (nque [pa] == 0) tail [pa] = tail [k] ;
00049             next [tail [k]] = head [pa] ;
00050             head [pa] = next [i] ;
00051             nque [pa] += nque [k] ;
00052         }
00053     }
00054     for (i = 0 ; i < m ; i++) if (pinv [i] < 0) pinv [i] = k++ ;
00055     cs_free (w) ;
00056     return (1) ;
00057 }
00058
00059 /* symbolic ordering and analysis for QR or LU */
00060 css *cs_sqr (csi order, const cs *A, csi qr)
00061 {
00062     csi n, k, ok = 1, *post ;
00063     css *S ;
00064     if (!CS_CSC (A)) return (NULL) ; /* check inputs */
00065     n = A->n ;
00066     S = cs_calloc (1, sizeof (css)) ; /* allocate result S */
00067     if (!S) return (NULL) ; /* out of memory */
00068     S->q = cs_amd (order, A) ; /* fill-reducing ordering */
00069     if (order && !S->q) return (cs_sfree (S)) ;
00070     if (qr) /* QR symbolic analysis */
00071     {
00072         cs *C = order ? cs_permute (A, NULL, S->q, 0) : ((cs *) A) ;
00073         S->parent = cs_etree (C, 1) ; /* etree of C'*C, where C=A(:,q) */
00074         post = cs_post (S->parent, n) ;
00075         S->cp = cs_counts (C, S->parent, post, 1) ; /* col counts chol(C'*C) */
00076         cs_free (post) ;
00077         ok = C && S->parent && S->cp && cs_vcount (C, S) ;
00078         if (ok) for (S->unz = 0, k = 0 ; k < n ; k++) S->unz += S->cp [k] ;
00079         if (order) cs_spfree (C) ;
00080     }
00081     else
00082     {

```

```

00083         S->unz = 4*(A->p [n]) + n ;           /* for LU factorization only, */
00084         S->lnz = S->unz ;                       /* guess nnz(L) and nnz(U) */
00085     }
00086     return (ok ? S : cs_sfree (S)) ;           /* return result S */
00087 }

```

## 4.125 csparse/Source/cs\_symperm.c File Reference

```
#include "cs.h"
```

### Functions

- `cs * cs_symperm` (const `cs` \*A, const `csi` \*pinv, `csi` values)

#### 4.125.1 Function Documentation

##### 4.125.1.1 cs\_symperm()

```

cs * cs_symperm (
    const cs * A,
    const csi * pinv,
    csi values )

```

Definition at line 3 of file `cs_symperm.c`.

## 4.126 cs\_symperm.c

[Go to the documentation of this file.](#)

```

00001 #include "cs.h"
00002 /* C = A(p,p) where A and C are symmetric the upper part stored; pinv not p */
00003 cs *cs_symperm (const cs *A, const csi *pinv, csi values)
00004 {
00005     csi i, j, p, q, i2, j2, n, *Ap, *Ai, *Cp, *Ci, *w ;
00006     double *Cx, *Ax ;
00007     cs *C ;
00008     if (!CS_CSC (A)) return (NULL) ;           /* check inputs */
00009     n = A->n ; Ap = A->p ; Ai = A->i ; Ax = A->x ;
00010     C = cs_spalloc (n, n, Ap [n], values && (Ax != NULL), 0) ; /* alloc result */
00011     w = cs_calloc (n, sizeof (csi)) ;           /* get workspace */
00012     if (!C || !w) return (cs_done (C, w, NULL, 0)) ; /* out of memory */
00013     Cp = C->p ; Ci = C->i ; Cx = C->x ;
00014     for (j = 0 ; j < n ; j++)                  /* count entries in each column of C */
00015     {
00016         j2 = pinv ? pinv [j] : j ;             /* column j of A is column j2 of C */
00017         for (p = Ap [j] ; p < Ap [j+1] ; p++)
00018         {
00019             i = Ai [p] ;
00020             if (i > j) continue ;               /* skip lower triangular part of A */
00021             i2 = pinv ? pinv [i] : i ;          /* row i of A is row i2 of C */
00022             w [CS_MAX (i2, j2)]++ ;           /* column count of C */
00023         }
00024     }
00025     cs_cumsum (Cp, w, n) ;                      /* compute column pointers of C */
00026     for (j = 0 ; j < n ; j++)
00027     {
00028         j2 = pinv ? pinv [j] : j ;             /* column j of A is column j2 of C */

```

```

00029         for (p = Ap [j] ; p < Ap [j+1] ; p++)
00030         {
00031             i = Ai [p] ;
00032             if (i > j) continue ;          /* skip lower triangular part of A*/
00033             i2 = pinv ? pinv [i] : i ;    /* row i of A is row i2 of C */
00034             Ci [q = w [CS_MAX (i2, j2)]++] = CS_MIN (i2, j2) ;
00035             if (Cx) Cx [q] = Ax [p] ;
00036         }
00037     }
00038     return (cs_done (C, w, NULL, 1)) ; /* success; free workspace, return C */
00039 }

```

## 4.127 csparse/Source/cs\_tdfs.c File Reference

```
#include "cs.h"
```

### Functions

- [csi cs\\_tdfs](#) ([csi](#) j, [csi](#) k, [csi](#) \*head, const [csi](#) \*next, [csi](#) \*post, [csi](#) \*stack)

### 4.127.1 Function Documentation

#### 4.127.1.1 cs\_tdfs()

```

csi cs_tdfs (
    csi j,
    csi k,
    csi * head,
    const csi * next,
    csi * post,
    csi * stack )

```

Definition at line 3 of file [cs\\_tdfs.c](#).

## 4.128 cs\_tdfs.c

[Go to the documentation of this file.](#)

```

00001 #include "cs.h"
00002 /* depth-first search and postorder of a tree rooted at node j */
00003 csi cs_tdfs (csi j, csi k, csi *head, const csi *next, csi *post, csi *stack)
00004 {
00005     csi i, p, top = 0 ;
00006     if (!head || !next || !post || !stack) return (-1) ; /* check inputs */
00007     stack [0] = j ; /* place j on the stack */
00008     while (top >= 0) /* while (stack is not empty) */
00009     {
00010         p = stack [top] ; /* p = top of stack */
00011         i = head [p] ; /* i = youngest child of p */
00012         if (i == -1)
00013         {
00014             top-- ; /* p has no unordered children left */
00015             post [k++] = p ; /* node p is the kth postordered node */
00016         }
00017         else
00018         {
00019             head [p] = next [i] ; /* remove i from children of p */
00020             stack [++top] = i ; /* start dfs on child node i */
00021         }
00022     }
00023     return (k) ;
00024 }

```



## 4.129 csparse/Source/cs\_transpose.c File Reference

```
#include "cs.h"
```

### Functions

- `cs * cs_transpose` (const `cs` \*A, `csi` values)

### 4.129.1 Function Documentation

#### 4.129.1.1 cs\_transpose()

```
cs * cs_transpose (
    const cs * A,
    csi values )
```

Definition at line 3 of file `cs_transpose.c`.

## 4.130 cs\_transpose.c

[Go to the documentation of this file.](#)

```
00001 #include "cs.h"
00002 /* C = A' */
00003 cs *cs_transpose (const cs *A, csi values)
00004 {
00005     csi p, q, j, *Cp, *Ci, n, m, *Ap, *Ai, *w ;
00006     double *Cx, *Ax ;
00007     cs *C ;
00008     if (!CS_CSC (A)) return (NULL) ; /* check inputs */
00009     m = A->m ; n = A->n ; Ap = A->p ; Ai = A->i ; Ax = A->x ;
00010     C = cs_spalloc (n, m, Ap [n], values && Ax, 0) ; /* allocate result */
00011     w = cs_calloc (m, sizeof (csi)) ; /* get workspace */
00012     if (!C || !w) return (cs_done (C, w, NULL, 0)) ; /* out of memory */
00013     Cp = C->p ; Ci = C->i ; Cx = C->x ;
00014     for (p = 0 ; p < Ap [n] ; p++) w [Ai [p]]++ ; /* row counts */
00015     cs_cumsum (Cp, w, m) ; /* row pointers */
00016     for (j = 0 ; j < n ; j++)
00017     {
00018         for (p = Ap [j] ; p < Ap [j+1] ; p++)
00019         {
00020             Ci [q = w [Ai [p]]++] = j ; /* place A(i,j) as entry C(j,i) */
00021             if (Cx) Cx [q] = Ax [p] ;
00022         }
00023     }
00024     return (cs_done (C, w, NULL, 1)) ; /* success; free w and return C */
00025 }
```

## 4.131 csparse/Source/cs\_updown.c File Reference

```
#include "cs.h"
```

## Functions

- `csi cs_updown (cs *L, csi sigma, const cs *C, const csi *parent)`

### 4.131.1 Function Documentation

#### 4.131.1.1 cs\_updown()

```
csi cs_updown (
    cs * L,
    csi sigma,
    const cs * C,
    const csi * parent )
```

Definition at line 3 of file [cs\\_updown.c](#).

### 4.132 cs\_updown.c

[Go to the documentation of this file.](#)

```
00001 #include "cs.h"
00002 /* sparse Cholesky update/downdate, L*L' + sigma*w*w' (sigma = +1 or -1) */
00003 csi cs_updown (cs *L, csi sigma, const cs *C, const csi *parent)
00004 {
00005     csi n, p, f, j, *Lp, *Li, *Cp, *Ci ;
00006     double *Lx, *Cx, alpha, beta = 1, delta, gamma, w1, w2, *w, beta2 = 1 ;
00007     if (!CS_CSC (L) || !CS_CSC (C) || !parent) return (0) ; /* check inputs */
00008     Lp = L->p ; Li = L->i ; Lx = L->x ; n = L->n ;
00009     Cp = C->p ; Ci = C->i ; Cx = C->x ;
00010     if ((p = Cp [0]) >= Cp [1]) return (1) ; /* return if C empty */
00011     w = cs_malloc (n, sizeof (double)) ; /* get workspace */
00012     if (!w) return (0) ; /* out of memory */
00013     f = Ci [p] ;
00014     for ( ; p < Cp [1] ; p++) f = CS_MIN (f, Ci [p]) ; /* f = min (find (C)) */
00015     for (j = f ; j != -1 ; j = parent [j]) w [j] = 0 ; /* clear workspace w */
00016     for (p = Cp [0] ; p < Cp [1] ; p++) w [Ci [p]] = Cx [p] ; /* w = C */
00017     for (j = f ; j != -1 ; j = parent [j]) /* walk path f up to root */
00018     {
00019         p = Lp [j] ;
00020         alpha = w [j] / Lx [p] ; /* alpha = w(j) / L(j,j) */
00021         beta2 = beta*beta + sigma*alpha*alpha ;
00022         if (beta2 <= 0) break ; /* not positive definite */
00023         beta2 = sqrt (beta2) ;
00024         delta = (sigma > 0) ? (beta / beta2) : (beta2 / beta) ;
00025         gamma = sigma * alpha / (beta2 * beta) ;
00026         Lx [p] = delta * Lx [p] + ((sigma > 0) ? (gamma * w [j]) : 0) ;
00027         beta = beta2 ;
00028         for (p++ ; p < Lp [j+1] ; p++)
00029         {
00030             w1 = w [Li [p]] ;
00031             w [Li [p]] = w2 = w1 - alpha * Lx [p] ;
00032             Lx [p] = delta * Lx [p] + gamma * ((sigma > 0) ? w1 : w2) ;
00033         }
00034     }
00035     cs_free (w) ;
00036     return (beta2 > 0) ;
00037 }
```

### 4.133 csparse/Source/cs\_usolve.c File Reference

```
#include "cs.h"
```

## Functions

- `csi cs_usolve` (const `cs` \*U, double \*x)

### 4.133.1 Function Documentation

#### 4.133.1.1 cs\_usolve()

```
csi cs_usolve (
    const cs * U,
    double * x )
```

Definition at line 3 of file `cs_usolve.c`.

## 4.134 cs\_usolve.c

[Go to the documentation of this file.](#)

```
00001 #include "cs.h"
00002 /* solve Ux=b where x and b are dense.  x=b on input, solution on output. */
00003 csi cs_usolve (const cs *U, double *x)
00004 {
00005     csi p, j, n, *Up, *Ui ;
00006     double *Ux ;
00007     if (!CS_CSC (U) || !x) return (0) ; /* check inputs */
00008     n = U->n ; Up = U->p ; Ui = U->i ; Ux = U->x ;
00009     for (j = n-1 ; j >= 0 ; j--)
00010     {
00011         x [j] /= Ux [Up [j+1]-1] ;
00012         for (p = Up [j] ; p < Up [j+1]-1 ; p++)
00013         {
00014             x [Ui [p]] -= Ux [p] * x [j] ;
00015         }
00016     }
00017     return (1) ;
00018 }
```

## 4.135 csparse/Source/cs\_util.c File Reference

```
#include "cs.h"
```

## Functions

- `cs` \* `cs_spalloc` (csi m, csi n, csi nzmax, csi values, csi triplet)
- `csi` \* `cs_sprealloc` (cs \*A, csi nzmax)
- `cs` \* `cs_spfree` (cs \*A)
- `csn` \* `cs_nfree` (csn \*N)
- `css` \* `cs_sfree` (css \*S)
- `csd` \* `cs_dalloc` (csi m, csi n)
- `csd` \* `cs_dfree` (csd \*D)
- `cs` \* `cs_done` (cs \*C, void \*w, void \*x, csi ok)
- `csi` \* `cs_idone` (csi \*p, cs \*C, void \*w, csi ok)
- `csn` \* `cs_ndone` (csn \*N, cs \*C, void \*w, void \*x, csi ok)
- `csd` \* `cs_ddone` (csd \*D, cs \*C, void \*w, csi ok)

## 4.135.1 Function Documentation

### 4.135.1.1 `cs_dalloc()`

```
csd * cs_dalloc (
    csi m,
    csi n )
```

Definition at line 66 of file [cs\\_util.c](#).

### 4.135.1.2 `cs_ddone()`

```
csd * cs_ddone (
    csd * D,
    cs * C,
    void * w,
    csi ok )
```

Definition at line 115 of file [cs\\_util.c](#).

### 4.135.1.3 `cs_dfree()`

```
csd * cs_dfree (
    csd * D )
```

Definition at line 79 of file [cs\\_util.c](#).

### 4.135.1.4 `cs_done()`

```
cs * cs_done (
    cs * C,
    void * w,
    void * x,
    csi ok )
```

Definition at line 90 of file [cs\\_util.c](#).

#### 4.135.1.5 cs\_idone()

```
csi * cs_idone (
    csi * p,
    cs * C,
    void * w,
    csi ok )
```

Definition at line 98 of file [cs\\_util.c](#).

#### 4.135.1.6 cs\_ndone()

```
csn * cs_ndone (
    csn * N,
    cs * C,
    void * w,
    void * x,
    csi ok )
```

Definition at line 106 of file [cs\\_util.c](#).

#### 4.135.1.7 cs\_nfree()

```
csn * cs_nfree (
    csn * N )
```

Definition at line 43 of file [cs\\_util.c](#).

#### 4.135.1.8 cs\_sfree()

```
css * cs_sfree (
    css * S )
```

Definition at line 54 of file [cs\\_util.c](#).

#### 4.135.1.9 cs\_spalloc()

```
cs * cs_spalloc (
    csi m,
    csi n,
    csi nzmax,
    csi values,
    csi triplet )
```

Definition at line 3 of file [cs\\_util.c](#).

#### 4.135.1.10 cs\_spfree()

```
cs * cs_spfree (
    cs * A )
```

Definition at line 33 of file [cs\\_util.c](#).

#### 4.135.1.11 cs\_sprealloc()

```
csi cs_sprealloc (
    cs * A,
    csi nzmax )
```

Definition at line 18 of file [cs\\_util.c](#).

### 4.136 cs\_util.c

[Go to the documentation of this file.](#)

```
00001 #include "cs.h"
00002 /* allocate a sparse matrix (triplet form or compressed-column form) */
00003 cs *cs_spalloc (csi m, csi n, csi nzmax, csi values, csi triplet)
00004 {
00005     cs *A = cs_calloc (1, sizeof (cs)) ; /* allocate the cs struct */
00006     if (!A) return (NULL) ; /* out of memory */
00007     A->m = m ; /* define dimensions and nzmax */
00008     A->n = n ;
00009     A->nzmax = nzmax = CS_MAX (nzmax, 1) ;
00010     A->nz = triplet ? 0 : -1 ; /* allocate triplet or comp.col */
00011     A->p = cs_malloc (triplet ? nzmax : n+1, sizeof (csi)) ;
00012     A->i = cs_malloc (nzmax, sizeof (csi)) ;
00013     A->x = values ? cs_malloc (nzmax, sizeof (double)) : NULL ;
00014     return ((!A->p || !A->i || (values && !A->x)) ? cs_spfree (A) : A) ;
00015 }
00016
00017 /* change the max # of entries sparse matrix */
00018 csi cs_sprealloc (cs *A, csi nzmax)
00019 {
00020     csi ok, oki, okj = 1, okx = 1 ;
00021     if (!A) return (0) ;
00022     if (nzmax <= 0) nzmax = (CS_CSC (A)) ? (A->p [A->n]) : A->nz ;
00023     nzmax = CS_MAX (nzmax, 1) ;
00024     A->i = cs_realloc (A->i, nzmax, sizeof (csi), &oki) ;
00025     if (CS_TRIPLET (A)) A->p = cs_realloc (A->p, nzmax, sizeof (csi), &okj) ;
00026     if (A->x) A->x = cs_realloc (A->x, nzmax, sizeof (double), &okx) ;
00027     ok = (oki && okj && okx) ;
00028     if (ok) A->nzmax = nzmax ;
00029     return (ok) ;
00030 }
00031
00032 /* free a sparse matrix */
00033 cs *cs_spfree (cs *A)
00034 {
00035     if (!A) return (NULL) ; /* do nothing if A already NULL */
00036     cs_free (A->p) ;
00037     cs_free (A->i) ;
00038     cs_free (A->x) ;
00039     return ((cs *) cs_free (A)) ; /* free the cs struct and return NULL */
00040 }
00041
00042 /* free a numeric factorization */
00043 csn *cs_nfree (csn *N)
00044 {
00045     if (!N) return (NULL) ; /* do nothing if N already NULL */
00046     cs_spfree (N->L) ;
00047     cs_spfree (N->U) ;
00048     cs_free (N->pinv) ;
00049     cs_free (N->B) ;
00050     return ((csn *) cs_free (N)) ; /* free the csn struct and return NULL */
00051 }
```

```

00051 }
00052
00053 /* free a symbolic factorization */
00054 css *cs_sfree (css *S)
00055 {
00056     if (!S) return (NULL) ;      /* do nothing if S already NULL */
00057     cs_free (S->pinv) ;
00058     cs_free (S->q) ;
00059     cs_free (S->parent) ;
00060     cs_free (S->cp) ;
00061     cs_free (S->leftmost) ;
00062     return ((css *) cs_free (S)) ; /* free the css struct and return NULL */
00063 }
00064
00065 /* allocate a cs_dmperm or cs_scc result */
00066 csd *cs_dalloc (csi m, csi n)
00067 {
00068     csd *D ;
00069     D = cs_calloc (1, sizeof (csd)) ;
00070     if (!D) return (NULL) ;
00071     D->p = cs_malloc (m, sizeof (csi)) ;
00072     D->r = cs_malloc (m+6, sizeof (csi)) ;
00073     D->q = cs_malloc (n, sizeof (csi)) ;
00074     D->s = cs_malloc (n+6, sizeof (csi)) ;
00075     return ((!D->p || !D->r || !D->q || !D->s) ? cs_dfree (D) : D) ;
00076 }
00077
00078 /* free a cs_dmperm or cs_scc result */
00079 csd *cs_dfree (csd *D)
00080 {
00081     if (!D) return (NULL) ;      /* do nothing if D already NULL */
00082     cs_free (D->p) ;
00083     cs_free (D->q) ;
00084     cs_free (D->r) ;
00085     cs_free (D->s) ;
00086     return ((csd *) cs_free (D)) ; /* free the csd struct and return NULL */
00087 }
00088
00089 /* free workspace and return a sparse matrix result */
00090 cs *cs_done (cs *C, void *w, void *x, csi ok)
00091 {
00092     cs_free (w) ;                /* free workspace */
00093     cs_free (x) ;
00094     return (ok ? C : cs_spfree (C)) ; /* return result if OK, else free it */
00095 }
00096
00097 /* free workspace and return csi array result */
00098 csi *cs_idone (csi *p, cs *C, void *w, csi ok)
00099 {
00100     cs_spfree (C) ;              /* free temporary matrix */
00101     cs_free (w) ;                /* free workspace */
00102     return (ok ? p : (csi *) cs_free (p)) ; /* return result, or free it */
00103 }
00104
00105 /* free workspace and return a numeric factorization (Cholesky, LU, or QR) */
00106 csn *cs_ndone (csn *N, cs *C, void *w, void *x, csi ok)
00107 {
00108     cs_spfree (C) ;              /* free temporary matrix */
00109     cs_free (w) ;                /* free workspace */
00110     cs_free (x) ;
00111     return (ok ? N : cs_nfree (N)) ; /* return result if OK, else free it */
00112 }
00113
00114 /* free workspace and return a csd result */
00115 csd *cs_ddone (csd *D, cs *C, void *w, csi ok)
00116 {
00117     cs_spfree (C) ;              /* free temporary matrix */
00118     cs_free (w) ;                /* free workspace */
00119     return (ok ? D : cs_dfree (D)) ; /* return result if OK, else free it */
00120 }

```

## 4.137 csparse/Source/cs\_utsolve.c File Reference

```
#include "cs.h"
```

### Functions

- [csi cs\\_utsolve](#) (const [cs](#) \*U, double \*x)

## 4.137.1 Function Documentation

### 4.137.1.1 cs\_utsolve()

```
csi cs_utsolve (
    const cs * U,
    double * x )
```

Definition at line 3 of file [cs\\_utsolve.c](#).

## 4.138 cs\_utsolve.c

[Go to the documentation of this file.](#)

```
00001 #include "cs.h"
00002 /* solve U'x=b where x and b are dense.  x=b on input, solution on output. */
00003 csi cs_utsolve (const cs *U, double *x)
00004 {
00005     csi p, j, n, *Up, *Ui ;
00006     double *Ux ;
00007     if (!CS_CSC (U) || !x) return (0) ;                /* check inputs */
00008     n = U->n ; Up = U->p ; Ui = U->i ; Ux = U->x ;
00009     for (j = 0 ; j < n ; j++)
00010     {
00011         for (p = Up [j] ; p < Up [j+1]-1 ; p++)
00012         {
00013             x [j] -= Ux [p] * x [Ui [p]] ;
00014         }
00015         x [j] /= Ux [Up [j+1]-1] ;
00016     }
00017     return (1) ;
00018 }
```

## 4.139 include/abip.h File Reference

```
#include "glbopts.h"
#include <string.h>
#include "amatrix.h"
```

## Classes

- struct [solve\\_specific\\_problem](#)
- struct [ABIP\\_CONE](#)
- struct [ABIP\\_PROBLEM\\_DATA](#)
- struct [ABIP\\_SETTINGS](#)
- struct [ABIP\\_SOL\\_VARS](#)
- struct [ABIP\\_INFO](#)
- struct [ABIP\\_WORK](#)
- struct [ABIP\\_RESIDUALS](#)



## Typedefs

- typedef struct [ABIP\\_A\\_DATA\\_MATRIX](#) [ABIPMatrix](#)
- typedef struct [ABIP\\_LIN\\_SYS\\_WORK](#) [ABIPLinSysWork](#)
- typedef struct [ABIP\\_PROBLEM\\_DATA](#) [ABIPData](#)
- typedef struct [ABIP\\_SETTINGS](#) [ABIPSettings](#)
- typedef struct [ABIP\\_SOL\\_VARS](#) [ABIPSolution](#)
- typedef struct [ABIP\\_INFO](#) [ABIPInfo](#)
- typedef struct [ABIP\\_WORK](#) [ABIPWork](#)
- typedef struct [ABIP\\_ADAPTIVE\\_WORK](#) [ABIPAdaptWork](#)
- typedef struct [ABIP\\_RESIDUALS](#) [ABIPResiduals](#)
- typedef struct [ABIP\\_CONE](#) [ABIPCone](#)
- typedef struct [mkl\\_lin\\_sys](#) [MKLlinsys](#)
- typedef struct [solve\\_specific\\_problem](#) [spe\\_problem](#)

## Enumerations

- enum [problem\\_type](#) { [LASSO](#) , [SVM](#) , [QCP](#) , [SVMQP](#) }

## Functions

- [ABIPWork](#) \*[ABIP](#)() [init](#) (const [ABIPData](#) \*d, [ABIPInfo](#) \*info, [spe\\_problem](#) \*s, [ABIPCone](#) \*c)  
*Initialize and allocate memory for the solver.*
- [abip\\_int](#) [ABIP](#)() [solve](#) ([ABIPWork](#) \*w, const [ABIPData](#) \*d, [ABIPSolution](#) \*sol, [ABIPInfo](#) \*info, [ABIPCone](#) \*c, [spe\\_problem](#) \*s)  
*Main solve iteration of the solver.*
- void [ABIP](#)() [finish](#) ([ABIPWork](#) \*w, [spe\\_problem](#) \*spe)  
*Free the memory allocated for the solver.*
- [abip\\_int](#) [ABIP](#)() [main](#) (const [ABIPData](#) \*d, [ABIPSolution](#) \*sol, [ABIPInfo](#) \*info)
- const char \*[ABIP](#)() [version](#) (void)
- [abip\\_int](#) [abip](#) (const [ABIPData](#) \*d, [ABIPSolution](#) \*sol, [ABIPInfo](#) \*info, [ABIPCone](#) \*K)  
*the main function to call the abip conic solver*

### 4.139.1 Typedef Documentation

#### 4.139.1.1 ABIPAdaptWork

```
typedef struct ABIP_ADAPTIVE_WORK ABIPAdaptWork
```

Definition at line 23 of file [abip.h](#).

#### 4.139.1.2 ABIPCone

```
typedef struct ABIP_CONE ABIPCone
```

Definition at line 25 of file [abip.h](#).

#### 4.139.1.3 ABIPData

```
typedef struct ABIP_PROBLEM_DATA ABIPData
```

Definition at line 18 of file [abip.h](#).

#### 4.139.1.4 ABIPInfo

```
typedef struct ABIP_INFO ABIPInfo
```

Definition at line 21 of file [abip.h](#).

#### 4.139.1.5 ABIPLinSysWork

```
typedef struct ABIP_LIN_SYS_WORK ABIPLinSysWork
```

Definition at line 16 of file [abip.h](#).

#### 4.139.1.6 ABIPMatrix

```
typedef struct ABIP_A_DATA_MATRIX ABIPMatrix
```

Definition at line 15 of file [abip.h](#).

#### 4.139.1.7 ABIPResiduals

```
typedef struct ABIP_RESIDUALS ABIPResiduals
```

Definition at line 24 of file [abip.h](#).

#### 4.139.1.8 ABIPSettings

```
typedef struct ABIP_SETTINGS ABIPSettings
```

Definition at line 19 of file [abip.h](#).

#### 4.139.1.9 ABIPSolution

```
typedef struct ABIP_SOL_VARS ABIPSolution
```

Definition at line 20 of file [abip.h](#).

#### 4.139.1.10 ABIPWork

```
typedef struct ABIP_WORK ABIPWork
```

Definition at line 22 of file [abip.h](#).

#### 4.139.1.11 MKLlinsys

```
typedef struct mkl_lin_sys MKLlinsys
```

Definition at line 26 of file [abip.h](#).

#### 4.139.1.12 spe\_problem

```
typedef struct solve_specific_problem spe_problem
```

Definition at line 27 of file [abip.h](#).

### 4.139.2 Enumeration Type Documentation

#### 4.139.2.1 problem\_type

```
enum problem_type
```

### Enumerator

LASSO	
SVM	
QCP	
SVMQP	

Definition at line 13 of file [abip.h](#).

## 4.139.3 Function Documentation

### 4.139.3.1 `abip()`

```
abip_int abip (
    const ABIPData * d,
    ABIPSolution * sol,
    ABIPInfo * info,
    ABIPCone * K )
```

the main function to call the abip conic solver

Definition at line 1335 of file [abip.c](#).

### 4.139.3.2 `finish()`

```
void ABIP() finish (
    ABIPWork * w,
    spe_problem * spe )
```

Free the memory allocated for the solver.

Definition at line 1254 of file [abip.c](#).

### 4.139.3.3 `init()`

```
ABIPWork *ABIP() init (
    const ABIPData * d,
    ABIPInfo * info,
    spe_problem * s,
    ABIPCone * c )
```

Initialize and allocate memory for the solver.

Definition at line 1271 of file [abip.c](#).

**4.139.3.4 main()**

```
abip_int ABIP() main (
    const ABIPData * d,
    ABIPSolution * sol,
    ABIPInfo * info )
```

**4.139.3.5 solve()**

```
abip_int ABIP() solve (
    ABIPWork * w,
    const ABIPData * d,
    ABIPSolution * sol,
    ABIPInfo * info,
    ABIPConc * c,
    spe_problem * s )
```

Main solve iteration of the solver.

Definition at line 1076 of file [abip.c](#).

**4.139.3.6 version()**

```
const char *ABIP() version (
    void )
```

Definition at line 3 of file [abip\\_version.c](#).

**4.140 abip.h**

[Go to the documentation of this file.](#)

```
00001 #ifndef ABIP_H_GUARD
00002 #define ABIP_H_GUARD
00003
00004 #ifdef __cplusplus
00005 extern "C" {
00006 #endif
00007
00008 #include "glbopts.h"
00009 #include <string.h>
00010 #include "amatrix.h"
00011
00012 /*-----special problem to solve-----*/
00013 enum problem_type{LASSO, SVM, QCP, SVMQP};
00014
00015 typedef struct ABIP_A_DATA_MATRIX ABIPMatrix;
00016 typedef struct ABIP_LIN_SYS_WORK ABIPLinSysWork;
00017
00018 typedef struct ABIP_PROBLEM_DATA ABIPData;
00019 typedef struct ABIP_SETTINGS ABIPSettings;
00020 typedef struct ABIP_SOL_VARS ABIPSolution;
00021 typedef struct ABIP_INFO ABIPInfo;
00022 typedef struct ABIP_WORK ABIPWork;
00023 typedef struct ABIP_ADAPTIVE_WORK ABIPAdaptWork;
00024 typedef struct ABIP_RESIDUALS ABIPResiduals;
00025 typedef struct ABIP_CONC ABIPConc;
```

```

00026 typedef struct mkl_lin_sys MKLlinsys;
00027 typedef struct solve_specific_problem spe_problem;
00028
00029 struct solve_specific_problem{
00030
00031     enum problem_type pro_type;
00032     abip_int m; //rows of input data A
00033     abip_int n; //cols of input data A
00034     abip_int p; //rows of ABIP constraint matrix A
00035     abip_int q; //cols of ABIP constraint matrix A
00036     ABIPLinSysWork *L;
00037     ABIPSettings *stgs;
00038     ABIPData *data; //original data
00039     abip_float sparsity;
00040
00041     abip_float *rho_dr; // non-identity DR scaling
00042
00043     /* scaled data */
00044     ABIPMatrix *A; //scaled original constraint matrix
00045     ABIPMatrix *Q; //scaled original quadratic matrix
00046     abip_float *b; //scaled reformulated vector for abip
00047     abip_float *c; //scaled reformulated vector for abip
00048     /*-----*/
00049
00050     void (*scaling_data)(spe_problem *self, ABIPConc *k);
00051     void (*un_scaling_sol)(spe_problem *self, ABIPSolution *sol);
00052     void (*calc_residuals)(spe_problem *self, ABIPWork *w, ABIPResiduals *r, abip_int ipm_iter,
00053         abip_int admm_iter);
00053     abip_int (*init_spe_linsys_work)(spe_problem *self);
00054     abip_int (*solve_spe_linsys)(spe_problem *self, abip_float *b, abip_float *pcg_warm_start,
00055         abip_int iter, abip_float error_ratio);
00055     void (*free_spe_linsys_work)(spe_problem *self);
00056     void (*spe_A_times)(spe_problem *self, const abip_float *x, abip_float *y); //y += Ax, where A is
00057         the reformulated constraint matrix of ABIP
00057     void (*spe_AT_times)(spe_problem *self, const abip_float *x, abip_float *y); //y += A'x, where A
00058         is the reformulated constraint matrix of ABIP
00058     abip_float (*inner_conv_check)(spe_problem *self, ABIPWork *w);
00059
00060 };
00061
00062
00063 /* cols of data matrix A must be specified in this exact order
00064 if change the definition of this order, remember to change the order in
00065 'solve_sub_problem' too
00066 */
00067 struct ABIP_CONC {
00068     abip_int* q; /* array of second-order cone constraints */
00069     abip_int qsize; /* length of SOC array */
00070     abip_int* rq; /* array of rotated second-order cone constraints */
00071     abip_int rqsize; /* length of RSOC array */
00072     abip_int f; /* length of free cone */
00073     abip_int z; /* length of zero cone */
00074     abip_int l; /* length of LP cone */
00075
00076 };
00077
00078
00079 struct ABIP_PROBLEM_DATA
00080 {
00081     abip_int m;
00082     abip_int n;
00083     ABIPMatrix *A;
00084     ABIPMatrix *Q;
00085
00086     abip_float *b;
00087     abip_float *c;
00088
00089     abip_float lambda;
00090     ABIPSettings *stgs;
00091 };
00092
00093 struct ABIP_SETTINGS
00094 {
00095     abip_int normalize;
00096     abip_int scale_E;
00097     abip_int scale_bc;
00098     abip_float scale;
00099     abip_float rho_x;
00100     abip_float rho_y;
00101     abip_float rho_tau;
00102
00103     abip_int max_ipm_iters;
00104     abip_int max_admm_iters;
00105     abip_float eps;
00106     abip_float eps_p;
00107     abip_float eps_d;
00108     abip_float eps_g;

```

```

00109     abip_float eps_inf;
00110     abip_float eps_unb;
00111
00112     abip_float err_dif; //tol between max(dres,pres,dgap) of two consecutive inters

00113     abip_float alpha;
00114     abip_float cg_rate; /* for indirect, tolerance goes down like (1/iter)^cg_rate: 2 */

00115
00116     abip_int use_indirect;
00117     abip_int inner_check_period;
00118     abip_int outer_check_period;
00119
00120     abip_int verbose; /* boolean, write out progress: 1 */
00121     abip_int linsys_solver; // 0:mkldss, 1:qddldl, 2:sparse cholesky, 3:pcg, 4:pardiso, 5:dense
cholesky
00122     abip_int probt_type; // 0:general_qp, 1:lasso, 2:svm, 3:qcp
00123     abip_float time_limit; // in s
00124     abip_float psi;
00125
00126     abip_int origin_scaling;
00127     abip_int ruiz_scaling;
00128     abip_int pc_scaling;
00129
00130 };
00131
00132 struct ABIP_SOL_VARS
00133 {
00134     abip_float *x;
00135     abip_float *y;
00136     abip_float *s;
00137 };
00138
00139 struct ABIP_INFO
00140 {
00141     char status[32];
00142     abip_int status_val;
00143     abip_int ipm_iter;
00144     abip_int admm_iter;
00145
00146     abip_float pobj;
00147     abip_float dobj;
00148     abip_float res_pri;
00149     abip_float res_dual;
00150     abip_float rel_gap;
00151     abip_float res_infeas;
00152     abip_float res_unbdd;
00153
00154     abip_float setup_time;
00155     abip_float solve_time;
00156     abip_float avg_linsys_time;
00157     abip_float avg_cg_iters;
00158 };
00159
00160
00161 struct ABIP_WORK
00162 {
00163     abip_float sigma;
00164     abip_float gamma;
00165     abip_float mu;
00166     abip_float beta;
00167     abip_float *u;
00168     abip_float *v;
00169     abip_float *v_origin;
00170     abip_float *u_t;
00171     abip_float *rel_ut;
00172     abip_float nm_inf_b;
00173     abip_float nm_inf_c;
00174     abip_int m;
00175     abip_int n;
00176     ABIPMatrix *A;
00177     abip_float *r;
00178     abip_float a;
00179
00180 };
00181
00182 struct ABIP_RESIDUALS
00183 {
00184     abip_int last_ipm_iter;
00185     abip_int last_admm_iter;
00186     abip_float last_mu;
00187
00188     abip_float res_pri;
00189     abip_float res_dual;
00190     abip_float rel_gap;
00191     abip_float res_infeas;
00192     abip_float res_unbdd;

```

```

00193
00194     abip_float ct_x_by_tau;
00195     abip_float bt_y_by_tau;
00196
00197     abip_float pobj;
00198     abip_float dobj;
00199
00200     abip_float tau;
00201     abip_float kap;
00202
00203     abip_float res_dif;
00204     abip_float error_ratio;
00205
00206     abip_float Ax_b_norm;
00207     abip_float Qx_ATy_c_s_norm;
00208 };
00209
00210
00211 ABIPWork *ABIP(init)
00212 (
00213     const ABIPData *d,
00214     ABIPInfo *info,
00215     spe_problem *s,
00216     ABIPCone *c
00217 );
00218 abip_int ABIP(solve)
00219 (
00220     ABIPWork *w,
00221     const ABIPData *d,
00222     ABIPSolution *sol,
00223     ABIPInfo *info,
00224     ABIPCone *c,
00225     spe_problem *s
00226 );
00227 void ABIP(finish)
00228 (
00229     ABIPWork *w,
00230     spe_problem *spe
00231 );
00232
00233 abip_int ABIP(main) (const ABIPData *d, ABIPSolution *sol, ABIPInfo *info);
00234 const char *ABIP(version) (void);
00235 abip_int abip
00236 (
00237     const ABIPData* d,
00238     ABIPSolution* sol,
00239     ABIPInfo* info,
00240     ABIPCone *K
00241 );
00242
00243
00244 #ifdef __cplusplus
00245 }
00246 #endif
00247 #endif

```

## 4.141 include/amatrix.h File Reference

```
#include "glbopts.h"
```

### Classes

- struct [ABIP\\_A\\_DATA\\_MATRIX](#)

## 4.142 amatrix.h

[Go to the documentation of this file.](#)

```

00001 #ifndef AMATRIX_H_GUARD
00002 #define AMATRIX_H_GUARD
00003

```



```

00004 #ifdef __cplusplus
00005 extern "C" {
00006 #endif
00007
00008 #include "glbopts.h"
00009
00010 /* A is supplied in column compressed format */
00011 struct ABIP_A_DATA_MATRIX
00012 {
00013     abip_float *x;      /* A values, size: NNZ A */
00014     abip_int *i;        /* A row index, size: NNZ A */
00015     abip_int *p;        /* A column pointer, size: n+1 */
00016     abip_int m;         /* m rows, n cols */
00017     abip_int n;
00018 };
00019
00020 #ifdef __cplusplus
00021 }
00022 #endif
00023 #endif

```

## 4.143 include/cones.h File Reference

```

#include "glbopts.h"
#include "abip.h"
#include "linalg.h"
#include "mkl.h"
#include "mkl_lapacke.h"

```

### Functions

- char \*ABIP() [get\\_cone\\_header](#) (const [ABIPCone](#) \*k)  
*Get the number of variables and blocks of each cone.*
- [abip\\_int](#) ABIP() [validate\\_cones](#) ([spe\\_problem](#) \*spe, const [ABIPCone](#) \*k)  
*Check if the cone dimensions are valid.*
- [abip\\_int](#) ABIP() [get\\_cone\\_dims](#) (const [ABIPCone](#) \*k)  
*Calculate the total number of dimensions of all the cones.*
- void ABIP() [soc\\_barrier\\_subproblem](#) ([abip\\_float](#) \*x, [abip\\_float](#) \*tmp, [abip\\_float](#) lambda, [abip\\_int](#) n)  
*Barrier subproblem for the second order cone.*
- void ABIP() [rsoc\\_barrier\\_subproblem](#) ([abip\\_float](#) \*x, [abip\\_float](#) \*tmp, [abip\\_float](#) lambda, [abip\\_int](#) n)  
*Barrier subproblem for the rotated second order cone.*
- void ABIP() [free\\_barrier\\_subproblem](#) ([abip\\_float](#) \*x, [abip\\_float](#) \*tmp, [abip\\_float](#) lambda, [abip\\_int](#) n)  
*Barrier subproblem for the free cone.*
- void ABIP() [zero\\_barrier\\_subproblem](#) ([abip\\_float](#) \*x, [abip\\_float](#) \*tmp, [abip\\_float](#) lambda, [abip\\_int](#) n)  
*Barrier subproblem for the zero cone.*
- void ABIP() [positive\\_orthant\\_barrier\\_subproblem](#) ([abip\\_float](#) \*x, [abip\\_float](#) \*tmp, [abip\\_float](#) lambda, [abip\\_int](#) n)  
*Barrier subproblem for the positive orthant cone.*

### 4.143.1 Function Documentation

#### 4.143.1.1 free\_barrier\_subproblem()

```
void ABIP() free_barrier_subproblem (
    abip_float * x,
    abip_float * tmp,
    abip_float lambda,
    abip_int n )
```

Barrier subproblem for the free cone.

Definition at line 255 of file [cones.c](#).

#### 4.143.1.2 get\_cone\_dims()

```
abip_int ABIP() get_cone_dims (
    const ABIPConc * k )
```

Calculate the total number of dimensions of all the cones.

Definition at line 8 of file [cones.c](#).

#### 4.143.1.3 get\_cone\_header()

```
char *ABIP() get_cone_header (
    const ABIPConc * k )
```

Get the number of variables and blocks of each cone.

Definition at line 87 of file [cones.c](#).

#### 4.143.1.4 positive\_orthant\_barrier\_subproblem()

```
void ABIP() positive_orthant_barrier_subproblem (
    abip_float * x,
    abip_float * tmp,
    abip_float lambda,
    abip_int n )
```

Barrier subproblem for the positive orthant cone.

Definition at line 279 of file [cones.c](#).

#### 4.143.1.5 rsoc\_barrier\_subproblem()

```
void ABIP() rsoc_barrier_subproblem (
    abip_float * x,
    abip_float * tmp,
    abip_float lambda,
    abip_int n )
```

Barrier subproblem for the rotated second order cone.

Definition at line 169 of file [cones.c](#).

#### 4.143.1.6 soc\_barrier\_subproblem()

```
void ABIP() soc_barrier_subproblem (
    abip_float * x,
    abip_float * tmp,
    abip_float lambda,
    abip_int n )
```

Barrier subproblem for the second order cone.

Definition at line 130 of file [cones.c](#).

#### 4.143.1.7 validate\_cones()

```
abip_int ABIP() validate_cones (
    spe_problem * spe,
    const ABIPConc * k )
```

Check if the cone dimensions are valid.

Definition at line 37 of file [cones.c](#).

#### 4.143.1.8 zero\_barrier\_subproblem()

```
void ABIP() zero_barrier_subproblem (
    abip_float * x,
    abip_float * tmp,
    abip_float lambda,
    abip_int n )
```

Barrier subproblem for the zero cone.

Definition at line 267 of file [cones.c](#).

## 4.144 cones.h

[Go to the documentation of this file.](#)

```

00001 #ifndef CONES_H_GUARD
00002 #define CONES_H_GUARD
00003
00004 #ifdef __cplusplus
00005 extern "C" {
00006 #endif
00007
00008 #include "glbopts.h"
00009 #include "abip.h"
00010 #include "linalg.h"
00011 #include "mkl.h"
00012 #include "mkl_lapacke.h"
00013
00014
00015
00016 char* ABIP(get_cone_header)(const ABIPCone* k);
00017 abip_int ABIP(validate_cones)(spe_problem *spe, const ABIPCone* k);
00018
00019 abip_int ABIP(get_cone_dims)(const ABIPCone* k);
00020
00021
00022 void ABIP(soc_barrier_subproblem)(
00023     abip_float* x,
00024     abip_float* tmp,
00025     abip_float lambda,
00026     abip_int n
00027 );
00028
00029 void ABIP(rsoc_barrier_subproblem)(
00030     abip_float* x,
00031     abip_float* tmp,
00032     abip_float lambda,
00033     abip_int n
00034 );
00035
00036 void ABIP(free_barrier_subproblem)(
00037     abip_float* x,
00038     abip_float* tmp,
00039     abip_float lambda,
00040     abip_int n
00041 );
00042
00043 void ABIP(zero_barrier_subproblem)(
00044     abip_float* x,
00045     abip_float* tmp,
00046     abip_float lambda,
00047     abip_int n
00048 );
00049
00050 void ABIP(positive_orthant_barrier_subproblem)(
00051     abip_float* x,
00052     abip_float* tmp,
00053     abip_float lambda,
00054     abip_int n
00055 );
00056
00057
00058 #ifdef __cplusplus
00059 }
00060 #endif
00061 #endif
00062

```

## 4.145 include/ctrlc.h File Reference

### Macros

- #define [abip\\_start\\_interrupt\\_listener\(\)](#)
- #define [abip\\_end\\_interrupt\\_listener\(\)](#)
- #define [abip\\_is\\_interrupted\(\)](#) 0

## Typedefs

- typedef int [abip\\_make\\_iso\\_compilers\\_happy](#)

### 4.145.1 Macro Definition Documentation

#### 4.145.1.1 [abip\\_end\\_interrupt\\_listener](#)

```
#define abip_end_interrupt_listener( )
```

Definition at line 36 of file [ctrlc.h](#).

#### 4.145.1.2 [abip\\_is\\_interrupted](#)

```
#define abip_is_interrupted( ) 0
```

Definition at line 37 of file [ctrlc.h](#).

#### 4.145.1.3 [abip\\_start\\_interrupt\\_listener](#)

```
#define abip_start_interrupt_listener( )
```

Definition at line 35 of file [ctrlc.h](#).

### 4.145.2 Typedef Documentation

#### 4.145.2.1 [abip\\_make\\_iso\\_compilers\\_happy](#)

```
typedef int abip\_make\_iso\_compilers\_happy
```

Definition at line 33 of file [ctrlc.h](#).

## 4.146 ctrlc.h

[Go to the documentation of this file.](#)

```
00001 /* Interface for ABIP signal handling. */
00002
00003 #ifndef CTRLC_H_GUARD
00004 #define CTRLC_H_GUARD
00005
00006 #ifdef __cplusplus
00007 extern "C" {
00008 #endif
00009
00010 #if CTRLC > 0
00011
00012 #if defined MATLAB_MEX_FILE
00013
00014 extern int utIsInterruptPending();
00015 extern int utSetInterruptEnabled(int);
00016
00017 #elif (defined _WIN32 || defined _WIN64 || defined _WINDLL)
00018
00019 #include <windows.h>
00020
00021 #else
00022
00023 #include <signal.h>
00024
00025 #endif
00026
00027 void abip_start_interrupt_listener(void);
00028 void abip_end_interrupt_listener(void);
00029 int abip_is_interrupted(void);
00030
00031 #else
00032
00033 typedef int abip_make_iso_compilers_happy;
00034
00035 #define abip_start_interrupt_listener()
00036 #define abip_end_interrupt_listener()
00037 #define abip_is_interrupted() 0
00038
00039 #endif
00040
00041 #ifdef __cplusplus
00042 }
00043 #endif
00044 #endif
```

## 4.147 include/glbopts.h File Reference

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

### Macros

- #define [ABIP\(x\)](#) abip\_##x
- #define [ABIP\\_VERSION](#) ("2.0.0") /\* string literals automatically null-terminated \*/
- #define [ABIP\\_INFEASIBLE\\_INACCURATE](#) (-7)
- #define [ABIP\\_UNBOUNDED\\_INACCURATE](#) (-6)
- #define [ABIP\\_SIGINT](#) (-5)
- #define [ABIP\\_FAILED](#) (-4)
- #define [ABIP\\_INDETERMINATE](#) (-3)
- #define [ABIP\\_INFEASIBLE](#) (-2)
- #define [ABIP\\_UNBOUNDED](#) (-1)
- #define [ABIP\\_UNFINISHED](#) (0)

- #define [ABIP\\_SOLVED](#) (1)
- #define [ABIP\\_SOLVED\\_INACCURATE](#) (2)
- #define [ABIP\\_UNSOLVED](#) (3)
- #define [SIGMA](#) (0.8)
- #define [GAMMA](#) (1.6)
- #define [MAX\\_IPM\\_ITERS](#) (500)
- #define [MAX\\_ADMM\\_ITERS](#) (10000000)
- #define [EPS](#) (1E-3)
- #define [ALPHA](#) (1.8)
- #define [CG\\_RATE](#) (2.0)
- #define [CG\\_BEST\\_TOL](#) (1e-9)
- #define [CG\\_MIN\\_TOL](#) (1e-5)
- #define [NORMALIZE](#) (1)
- #define [SCALE](#) (1.0)
- #define [SPARSITY\\_RATIO](#) (0.01)
- #define [RHO\\_Y](#) (1E-3)
- #define [ADAPTIVE](#) (1)
- #define [EPS\\_COR](#) (0.2)
- #define [EPS\\_PEN](#) (0.1)
- #define [ADAPTIVE\\_LOOKBACK](#) (20)
- #define [VERBOSE](#) (1)
- #define [WARM\\_START](#) (0)
- #define [CONE\\_TOL](#) (1e-8)
- #define [abip\\_printf](#) printf
- #define [\\_abip\\_free](#) free
- #define [\\_abip\\_malloc](#) malloc
- #define [\\_abip\\_calloc](#) calloc
- #define [\\_abip\\_realloc](#) realloc
- #define [abip\\_free](#)(x)
- #define [abip\\_malloc](#)(x) [\\_abip\\_malloc](#)(x)
- #define [abip\\_calloc](#)(x, y) [\\_abip\\_calloc](#)(x, y)
- #define [abip\\_realloc](#)(x, y) [\\_abip\\_realloc](#)(x, y)
- #define [NAN](#) (([abip\\_float](#))0x7ff8000000000000)
- #define [INFINITY](#) NAN
- #define [ABIP\\_NULL](#) 0
- #define [MAX](#)(a, b) (((a) > (b)) ? (a) : (b))
- #define [MIN](#)(a, b) (((a) < (b)) ? (a) : (b))
- #define [ABS](#)(x) (((x) < 0) ? -(x) : (x))
- #define [POWF](#) pow
- #define [SQRTF](#) sqrt
- #define [DEBUG\\_FUNC](#)
- #define [RETURN](#) return
- #define [EPS\\_TOL](#) (1E-18)
- #define [SAFEDIV\\_POS](#)(X, Y) ((Y) < [EPS\\_TOL](#) ? ((X) / [EPS\\_TOL](#)) : (X) / (Y))
- #define [CONVERGED\\_INTERVAL](#) (1)
- #define [INDETERMINATE\\_TOL](#) (1e-9)

## Typedefs

- typedef int [abip\\_int](#)
- typedef double [abip\\_float](#)

## 4.147.1 Macro Definition Documentation

### 4.147.1.1 `_abip_calloc`

```
#define _abip_calloc calloc
```

Definition at line 80 of file [glbopts.h](#).

### 4.147.1.2 `_abip_free`

```
#define _abip_free free
```

Definition at line 78 of file [glbopts.h](#).

### 4.147.1.3 `_abip_malloc`

```
#define _abip_malloc malloc
```

Definition at line 79 of file [glbopts.h](#).

### 4.147.1.4 `_abip_realloc`

```
#define _abip_realloc realloc
```

Definition at line 81 of file [glbopts.h](#).

### 4.147.1.5 `ABIP`

```
#define ABIP(  
    x ) abip_##x
```

Definition at line 13 of file [glbopts.h](#).



#### 4.147.1.6 abip\_malloc

```
#define abip_malloc(  
    x,  
    y ) _abip_malloc(x, y)
```

Definition at line 88 of file [glbopts.h](#).

#### 4.147.1.7 ABIP\_FAILED

```
#define ABIP_FAILED (-4)
```

Definition at line 23 of file [glbopts.h](#).

#### 4.147.1.8 abip\_free

```
#define abip_free(  
    x )
```

**Value:**

```
_abip_free(x);  
x = ABIP_NULL
```

Definition at line 84 of file [glbopts.h](#).

#### 4.147.1.9 ABIP\_INDETERMINATE

```
#define ABIP_INDETERMINATE (-3)
```

Definition at line 24 of file [glbopts.h](#).

#### 4.147.1.10 ABIP\_INFEASIBLE

```
#define ABIP_INFEASIBLE (-2)
```

Definition at line 25 of file [glbopts.h](#).

#### 4.147.1.11 ABIP\_INFEASIBLE\_INACCURATE

```
#define ABIP_INFEASIBLE_INACCURATE (-7)
```

Definition at line 20 of file [glbopts.h](#).

#### 4.147.1.12 abip\_malloc

```
#define abip_malloc(  
    x ) _abip_malloc(x)
```

Definition at line 87 of file [glbopts.h](#).

#### 4.147.1.13 ABIP\_NULL

```
#define ABIP_NULL 0
```

Definition at line 135 of file [glbopts.h](#).

#### 4.147.1.14 abip\_printf

```
#define abip_printf printf
```

Definition at line 77 of file [glbopts.h](#).

#### 4.147.1.15 abip\_realloc

```
#define abip_realloc(  
    x,  
    y ) _abip_realloc(x, y)
```

Definition at line 89 of file [glbopts.h](#).

#### 4.147.1.16 ABIP\_SIGINT

```
#define ABIP_SIGINT (-5)
```

Definition at line 22 of file [glbopts.h](#).

**4.147.1.17 ABIP\_SOLVED**

```
#define ABIP_SOLVED (1)
```

Definition at line 28 of file [glbopts.h](#).

**4.147.1.18 ABIP\_SOLVED\_INACCURATE**

```
#define ABIP_SOLVED_INACCURATE (2)
```

Definition at line 29 of file [glbopts.h](#).

**4.147.1.19 ABIP\_UNBOUNDED**

```
#define ABIP_UNBOUNDED (-1)
```

Definition at line 26 of file [glbopts.h](#).

**4.147.1.20 ABIP\_UNBOUNDED\_INACCURATE**

```
#define ABIP_UNBOUNDED_INACCURATE (-6)
```

Definition at line 21 of file [glbopts.h](#).

**4.147.1.21 ABIP\_UNFINISHED**

```
#define ABIP_UNFINISHED (0)
```

Definition at line 27 of file [glbopts.h](#).

**4.147.1.22 ABIP\_UNSOLVED**

```
#define ABIP_UNSOLVED (3)
```

Definition at line 30 of file [glbopts.h](#).

#### 4.147.1.23 ABIP\_VERSION

```
#define ABIP_VERSION ("2.0.0") /* string literals automatically null-terminated */
```

Definition at line 17 of file [glbopts.h](#).

#### 4.147.1.24 ABS

```
#define ABS(  
    x ) ((x) < 0) ? -(x) : (x)
```

Definition at line 146 of file [glbopts.h](#).

#### 4.147.1.25 ADAPTIVE

```
#define ADAPTIVE (1)
```

Definition at line 45 of file [glbopts.h](#).

#### 4.147.1.26 ADAPTIVE\_LOOKBACK

```
#define ADAPTIVE_LOOKBACK (20)
```

Definition at line 48 of file [glbopts.h](#).

#### 4.147.1.27 ALPHA

```
#define ALPHA (1.8)
```

Definition at line 37 of file [glbopts.h](#).

#### 4.147.1.28 CG\_BEST\_TOL

```
#define CG_BEST_TOL (1e-9)
```

Definition at line 39 of file [glbopts.h](#).

#### 4.147.1.29 CG\_MIN\_TOL

```
#define CG_MIN_TOL (1e-5)
```

Definition at line 40 of file [glbopts.h](#).

#### 4.147.1.30 CG\_RATE

```
#define CG_RATE (2.0)
```

Definition at line 38 of file [glbopts.h](#).

#### 4.147.1.31 CONE\_TOL

```
#define CONE_TOL (1e-8)
```

Definition at line 52 of file [glbopts.h](#).

#### 4.147.1.32 CONVERGED\_INTERVAL

```
#define CONVERGED_INTERVAL (1)
```

Definition at line 181 of file [glbopts.h](#).

#### 4.147.1.33 DEBUG\_FUNC

```
#define DEBUG_FUNC
```

Definition at line 174 of file [glbopts.h](#).

#### 4.147.1.34 EPS

```
#define EPS (1E-3)
```

Definition at line 36 of file [glbopts.h](#).

#### 4.147.1.35 EPS\_COR

```
#define EPS_COR (0.2)
```

Definition at line 46 of file [glbopts.h](#).

#### 4.147.1.36 EPS\_PEN

```
#define EPS_PEN (0.1)
```

Definition at line 47 of file [glbopts.h](#).

#### 4.147.1.37 EPS\_TOL

```
#define EPS_TOL (1E-18)
```

Definition at line 178 of file [glbopts.h](#).

#### 4.147.1.38 GAMMA

```
#define GAMMA (1.6)
```

Definition at line 32 of file [glbopts.h](#).

#### 4.147.1.39 INDETERMINATE\_TOL

```
#define INDETERMINATE_TOL (1e-9)
```

Definition at line 182 of file [glbopts.h](#).

#### 4.147.1.40 INFINITY

```
#define INFINITY NAN
```

Definition at line 123 of file [glbopts.h](#).

#### 4.147.1.41 MAX

```
#define MAX(  
    a,  
    b ) ((a) > (b)) ? (a) : (b)
```

Definition at line 138 of file [glbopts.h](#).

#### 4.147.1.42 MAX\_ADMM\_ITERS

```
#define MAX_ADMM_ITERS (10000000)
```

Definition at line 35 of file [glbopts.h](#).

#### 4.147.1.43 MAX\_IPM\_ITERS

```
#define MAX_IPM_ITERS (500)
```

Definition at line 34 of file [glbopts.h](#).

#### 4.147.1.44 MIN

```
#define MIN(  
    a,  
    b ) ((a) < (b)) ? (a) : (b)
```

Definition at line 142 of file [glbopts.h](#).

#### 4.147.1.45 NAN

```
#define NAN ((abip_float)0x7ff8000000000000)
```

Definition at line 120 of file [glbopts.h](#).

#### 4.147.1.46 NORMALIZE

```
#define NORMALIZE (1)
```

Definition at line 41 of file [glbopts.h](#).

#### 4.147.1.47 POWF

```
#define POWF pow
```

Definition at line 153 of file [glbopts.h](#).

#### 4.147.1.48 RETURN

```
#define RETURN return
```

Definition at line 175 of file [glbopts.h](#).

#### 4.147.1.49 RHO\_Y

```
#define RHO_Y (1E-3)
```

Definition at line 44 of file [glbopts.h](#).

#### 4.147.1.50 SAFEDIV\_POS

```
#define SAFEDIV_POS(  
    X,  
    Y ) ((Y) < EPS_TOL ? ((X) / EPS_TOL) : (X) / (Y))
```

Definition at line 179 of file [glbopts.h](#).

#### 4.147.1.51 SCALE

```
#define SCALE (1.0)
```

Definition at line 42 of file [glbopts.h](#).

#### 4.147.1.52 SIGMA

```
#define SIGMA (0.8)
```

Definition at line 31 of file [glbopts.h](#).



#### 4.147.1.53 SPARSITY\_RATIO

```
#define SPARSITY_RATIO (0.01)
```

Definition at line 43 of file [glbopts.h](#).

#### 4.147.1.54 SQRTF

```
#define SQRTF sqrt
```

Definition at line 161 of file [glbopts.h](#).

#### 4.147.1.55 VERBOSE

```
#define VERBOSE (1)
```

Definition at line 49 of file [glbopts.h](#).

#### 4.147.1.56 WARM\_START

```
#define WARM_START (0)
```

Definition at line 50 of file [glbopts.h](#).

### 4.147.2 Typedef Documentation

#### 4.147.2.1 abip\_float

```
typedef double abip_float
```

Definition at line 118 of file [glbopts.h](#).

#### 4.147.2.2 abip\_int

```
typedef int abip_int
```

Definition at line 113 of file [glbopts.h](#).

## 4.148 glbopts.h

[Go to the documentation of this file.](#)

```

00001 #ifndef GLB_H_GUARD
00002 #define GLB_H_GUARD
00003
00004 #ifdef __cplusplus
00005 extern "C" {
00006 #endif
00007
00008 #include <math.h>
00009
00010 // #define DLONG
00011
00012 #ifndef ABIP
00013 #define ABIP(x) abip_##x
00014 #endif
00015
00016 /* ABIP VERSION NUMBER ----- */
00017 #define ABIP_VERSION
00018 ("2.0.0") /* string literals automatically null-terminated */
00019
00020 #define ABIP_INFEASIBLE_INACCURATE (-7)
00021 #define ABIP_UNBOUNDED_INACCURATE (-6)
00022 #define ABIP_SIGINT (-5)
00023 #define ABIP_FAILED (-4)
00024 #define ABIP_INDETERMINATE (-3)
00025 #define ABIP_INFEASIBLE (-2)
00026 #define ABIP_UNBOUNDED (-1)
00027 #define ABIP_UNFINISHED (0)
00028 #define ABIP_SOLVED (1)
00029 #define ABIP_SOLVED_INACCURATE (2)
00030 #define ABIP_UNSOLVED (3)
00031 #define SIGMA (0.8)
00032 #define GAMMA (1.6)
00033
00034 #define MAX_IPM_ITERS (500)
00035 #define MAX_ADMM_ITERS (10000000)
00036 #define EPS (1E-3)
00037 #define ALPHA (1.8)
00038 #define CG_RATE (2.0)
00039 #define CG_BEST_TOL (1e-9)
00040 #define CG_MIN_TOL (1e-5)
00041 #define NORMALIZE (1)
00042 #define SCALE (1.0)
00043 #define SPARSITY_RATIO (0.01)
00044 #define RHO_Y (1E-3)
00045 #define ADAPTIVE (1)
00046 #define EPS_COR (0.2)
00047 #define EPS_PEN (0.1)
00048 #define ADAPTIVE_LOOKBACK (20)
00049 #define VERBOSE (1)
00050 #define WARM_START (0)
00051
00052 #define CONE_TOL (1e-8)
00053
00054 #ifdef MATLAB_MEX_FILE
00055 #include "mex.h"
00056 #define abip_printf mexPrintf
00057 #define _abip_free mxFree
00058 #define _abip_malloc mxMalloc
00059 #define _abip_calloc mxCalloc
00060 #define _abip_realloc mxRealloc
00061 #elif defined PYTHON
00062 #include <Python.h>
00063 #include <stdlib.h>
00064 #define abip_printf(...)
00065 {
00066     PyGILState_STATE gilstate = PyGILState_Ensure();
00067     PySys_WriteStdout(__VA_ARGS__);
00068     PyGILState_Release(gilstate);
00069 }
00070 #define _abip_free free
00071 #define _abip_malloc malloc
00072 #define _abip_calloc calloc
00073 #define _abip_realloc realloc
00074 #else
00075 #include <stdio.h>
00076 #include <stdlib.h>
00077 #define abip_printf printf
00078 #define _abip_free free
00079 #define _abip_malloc malloc
00080 #define _abip_calloc calloc
00081 #define _abip_realloc realloc
00082 #endif

```

```

00083
00084 #define abip_free(x) \
00085     _abip_free(x); \
00086     x = ABIP_NULL
00087 #define abip_malloc(x) _abip_malloc(x)
00088 #define abip_calloc(x, y) _abip_calloc(x, y)
00089 #define abip_realloc(x, y) _abip_realloc(x, y)
00090
00091 // //ifdef DLONG
00092 // //ifdef _WIN64
00093 // //typedef __int64 abip_int;
00094 // //else
00095 // //typedef long abip_int;
00096 // //endif
00097 // //else
00098 // //typedef int abip_int;
00099 // //endif
00100 // typedef int abip_int;
00101
00102
00103 #ifdef DLONG
00104 /*#ifdef _WIN64
00105 #include <stdint.h>
00106 typedef int64_t abip_int;
00107 #else
00108 typedef long abip_int;
00109 #endif
00110 */
00111 typedef long long abip_int;
00112 #else
00113 typedef int abip_int;
00114 #endif
00115
00116
00117 #ifndef SFLOAT
00118 typedef double abip_float;
00119 #ifndef NAN
00120 #define NAN ((abip_float)0x7ff8000000000000)
00121 #endif
00122 #ifndef INFINITY
00123 #define INFINITY NAN
00124 #endif
00125 #else
00126 typedef float abip_float;
00127 #ifndef NAN
00128 #define NAN ((float)0x7fc00000)
00129 #endif
00130 #ifndef INFINITY
00131 #define INFINITY NAN
00132 #endif
00133 #endif
00134
00135 #define ABIP_NULL 0
00136
00137 #ifndef MAX
00138 #define MAX(a, b) (((a) > (b)) ? (a) : (b))
00139 #endif
00140
00141 #ifndef MIN
00142 #define MIN(a, b) (((a) < (b)) ? (a) : (b))
00143 #endif
00144
00145 #ifndef ABS
00146 #define ABS(x) (((x) < 0) ? -(x) : (x))
00147 #endif
00148
00149 #ifndef POWF
00150 #ifndef SFLOAT
00151 #define POWF powf
00152 #else
00153 #define POWF pow
00154 #endif
00155 #endif
00156
00157 #ifndef SQRTF
00158 #ifndef SFLOAT
00159 #define SQRTF sqrtf
00160 #else
00161 #define SQRTF sqrt
00162 #endif
00163 #endif
00164
00165 #if EXTRA_VERBOSE > 1
00166 #if (defined _WIN32 || defined _WIN64 || defined _WINDLL)
00167 #define __func__ __FUNCTION__
00168 #endif
00169 #define DEBUG_FUNC abip_printf("IN function: %s, time: %4f ms, file: %s, line: %i\n", __func__,

```

```

    ABIP(tocq) (&global_timer), __FILE__, __LINE__);
00170 #define RETURN
00171     abip_printf("EXIT function: %s, time: %4f ms, file: %s, line: %i\n", __func__,
    ABIP(tocq) (&global_timer), __FILE__, __LINE__); \
00172     return
00173 #else
00174 #define DEBUG_FUNC
00175 #define RETURN return
00176 #endif
00177
00178 #define EPS_TOL (1E-18)
00179 #define SAFEDIV_POS(X, Y) ((Y) < EPS_TOL ? ((X) / EPS_TOL) : (X) / (Y))
00180
00181 #define CONVERGED_INTERVAL (1)
00182 #define INDETERMINATE_TOL (1e-9)
00183
00184 #ifdef __cplusplus
00185 }
00186 #endif
00187 #endif

```

## 4.149 include/lasso\_config.h File Reference

```

#include "glbopts.h"
#include "abip.h"
#include "amatrix.h"
#include "linsys.h"

```

### Classes

- struct [Lasso](#)

### Typedefs

- typedef struct [Lasso](#) [lasso](#)

### Functions

- [abip\\_int init\\_lasso](#) ([lasso](#) \*\*gen\_lasso, [ABIPData](#) \*d, [ABIPSettings](#) \*stgs)  
*Initialize the lasso problem structure.*
- void [scaling\\_lasso\\_data](#) ([lasso](#) \*self, [ABIPCone](#) \*k)  
*Customized scaling procedure for the lasso problem.*
- void [un\\_scaling\\_lasso\\_sol](#) ([lasso](#) \*self, [ABIPSolution](#) \*sol)  
*Get the unscaled solution of the original lasso problem.*
- void [calc\\_lasso\\_residuals](#) ([lasso](#) \*self, [ABIPWork](#) \*w, [ABIPResiduals](#) \*r, [abip\\_int](#) ipm\_iter, [abip\\_int](#) admm\_iter)  
*Calculate the residuals of the lasso qcp problem.*
- [abip\\_int init\\_lasso\\_linsys\\_work](#) ([lasso](#) \*self)  
*Initialize the linear system solver work space for the lasso problem.*
- [abip\\_int solve\\_lasso\\_linsys](#) ([lasso](#) \*self, [abip\\_float](#) \*b, [abip\\_float](#) \*pcg\_warm\_start, [abip\\_int](#) iter, [abip\\_float](#) error\_ratio)  
*Customized linear system solver for the lasso problem.*
- void [free\\_lasso\\_linsys\\_work](#) ([lasso](#) \*self)  
*Free the linear system solver work space for the lasso problem.*
- void [lasso\\_A\\_times](#) ([lasso](#) \*self, const [abip\\_float](#) \*x, [abip\\_float](#) \*y)  
*Customized matrix-vector multiplication for the lasso problem with A untransposed.*
- void [lasso\\_AT\\_times](#) ([lasso](#) \*self, const [abip\\_float](#) \*x, [abip\\_float](#) \*y)  
*Customized matrix-vector multiplication for the lasso problem with A transposed.*
- [abip\\_float lasso\\_inner\\_conv\\_check](#) ([lasso](#) \*self, [ABIPWork](#) \*w)  
*Check whether the inner loop of the lasso problem has converged.*

## 4.149.1 Typedef Documentation

### 4.149.1.1 lasso

```
typedef struct Lasso lasso
```

Definition at line 12 of file [lasso\\_config.h](#).

## 4.149.2 Function Documentation

### 4.149.2.1 calc\_lasso\_residuals()

```
void calc_lasso_residuals (
    lasso * self,
    ABIPWork * w,
    ABIPResiduals * r,
    abip_int ipm_iter,
    abip_int admm_iter )
```

Calculate the residuals of the lasso qcp problem.

Definition at line 367 of file [lasso\\_config.c](#).

### 4.149.2.2 free\_lasso\_linsys\_work()

```
void free_lasso_linsys_work (
    lasso * self )
```

Free the linear system solver work space for the lasso problem.

Definition at line 722 of file [lasso\\_config.c](#).

### 4.149.2.3 init\_lasso()

```
abip_int init_lasso (
    lasso ** gen_lasso,
    ABIPData * d,
    ABIPSettings * stgs )
```

Initialize the lasso problem structure.

Definition at line 8 of file [lasso\\_config.c](#).

#### 4.149.2.4 `init_lasso_linsys_work()`

```
abip_int init_lasso_linsys_work (
    lasso * self )
```

Initialize the linear system solver work space for the lasso problem.

Definition at line 624 of file [lasso\\_config.c](#).

#### 4.149.2.5 `lasso_A_times()`

```
void lasso_A_times (
    lasso * self,
    const abip_float * x,
    abip_float * y )
```

Customized matrix-vector multiplication for the lasso problem with A untransposed.

Definition at line 99 of file [lasso\\_config.c](#).

#### 4.149.2.6 `lasso_AT_times()`

```
void lasso_AT_times (
    lasso * self,
    const abip_float * x,
    abip_float * y )
```

Customized matrix-vector multiplication for the lasso problem with A transposed.

Definition at line 116 of file [lasso\\_config.c](#).

#### 4.149.2.7 `lasso_inner_conv_check()`

```
abip_float lasso_inner_conv_check (
    lasso * self,
    ABIPWork * w )
```

Check whether the inner loop of the lasso problem has converged.

Definition at line 323 of file [lasso\\_config.c](#).

**4.149.2.8 scaling\_lasso\_data()**

```
void scaling_lasso_data (
    lasso * self,
    ABIPCone * k )
```

Customized scaling procedure for the lasso problem.

Definition at line 131 of file [lasso\\_config.c](#).

**4.149.2.9 solve\_lasso\_linsys()**

```
abip_int solve_lasso_linsys (
    lasso * self,
    abip_float * b,
    abip_float * pcg_warm_start,
    abip_int iter,
    abip_float error_ratio )
```

Customized linear system solver for the lasso problem.

Definition at line 648 of file [lasso\\_config.c](#).

**4.149.2.10 un\_scaling\_lasso\_sol()**

```
void un_scaling_lasso_sol (
    lasso * self,
    ABIPSolution * sol )
```

Get the unscaled solution of the original lasso problem.

Definition at line 303 of file [lasso\\_config.c](#).

**4.150 lasso\_config.h**

[Go to the documentation of this file.](#)

```
00001 #ifndef LASSO_CONFIG_H_GUARD
00002 #define LASSO_CONFIG_H_GUARD
00003
00004 #ifdef __cplusplus
00005 extern "C" {
00006 #endif
00007
00008 #include "glbopts.h"
00009 #include "abip.h"
00010 #include "amatrix.h"
00011 #include "linsys.h"
00012 typedef struct Lasso lasso;
00013
00014 struct Lasso{
00015
00016     /*-----common parts-----*/
00017     enum problem_type pro_type;
00018     abip_int m; //rows of input data A
```

```

00019     abip_int n; //cols of input data A
00020     abip_int p; //rows of ABIP constraint matrix A
00021     abip_int q; //cols of ABIP constraint matrix A
00022     ABIPLinSysWork *L;
00023     ABIPSettings *stgs;
00024     ABIPData *data; //original data
00025     abip_float sparsity;
00026
00027     abip_float *rho_dr; // non-identity DR scaling
00028
00029     /* scaled data */
00030     ABIPMatrix *A;
00031     ABIPMatrix *Q;
00032     abip_float *b;
00033     abip_float *c;
00034     /*-----*/
00035
00036     void (*scaling_data)(lasso *self, ABIPCone *k);
00037     void (*un_scaling_sol)(lasso *self, ABIPSolution *sol);
00038     void (*calc_residuals)(lasso *self, ABIPWork *w, ABIPResiduals *r, abip_int ipm_iter, abip_int
admm_iter);
00039     abip_int (*init_spe_linsys_work)(lasso *self);
00040     abip_int (*solve_spe_linsys)(lasso *self, abip_float *b, abip_float *pcg_warm_start, abip_int
iter, abip_float error_ratio);
00041     void (*free_spe_linsys_work)(lasso *self);
00042     void (*spe_A_times)(lasso *self, const abip_float *x, abip_float *y); //y += Ax, where A is the
reformulated constraint matrix of ABIP
00043     void (*spe_AT_times)(lasso *self, const abip_float *x, abip_float *y); //y += A'x, where A is the
reformulated constraint matrix of ABIP
00044     abip_float (*inner_conv_check)(lasso *self, ABIPWork *w);
00045     /*-----*/
00046
00047     abip_float lambda;
00048     /*-----scaling-----*/
00049     abip_float *D_hat;
00050     abip_float *D;
00051     abip_float *E;
00052     abip_float sc_b;
00053     abip_float sc_c;
00054     abip_float sc;
00055     abip_float sc_cone1;
00056     abip_float sc_cone2;
00057     /*-----*/
00058
00059 };
00060
00061 abip_int init_lasso(lasso **gen_lasso, ABIPData *d, ABIPSettings *stgs);
00062
00063 void scaling_lasso_data(lasso *self, ABIPCone *k);
00064 void un_scaling_lasso_sol(lasso *self, ABIPSolution *sol);
00065 void calc_lasso_residuals(lasso *self, ABIPWork *w, ABIPResiduals *r, abip_int ipm_iter, abip_int
admm_iter);
00066 abip_int init_lasso_linsys_work(lasso *self);
00067 abip_int solve_lasso_linsys(lasso *self, abip_float *b, abip_float *pcg_warm_start, abip_int iter,
abip_float error_ratio);
00068 void free_lasso_linsys_work(lasso *self);
00069 void lasso_A_times(lasso *self, const abip_float *x, abip_float *y); //y += Ax, where A is the
reformulated constraint matrix of ABIP
00070 void lasso_AT_times(lasso *self, const abip_float *x, abip_float *y); //y += A'x, where A is the
reformulated constraint matrix of ABIP
00071 abip_float lasso_inner_conv_check(lasso *self, ABIPWork *w);
00072
00073 #ifdef __cplusplus
00074 }
00075 #endif
00076 #endif

```

## 4.151 include/linalg.h File Reference

```

#include "abip.h"
#include <math.h>
#include "cs.h"

```

### Macros

- #define RowMajor 0
- #define ColMajor 1



## Functions

- void [ABIP\(\)](#) [c\\_dot](#) ([abip\\_float](#) \*x, const [abip\\_float](#) \*y, const [abip\\_int](#) len)  
*Elementwise multiplication of two vectors.*
- [abip\\_float](#) [ABIP\(\)](#) [vec\\_mean](#) ([abip\\_float](#) \*x, [abip\\_int](#) len)  
*Calculate the mean of a vector.*
- void [ABIP\(\)](#) [set\\_as\\_scaled\\_array](#) ([abip\\_float](#) \*x, const [abip\\_float](#) \*a, const [abip\\_float](#) b, [abip\\_int](#) len)  
*Elementwise multiplication of a vector by a scalar.*
- void [ABIP\(\)](#) [set\\_as\\_sqrt](#) ([abip\\_float](#) \*x, const [abip\\_float](#) \*v, [abip\\_int](#) len)  
*Elementwise square root of a vector.*
- void [ABIP\(\)](#) [set\\_as\\_sq](#) ([abip\\_float](#) \*x, const [abip\\_float](#) \*v, [abip\\_int](#) len)  
*Elementwise square of a vector.*
- void [ABIP\(\)](#) [scale\\_array](#) ([abip\\_float](#) \*a, const [abip\\_float](#) b, [abip\\_int](#) len)  
*Elementwise multiplication of a vector by a scalar with replacement.*
- [abip\\_float](#) [ABIP\(\)](#) [dot](#) (const [abip\\_float](#) \*x, const [abip\\_float](#) \*y, [abip\\_int](#) len)  
*Dot product of two vectors.*
- [abip\\_float](#) [ABIP\(\)](#) [norm\\_sq](#) (const [abip\\_float](#) \*v, [abip\\_int](#) len)  
*L2 norm of a vector.*
- [abip\\_float](#) [ABIP\(\)](#) [norm\\_1](#) (const [abip\\_float](#) \*x, const [abip\\_int](#) len)  
*L1 norm of a vector.*
- [abip\\_float](#) [ABIP\(\)](#) [cone\\_norm\\_1](#) (const [abip\\_float](#) \*x, const [abip\\_int](#) len)  
*The absolute value of the largest component of x.*
- [abip\\_float](#) [ABIP\(\)](#) [norm](#) (const [abip\\_float](#) \*v, [abip\\_int](#) len)  
*Square of L2 norm of a vector.*
- [abip\\_float](#) [ABIP\(\)](#) [norm\\_inf](#) (const [abip\\_float](#) \*a, [abip\\_int](#) len)  
*Calculate the maximum absolute value of a vector.*
- void [ABIP\(\)](#) [add\\_array](#) ([abip\\_float](#) \*a, const [abip\\_float](#) b, [abip\\_int](#) len)  
*Elementwise addition of two vectors with coefficients.*
- void [ABIP\(\)](#) [add\\_scaled\\_array](#) ([abip\\_float](#) \*a, const [abip\\_float](#) \*b, [abip\\_int](#) n, const [abip\\_float](#) sc)  
*Elementwise addition of two vectors with coefficients 1.*
- [abip\\_float](#) [ABIP\(\)](#) [norm\\_diff](#) (const [abip\\_float](#) \*a, const [abip\\_float](#) \*b, [abip\\_int](#) len)  
*L2 norm of the difference of two vectors.*
- [abip\\_float](#) [ABIP\(\)](#) [norm\\_inf\\_diff](#) (const [abip\\_float](#) \*a, const [abip\\_float](#) \*b, [abip\\_int](#) len)  
*Maximum of the difference of two vectors.*
- [abip\\_float](#) \*[ABIP\(\)](#) [csc\\_to\\_dense](#) (const [cs](#) \*in\_csc, const [abip\\_int](#) out\_format)  
*Convert a CSC matrix to a dense matrix.*

### 4.151.1 Macro Definition Documentation

#### 4.151.1.1 ColMajor

```
#define ColMajor 1
```

Definition at line 12 of file [linalg.h](#).

#### 4.151.1.2 RowMajor

```
#define RowMajor 0
```

Definition at line 11 of file [linalg.h](#).

### 4.151.2 Function Documentation

#### 4.151.2.1 add\_array()

```
void ABIP() add_array (
    abip_float * a,
    const abip_float b,
    abip_int len )
```

Elementwise addition of two vectors with coefficients.

Definition at line 181 of file [linalg.c](#).

#### 4.151.2.2 add\_scaled\_array()

```
void ABIP() add_scaled_array (
    abip_float * a,
    const abip_float * b,
    abip_int n,
    const abip_float sc )
```

Elementwise addition of two vectors with coefficients 1.

Definition at line 192 of file [linalg.c](#).

#### 4.151.2.3 c\_dot()

```
void ABIP() c_dot (
    abip_float * x,
    const abip_float * y,
    const abip_int len )
```

Elementwise multiplication of two vectors.

Definition at line 9 of file [linalg.c](#).

#### 4.151.2.4 cone\_norm\_1()

```
abip_float ABIP() cone_norm_1 (
    const abip_float * x,
    const abip_int len )
```

The absolute value of the largest component of x.

Definition at line 144 of file [linalg.c](#).

#### 4.151.2.5 csc\_to\_dense()

```
abip_float *ABIP() csc_to_dense (
    const cs * in_csc,
    const abip_int out_format )
```

Convert a CSC matrix to a dense matrix.

Definition at line 247 of file [linalg.c](#).

#### 4.151.2.6 dot()

```
abip_float ABIP() dot (
    const abip_float * x,
    const abip_float * y,
    abip_int len )
```

Dot product of two vectors.

Definition at line 85 of file [linalg.c](#).

#### 4.151.2.7 norm()

```
abip_float ABIP() norm (
    const abip_float * v,
    abip_int len )
```

Square of L2 norm of a vector.

Definition at line 119 of file [linalg.c](#).

#### 4.151.2.8 norm\_1()

```
abip_float ABIP() norm_1 (
    const abip_float * x,
    const abip_int len )
```

L1 norm of a vector.

Definition at line 130 of file [linalg.c](#).

#### 4.151.2.9 norm\_diff()

```
abip_float ABIP() norm_diff (
    const abip_float * a,
    const abip_float * b,
    abip_int len )
```

L2 norm of the difference of two vectors.

Definition at line 207 of file [linalg.c](#).

#### 4.151.2.10 norm\_inf()

```
abip_float ABIP() norm_inf (
    const abip_float * a,
    abip_int len )
```

Calculate the maximum absolute value of a vector.

Definition at line 161 of file [linalg.c](#).

#### 4.151.2.11 norm\_inf\_diff()

```
abip_float ABIP() norm_inf_diff (
    const abip_float * a,
    const abip_float * b,
    abip_int len )
```

Maximum of the difference of two vectors.

Definition at line 223 of file [linalg.c](#).

#### 4.151.2.12 norm\_sq()

```
abip_float ABIP() norm_sq (
    const abip_float * v,
    abip_int len )
```

L2 norm of a vector.

Definition at line 102 of file [linalg.c](#).

#### 4.151.2.13 scale\_array()

```
void ABIP() scale_array (
    abip_float * a,
    const abip_float b,
    abip_int len )
```

Elementwise multiplication of a vector by a scalar with replacement.

Definition at line 71 of file [linalg.c](#).

#### 4.151.2.14 set\_as\_scaled\_array()

```
void ABIP() set_as_scaled_array (
    abip_float * x,
    const abip_float * a,
    const abip_float b,
    abip_int len )
```

Elementwise multiplication of a vector by a scalar.

Definition at line 37 of file [linalg.c](#).

#### 4.151.2.15 set\_as\_sq()

```
void ABIP() set_as_sq (
    abip_float * x,
    const abip_float * v,
    abip_int len )
```

Elementwise square of a vector.

Definition at line 60 of file [linalg.c](#).

#### 4.151.2.16 set\_as\_sqrt()

```
void ABIP() set_as_sqrt (
    abip_float * x,
    const abip_float * v,
    abip_int len )
```

Elementwise square root of a vector.

Definition at line 49 of file [linalg.c](#).

#### 4.151.2.17 vec\_mean()

```
abip_float ABIP() vec_mean (
    abip_float * x,
    abip_int len )
```

Calculate the mean of a vector.

Definition at line 19 of file [linalg.c](#).

## 4.152 linalg.h

[Go to the documentation of this file.](#)

```
00001 #ifndef LINALG_H_GUARD
00002 #define LINALG_H_GUARD
00003
00004 #ifdef __cplusplus
00005 extern "C" {
00006 #endif
00007
00008 #include "abip.h"
00009 #include <math.h>
00010 #include "cs.h"
00011 #define RowMajor 0
00012 #define ColMajor 1
00013
00014 void ABIP(c_dot)
00015 (
00016     abip_float *x,
00017     const abip_float *y,
00018     const abip_int len
00019 );
00020
00021 abip_float ABIP(vec_mean)
00022 (
00023     abip_float *x,
00024     abip_int len
00025 );
00026
00027 void ABIP(set_as_scaled_array)
00028 (
00029     abip_float *x,
00030     const abip_float *a,
00031     const abip_float b,
00032     abip_int len
00033 );
00034
00035 void ABIP(set_as_sqrt)
00036 (
00037     abip_float *x,
00038     const abip_float *v,
00039     abip_int len
00040 );
00041
00042 void ABIP(set_as_sq)
```

```

00043 (
00044     abip_float *x,
00045     const abip_float *v,
00046     abip_int len
00047 );
00048
00049 void ABIP(scale_array)
00050 (
00051     abip_float *a,
00052     const abip_float b,
00053     abip_int len
00054 );
00055
00056
00057 abip_float ABIP(dot)
00058 (
00059     const abip_float *x,
00060     const abip_float *y,
00061     abip_int len
00062 );
00063
00064 abip_float ABIP(norm_sq)
00065 (
00066     const abip_float *v,
00067     abip_int len
00068 );
00069
00070 abip_float ABIP(norm_l)
00071 (
00072     const abip_float *x,
00073     const abip_int len
00074 );
00075
00076 abip_float ABIP(cone_norm_l)
00077 (
00078     const abip_float *x,
00079     const abip_int len
00080 );
00081
00082 abip_float ABIP(norm)
00083 (
00084     const abip_float *v,
00085     abip_int len
00086 );
00087
00088 abip_float ABIP(norm_inf)
00089 (
00090     const abip_float *a,
00091     abip_int len
00092 );
00093
00094 void ABIP(add_array)
00095 (
00096     abip_float *a,
00097     const abip_float b,
00098     abip_int len
00099 );
00100
00101 void ABIP(add_scaled_array)
00102 (
00103     abip_float *a,
00104     const abip_float *b,
00105     abip_int n,
00106     const abip_float sc
00107 );
00108
00109 abip_float ABIP(norm_diff)
00110 (
00111     const abip_float *a,
00112     const abip_float *b,
00113     abip_int len
00114 );
00115
00116 abip_float ABIP(norm_inf_diff)
00117 (
00118     const abip_float *a,
00119     const abip_float *b,
00120     abip_int len
00121 );
00122
00123 abip_float * ABIP(csc_to_dense)(const cs * in_csc, const abip_int out_format);
00124
00125 #ifdef __cplusplus
00126 }
00127 #endif
00128 #endif

```

## 4.153 include/linsys.h File Reference

```
#include <math.h>
#include "abip.h"
#include "amatrix.h"
#include "linalg.h"
#include "glbopts.h"
#include "cs.h"
#include "mkl.h"
#include "mkl_dss.h"
#include "mkl_pardiso.h"
#include "mkl_types.h"
#include "mkl_lapacke.h"
#include "util.h"
#include "qdldl.h"
```

### Classes

- struct [ABIP\\_LIN\\_SYS\\_WORK](#)

### Functions

- void [ABIP\(\)](#) [accum\\_by\\_Atrans](#) (const [ABIPMatrix](#) \*A, const [abip\\_float](#) \*x, [abip\\_float](#) \*y)  
*Add the transposed matrix-vector product to a vector.*
- void [ABIP\(\)](#) [accum\\_by\\_A](#) (const [ABIPMatrix](#) \*A, const [abip\\_float](#) \*x, [abip\\_float](#) \*y)  
*Add the matrix-vector product to a vector.*
- [abip\\_int](#) [ABIP\(\)](#) [validate\\_lin\\_sys](#) (const [ABIPMatrix](#) \*A)  
*Check the validity of the linear system.*
- char \*[ABIP\(\)](#) [get\\_lin\\_sys\\_method](#) ([spe\\_problem](#) \*spe)  
*Get the method used to solve the linear system.*
- char \*[ABIP\(\)](#) [get\\_lin\\_sys\\_summary](#) ([spe\\_problem](#) \*self, [ABIPInfo](#) \*info)  
*Get the summary information of the linear system.*
- void [ABIP\(\)](#) [free\\_A\\_matrix](#) ([ABIPMatrix](#) \*A)  
*Free the memory of a matrix.*
- [abip\\_int](#) [ABIP\(\)](#) [copy\\_A\\_matrix](#) ([ABIPMatrix](#) \*\*dstp, const [ABIPMatrix](#) \*src)  
*Copy a matrix.*
- [abip\\_int](#) [LDL\\_factor](#) ([cs](#) \*A, [cs](#) \*\*L, [abip\\_float](#) \*Dinv)
- [abip\\_int](#) [ABIP\(\)](#) [init\\_linsys\\_work](#) ([spe\\_problem](#) \*spe)  
*Initialize linear system solver work space.*
- [abip\\_int](#) [ABIP\(\)](#) [solve\\_linsys](#) ([spe\\_problem](#) \*spe, [abip\\_float](#) \*b, [abip\\_int](#) n, [abip\\_float](#) \*pcg\_warm\_start, [abip\\_float](#) pcg\_tol)  
*solve linear system according to the specific linsys solver*
- [abip\\_int](#) [ABIP\(\)](#) [free\\_linsys](#) ([spe\\_problem](#) \*spe)  
*free memory for linear system solver*

### 4.153.1 Function Documentation



#### 4.153.1.1 accum\_by\_A()

```
void ABIP() accum_by_A (
    const ABIPMatrix * A,
    const abip_float * x,
    abip_float * y )
```

Add the matrix-vector product to a vector.

Definition at line 229 of file [linsys.c](#).

#### 4.153.1.2 accum\_by\_Atrans()

```
void ABIP() accum_by_Atrans (
    const ABIPMatrix * A,
    const abip_float * x,
    abip_float * y )
```

Add the transposed matrix-vector product to a vector.

Definition at line 191 of file [linsys.c](#).

#### 4.153.1.3 copy\_A\_matrix()

```
abip_int ABIP() copy_A_matrix (
    ABIPMatrix ** dstp,
    const ABIPMatrix * src )
```

Copy a matrix.

Definition at line 12 of file [linsys.c](#).

#### 4.153.1.4 free\_A\_matrix()

```
void ABIP() free_A_matrix (
    ABIPMatrix * A )
```

Free the memory of a matrix.

Definition at line 152 of file [linsys.c](#).

#### 4.153.1.5 free\_linsys()

```
abip_int ABIP() free_linsys (
    spe_problem * spe )
```

free memory for linear system solver

Definition at line 1181 of file [linsys.c](#).

#### 4.153.1.6 get\_lin\_sys\_method()

```
char *ABIP() get_lin_sys_method (
    spe_problem * spe )
```

Get the method used to solve the linear system.

Definition at line 39 of file [linsys.c](#).

#### 4.153.1.7 get\_lin\_sys\_summary()

```
char *ABIP() get_lin_sys_summary (
    spe_problem * self,
    ABIPInfo * info )
```

Get the summary information of the linear system.

Definition at line 71 of file [linsys.c](#).

#### 4.153.1.8 init\_linsys\_work()

```
abip_int ABIP() init_linsys_work (
    spe_problem * spe )
```

Initialize linear system solver work space.

Definition at line 1027 of file [linsys.c](#).

#### 4.153.1.9 LDL\_factor()

```
abip_int LDL_factor (
    cs * A,
    cs ** L,
    abip_float * Dinv )
```

Definition at line 542 of file [linsys.c](#).

**4.153.1.10 solve\_linsys()**

```

abip_int ABIP() solve_linsys (
    spe_problem * spe,
    abip_float * b,
    abip_int n,
    abip_float * pcg_warm_start,
    abip_float pcg_tol )

```

solve linear system according to the specific linsys solver

Definition at line 1141 of file [linsys.c](#).

**4.153.1.11 validate\_lin\_sys()**

```

abip_int ABIP() validate_lin_sys (
    const ABIPMatrix * A )

```

Check the validity of the linear system.

Definition at line 102 of file [linsys.c](#).

**4.154 linsys.h**

[Go to the documentation of this file.](#)

```

00001 #ifndef LINSYS_H_GUARD
00002 #define LINSYS_H_GUARD
00003
00004 #ifdef __cplusplus
00005 extern "C" {
00006 #endif
00007
00008 #include <math.h>
00009 #include "abip.h"
00010 #include "amatrix.h"
00011 #include "linalg.h"
00012 #include "glbopts.h"
00013 #include "cs.h"
00014 #include "mkl.h"
00015 #include "mkl_dss.h"
00016 #include "mkl_pardiso.h"
00017 #include "mkl_types.h"
00018 #include "mkl_lapacke.h"
00019 #include "util.h"
00020 #include "qdldl.h"
00021
00022
00023 struct ABIP_LIN_SYS_WORK
00024 {
00025     abip_float total_solve_time;
00026
00027     //matrix to factorize
00028     cs *K;
00029
00030     //mkl-dss
00031     _MKL_DSS_HANDLE_t handle;
00032
00033
00034     //mkl-pardiso
00035     void *pt[64];
00036     MKL_INT iparm[64];
00037     MKL_INT maxfct, mnum, error, msglvl;
00038     abip_float ddum; /* Double dummy */
00039     MKL_INT idum;
00040     MKL_INT mtype;

```

```

00041
00042
00043
00044 //pcg
00045 abip_float* M; //preconditioner for pcg
00046 abip_int total_cg_iters;
00047
00048
00049 //sparse cholesky
00050 css *S ;
00051 csn *N ;
00052
00053
00054 //qdldl
00055 cs *L; //matrix L of LDL' factorization
00056 abip_float *Dinv; //the diagonal vector of the diagonal matrix D L of LDL' factorization
00057 abip_int nnz_LDL;
00058 abip_int *P; //permutation for KKT matrix
00059 abip_float *bp;
00060
00061
00062 //lapack dense cholesky
00063 // A = UTU
00064 abip_float *U;
00065 };
00066
00067 /* y += A'*x
00068 A in column compressed format
00069 parallelizes over columns (rows of A')
00070 */
00071 void ABIP(accum_by_Atrans)
00072 (
00073     const ABIPMatrix* A,
00074     const abip_float* x,
00075     abip_float* y
00076 );
00077
00078 /* y += A*x
00079 A in column compressed format
00080 this parallelizes over columns and uses
00081 pragma atomic to prevent concurrent writes to y
00082 */
00083 void ABIP(accum_by_A)
00084 (
00085     const ABIPMatrix* A,
00086     const abip_float* x,
00087     abip_float* y
00088 );
00089
00090
00091
00092 abip_int ABIP(validate_lin_sys)
00093 (
00094     const ABIPMatrix *A
00095 );
00096
00097 char *ABIP(get_lin_sys_method)
00098 (
00099     spe_problem *spe
00100 );
00101
00102 char *ABIP(get_lin_sys_summary)
00103 (
00104     spe_problem *self,
00105     ABIPInfo *info
00106 );
00107
00108 void ABIP(free_A_matrix)
00109 (
00110     ABIPMatrix *A
00111 );
00112
00113 abip_int ABIP(copy_A_matrix)
00114 (
00115     ABIPMatrix **dstp,
00116     const ABIPMatrix *src
00117 );
00118
00119 abip_int LDL_factor(cs *A, cs **L, abip_float *Dinv);
00120
00121 abip_int ABIP(init_linsys_work)(spe_problem *spe);
00122
00123
00124 abip_int ABIP(solve_linsys)(spe_problem *spe, abip_float *b, abip_int n, abip_float *pcg_warm_start,
00125     abip_float pcg_tol);
00126

```

```

00127 abip_int ABIP (free_linsys) (spe_problem *spe);
00128 #ifdef __cplusplus
00129 }
00130 #endif
00131
00132 #endif

```

## 4.155 include/qcp\_config.h File Reference

```

#include "glbopts.h"
#include "abip.h"
#include "amatrix.h"
#include "linsys.h"

```

### Classes

- struct [qcp](#)

### Typedefs

- typedef struct [qcp](#) [qcp](#)

### Functions

- [abip\\_int init\\_qcp](#) ([qcp \\*\\*QCP](#), [ABIPData \\*d](#), [ABIPSettings \\*stgs](#))  
*Initialize the qcp problem structure.*
- void [scaling\\_qcp\\_data](#) ([qcp \\*self](#), [ABIPCone \\*k](#))  
*Scale the data for the qcp problem.*
- void [un\\_scaling\\_qcp\\_sol](#) ([qcp \\*self](#), [ABIPSolution \\*sol](#))  
*Get the unscaled solution of the general qcp problem.*
- void [calc\\_qcp\\_residuals](#) ([qcp \\*self](#), [ABIPWork \\*w](#), [ABIPResiduals \\*r](#), [abip\\_int ipm\\_iter](#), [abip\\_int admm\\_iter](#))  
*Calculate the residuals of the general qcp problem.*
- [abip\\_int init\\_qcp\\_linsys\\_work](#) ([qcp \\*self](#))  
*Initialize the linear system solver work space for the general qcp problem.*
- [abip\\_int solve\\_qcp\\_linsys](#) ([qcp \\*self](#), [abip\\_float \\*b](#), [abip\\_float \\*pcg\\_warm\\_start](#), [abip\\_int iter](#), [abip\\_float error\\_ratio](#))  
*Linear system solver for the general qcp problem.*
- void [free\\_qcp\\_linsys\\_work](#) ([qcp \\*self](#))  
*Free the linear system solver work space for the general qcp problem.*
- void [qcp\\_A\\_times](#) ([qcp \\*self](#), const [abip\\_float \\*x](#), [abip\\_float \\*y](#))  
*Matrix-vector multiplication for the general qcp problem with A untransposed.*
- void [qcp\\_AT\\_times](#) ([qcp \\*self](#), const [abip\\_float \\*x](#), [abip\\_float \\*y](#))  
*Matrix-vector multiplication for the general qcp problem with A transposed.*
- [abip\\_float qcp\\_inner\\_conv\\_check](#) ([qcp \\*self](#), [ABIPWork \\*w](#))  
*Check whether the inner loop of the genral qcp problem has converged.*

#### 4.155.1 Typedef Documentation

#### 4.155.1.1 qcp

```
typedef struct qcp qcp
```

Definition at line 12 of file [qcp\\_config.h](#).

### 4.155.2 Function Documentation

#### 4.155.2.1 calc\_qcp\_residuals()

```
void calc_qcp_residuals (
    qcp * self,
    ABIPWork * w,
    ABIPResiduals * r,
    abip_int ipm_iter,
    abip_int admm_iter )
```

Calculate the residuals of the general qcp problem.

Definition at line 562 of file [qcp\\_config.c](#).

#### 4.155.2.2 free\_qcp\_linsys\_work()

```
void free_qcp_linsys_work (
    qcp * self )
```

Free the linear system solver work space for the general qcp problem.

Definition at line 886 of file [qcp\\_config.c](#).

#### 4.155.2.3 init\_qcp()

```
abip_int init_qcp (
    qcp ** QCP,
    ABIPData * d,
    ABIPSettings * stgs )
```

Initialize the qcp problem structure.

Definition at line 8 of file [qcp\\_config.c](#).

#### 4.155.2.4 init\_qcp\_linsys\_work()

```
abip_int init_qcp_linsys_work (
    qcp * self )
```

Initialize the linear system solver work space for the general qcp problem.

Definition at line 799 of file [qcp\\_config.c](#).

#### 4.155.2.5 qcp\_A\_times()

```
void qcp_A_times (
    qcp * self,
    const abip_float * x,
    abip_float * y )
```

Matrix-vector multiplication for the general qcp problem with A untransposed.

Definition at line 72 of file [qcp\\_config.c](#).

#### 4.155.2.6 qcp\_AT\_times()

```
void qcp_AT_times (
    qcp * self,
    const abip_float * x,
    abip_float * y )
```

Matrix-vector multiplication for the general qcp problem with A transposed.

Definition at line 82 of file [qcp\\_config.c](#).

#### 4.155.2.7 qcp\_inner\_conv\_check()

```
abip_float qcp_inner_conv_check (
    qcp * self,
    ABIPWork * w )
```

Check whether the inner loop of the genral qcp problem has converged.

Definition at line 518 of file [qcp\\_config.c](#).

#### 4.155.2.8 scaling\_qcp\_data()

```
void scaling_qcp_data (
    qcp * self,
    ABIPCone * k )
```

Scale the data for the qcp problem.

Definition at line 91 of file [qcp\\_config.c](#).

#### 4.155.2.9 solve\_qcp\_linsys()

```
abip_int solve_qcp_linsys (
    qcp * self,
    abip_float * b,
    abip_float * pcg_warm_start,
    abip_int iter,
    abip_float error_ratio )
```

Linear system solver for the general qcp problem.

Definition at line 826 of file [qcp\\_config.c](#).

#### 4.155.2.10 un\_scaling\_qcp\_sol()

```
void un_scaling_qcp_sol (
    qcp * self,
    ABIPSolution * sol )
```

Get the unscaled solution of the general qcp problem.

Definition at line 496 of file [qcp\\_config.c](#).

### 4.156 qcp\_config.h

[Go to the documentation of this file.](#)

```
00001 #ifndef QCP_CONFIG_H_GUARD
00002 #define QCP_CONFIG_H_GUARD
00003
00004 #ifdef __cplusplus
00005 extern "C" {
00006 #endif
00007
00008 #include "glbopts.h"
00009 #include "abip.h"
00010 #include "amatrix.h"
00011 #include "linsys.h"
00012 typedef struct qcp qcp;
00013
00014 struct qcp{
00015
00016     /*-----common parts-----*/
00017     enum problem_type pro_type;
00018     abip_int m; //rows of input data A
```



```

00019     abip_int n; //cols of input data A
00020     abip_int p; //rows of ABIP constraint matrix A
00021     abip_int q; //cols of ABIP constraint matrix A
00022     ABIPLinSysWork *L;
00023     ABIPSettings *stgs;
00024     ABIPData *data; //original data
00025     abip_float sparsity;
00026
00027     abip_float *rho_dr; // non-identity DR scaling
00028
00029     /* scaled data */
00030     ABIPMatrix *A;
00031     ABIPMatrix *Q;
00032     abip_float *b;
00033     abip_float *c;
00034     /*-----*/
00035
00036     void (*scaling_data)(qcp *self, ABIPCone *k);
00037     void (*un_scaling_sol)(qcp *self, ABIPSolution *sol);
00038     void (*calc_residuals)(qcp *self, ABIPWork *w, ABIPResiduals *r, abip_int ipm_iter, abip_int
admm_iter);
00039     abip_int (*init_spe_linsys_work)(qcp *self);
00040     abip_int (*solve_spe_linsys)(qcp *self, abip_float *b, abip_float *pcg_warm_start, abip_int iter,
abip_float error_ratio);
00041     void (*free_spe_linsys_work)(qcp *self);
00042     void (*spe_A_times)(qcp *self, const abip_float *x, abip_float *y); //y += Ax, where A is the
reformulated constraint matrix of ABIP
00043     void (*spe_AT_times)(qcp *self, const abip_float *x, abip_float *y); //y += A'x, where A is the
reformulated constraint matrix of ABIP
00044     abip_float (*inner_conv_check)(qcp *self, ABIPWork *w);
00045     /*-----*/
00046
00047     /*-----scaling-----*/
00048     abip_float *D;
00049     abip_float *E;
00050     abip_float sc_b;
00051     abip_float sc_c;
00052     /*-----*/
00053
00054 };
00055
00056 abip_int init_qcp(qcp **QCP, ABIPData *d, ABIPSettings *stgs);
00057
00058 void scaling_qcp_data(qcp *self, ABIPCone *k);
00059 void un_scaling_qcp_sol(qcp *self, ABIPSolution *sol);
00060 void calc_qcp_residuals(qcp *self, ABIPWork *w, ABIPResiduals *r, abip_int ipm_iter, abip_int
admm_iter);
00061 abip_int init_qcp_linsys_work(qcp *self);
00062 abip_int solve_qcp_linsys(qcp *self, abip_float *b, abip_float *pcg_warm_start, abip_int iter,
abip_float error_ratio);
00063 void free_qcp_linsys_work(qcp *self);
00064 void qcp_A_times(qcp *self, const abip_float *x, abip_float *y); //y += Ax, where A is the reformulated
constraint matrix of ABIP
00065 void qcp_AT_times(qcp *self, const abip_float *x, abip_float *y); //y += A'x, where A is the
reformulated constraint matrix of ABIP
00066 abip_float qcp_inner_conv_check(qcp *self, ABIPWork *w);
00067 #ifdef __cplusplus
00068 }
00069 #endif
00070 #endif

```

## 4.157 include/svm\_config.h File Reference

```

#include "glbopts.h"
#include "abip.h"
#include "amatrix.h"
#include "linsys.h"

```

### Classes

- struct [Svm](#)

## Typedefs

- typedef struct [Svm](#) [svm](#)

## Functions

- [abip\\_int](#) [init\\_svm](#) ([svm](#) \*\*gen\_svm, [ABIPData](#) \*d, [ABIPSettings](#) \*stgs)  
*Initialize the svm socp formulation structure.*
- void [scaling\\_svm\\_data](#) ([svm](#) \*self, [ABIPConc](#) \*k)  
*Customized scaling procedure for the svm socp formulation.*
- void [un\\_scaling\\_svm\\_sol](#) ([svm](#) \*self, [ABIPSolution](#) \*sol)  
*Get the unscaled solution of the original svm problem.*
- void [calc\\_svm\\_residuals](#) ([svm](#) \*self, [ABIPWork](#) \*w, [ABIPResiduals](#) \*r, [abip\\_int](#) ipm\_iter, [abip\\_int](#) admm\_iter)  
*Calculate the residuals of the svm socp formulation.*
- [abip\\_int](#) [init\\_svm\\_linsys\\_work](#) ([svm](#) \*self)  
*Initialize the linear system solver work space for the svm socp formulation.*
- [abip\\_int](#) [solve\\_svm\\_linsys](#) ([svm](#) \*self, [abip\\_float](#) \*b, [abip\\_float](#) \*pcg\_warm\_start, [abip\\_int](#) iter, [abip\\_float](#) error\_ratio)  
*Customized linear system solver for the svm socp formulation.*
- void [free\\_svm\\_linsys\\_work](#) ([svm](#) \*self)  
*Free the linear system solver work space for the svm socp formulation.*
- void [svm\\_A\\_times](#) ([svm](#) \*self, const [abip\\_float](#) \*x, [abip\\_float](#) \*y)  
*Customized matrix-vector multiplication for the svm socp formulation with A untransposed.*
- void [svm\\_AT\\_times](#) ([svm](#) \*self, const [abip\\_float](#) \*x, [abip\\_float](#) \*y)  
*Customized matrix-vector multiplication for the svm socp formulation with A transposed.*
- [abip\\_float](#) [svm\\_inner\\_conv\\_check](#) ([svm](#) \*self, [ABIPWork](#) \*w)  
*Check whether the inner loop of the svm socp formulation has converged.*

## 4.157.1 Typedef Documentation

### 4.157.1.1 svm

```
typedef struct Svm svm
```

Definition at line 12 of file [svm\\_config.h](#).

## 4.157.2 Function Documentation

#### 4.157.2.1 calc\_svm\_residuals()

```
void calc_svm_residuals (
    svm * self,
    ABIPWork * w,
    ABIPResiduals * r,
    abip_int ipm_iter,
    abip_int admm_iter )
```

Calculate the residuals of the svm socp formulation.

Definition at line 445 of file [svm\\_config.c](#).

#### 4.157.2.2 free\_svm\_linsys\_work()

```
void free_svm_linsys_work (
    svm * self )
```

Free the linear system solver work space for the svm socp formulation.

Definition at line 812 of file [svm\\_config.c](#).

#### 4.157.2.3 init\_svm()

```
abip_int init_svm (
    svm ** gen_svm,
    ABIPData * d,
    ABIPSettings * stgs )
```

Initialize the svm socp formulation structure.

Definition at line 8 of file [svm\\_config.c](#).

#### 4.157.2.4 init\_svm\_linsys\_work()

```
abip_int init_svm_linsys_work (
    svm * self )
```

Initialize the linear system solver work space for the svm socp formulation.

Definition at line 702 of file [svm\\_config.c](#).

#### 4.157.2.5 scaling\_svm\_data()

```
void scaling_svm_data (
    svm * self,
    ABIPConc * k )
```

Customized scaling procedure for the svm socp formulation.

Definition at line 283 of file [svm\\_config.c](#).

#### 4.157.2.6 solve\_svm\_linsys()

```
abip_int solve_svm_linsys (
    svm * self,
    abip_float * b,
    abip_float * pcg_warm_start,
    abip_int iter,
    abip_float error_ratio )
```

Customized linear system solver for the svm socp formulation.

Definition at line 728 of file [svm\\_config.c](#).

#### 4.157.2.7 svm\_A\_times()

```
void svm_A_times (
    svm * self,
    const abip_float * x,
    abip_float * y )
```

Customized matrix-vector multiplication for the svm socp formulation with A untransposed.

Definition at line 175 of file [svm\\_config.c](#).

#### 4.157.2.8 svm\_AT\_times()

```
void svm_AT_times (
    svm * self,
    const abip_float * x,
    abip_float * y )
```

Customized matrix-vector multiplication for the svm socp formulation with A transposed.

Definition at line 202 of file [svm\\_config.c](#).

## 4.157.2.9 svm\_inner\_conv\_check()

```
abip_float svm_inner_conv_check (
    svm * self,
    ABIPWork * w )
```

Check whether the inner loop of the svm socp formulation has converged.

Definition at line 234 of file [svm\\_config.c](#).

## 4.157.2.10 un\_scaling\_svm\_sol()

```
void un_scaling_svm_sol (
    svm * self,
    ABIPSolution * sol )
```

Get the unscaled solution of the original svm problem.

Definition at line 413 of file [svm\\_config.c](#).

## 4.158 svm\_config.h

[Go to the documentation of this file.](#)

```
00001 #ifndef SVM_CONFIG_H_GUARD
00002 #define SVM_CONFIG_H_GUARD
00003
00004 #ifdef __cplusplus
00005 extern "C" {
00006 #endif
00007
00008 #include "glbopts.h"
00009 #include "abip.h"
00010 #include "amatrix.h"
00011 #include "linsys.h"
00012 typedef struct Svm svm;
00013
00014 struct Svm{
00015
00016     /*-----common parts-----*/
00017     enum problem_type pro_type;
00018     abip_int m; //rows of input data A
00019     abip_int n; //cols of input data A
00020     abip_int p; //rows of ABIP constraint matrix A
00021     abip_int q; //cols of ABIP constraint matrix A
00022     ABIPLinSysWork *L;
00023     ABIPSettings *stgs;
00024     ABIPData *data; //original data
00025     abip_float sparsity;
00026
00027     abip_float *rho_dr; // non-identity DR scaling
00028
00029     /* scaled data */
00030     ABIPMatrix *A;
00031     ABIPMatrix *Q;
00032     abip_float *b;
00033     abip_float *c;
00034     /*-----*/
00035
00036     void (*scaling_data)(svm *self, ABIPConc *k);
00037     void (*un_scaling_sol)(svm *self, ABIPSolution *sol);
00038     void (*calc_residuals)(svm *self, ABIPWork *w, ABIPResiduals *r, abip_int ipm_iter, abip_int
admm_iter);
00039     abip_int (*init_spe_linsys_work)(svm *self);
00040     abip_int (*solve_spe_linsys)(svm *self, abip_float *b, abip_float *pcg_warm_start, abip_int iter,
abip_float error_ratio);
00041     void (*free_spe_linsys_work)(svm *self);
```

```

00042     void (*spe_A_times)(svm *self, const abip_float *x, abip_float *y); //y += Ax, where A is the
reformulated constraint matrix of ABIP
00043     void (*spe_AT_times)(svm *self, const abip_float *x, abip_float *y); //y += A'x, where A is the
reformulated constraint matrix of ABIP
00044     abip_float (*inner_conv_check)(svm *self, ABIPWork *w);
00045     /*-----*/
00046
00047     abip_float lambda; // 'C' in svm
00048     /*-----scaling-----*/
00049     abip_float *sc_D;
00050     abip_float *sc_E;
00051     abip_float *sc_F;
00052     abip_float sc_b;
00053     abip_float sc_c;
00054     abip_float sc;
00055     abip_float sc_cone1;
00056     abip_float sc_cone2;
00057     /*-----*/
00058     ABIPMatrix *wA;
00059     abip_float *wy;
00060     abip_float *wB;
00061     abip_float *wC;
00062     abip_float *wD;
00063     abip_float *wE;
00064     abip_float *wF;
00065     abip_float *wG;
00066     abip_float *wH;
00067     ABIPMatrix *wX;
00068 };
00069
00070 abip_int init_svm(svm **gen_svm, ABIPData *d, ABIPSettings *stgs);
00071
00072 void scaling_svm_data(svm *self, ABIPConc *k);
00073 void un_scaling_svm_sol(svm *self, ABIPSolution *sol);
00074 void calc_svm_residuals(svm *self, ABIPWork *w, ABIPResiduals *r, abip_int ipm_iter, abip_int
admm_iter);
00075 abip_int init_svm_linsys_work(svm *self);
00076 abip_int solve_svm_linsys(svm *self, abip_float *b, abip_float *pcg_warm_start, abip_int iter,
abip_float error_ratio);
00077 void free_svm_linsys_work(svm *self);
00078 void svm_A_times(svm *self, const abip_float *x, abip_float *y); //y += Ax, where A is the reformulated
constraint matrix of ABIP
00079 void svm_AT_times(svm *self, const abip_float *x, abip_float *y); //y += A'x, where A is the
reformulated constraint matrix of ABIP
00080 abip_float svm_inner_conv_check(svm *self, ABIPWork *w);
00081
00082 #ifdef __cplusplus
00083 }
00084 #endif
00085 #endif

```

## 4.159 include/svm\_qp\_config.h File Reference

```

#include "glbopts.h"
#include "abip.h"
#include "amatrix.h"
#include "linsys.h"

```

### Classes

- struct [SVMqp](#)

### Typedefs

- typedef struct [SVMqp](#) svmqp

## Functions

- [abip\\_int init\\_svmqp](#) ([svmqp](#) \*\*gen\_svm, [ABIPData](#) \*d, [ABIPSettings](#) \*stgs)  
*Initialize the svm qp formulation structure.*
- void [scaling\\_svmqp\\_data](#) ([svmqp](#) \*self, [ABIPConc](#) \*k)  
*Customized scaling procedure for the svm qp formulation.*
- void [un\\_scaling\\_svmqp\\_sol](#) ([svmqp](#) \*self, [ABIPSolution](#) \*sol)  
*Get the unscaled solution of the original svm problem.*
- void [calc\\_svmqp\\_residuals](#) ([svmqp](#) \*self, [ABIPWork](#) \*w, [ABIPResiduals](#) \*r, [abip\\_int](#) ipm\_iter, [abip\\_int](#) admm\_iter)  
*Calculate the residuals of the svm qp formulation.*
- [abip\\_int init\\_svmqp\\_linsys\\_work](#) ([svmqp](#) \*self)  
*Initialize the linear system solver work space for the svm qp formulation.*
- [abip\\_int solve\\_svmqp\\_linsys](#) ([svmqp](#) \*self, [abip\\_float](#) \*b, [abip\\_float](#) \*pcg\_warm\_start, [abip\\_int](#) iter, [abip\\_float](#) error\_ratio)  
*Customized linear system solver for the svm qp formulation.*
- void [free\\_svmqp\\_linsys\\_work](#) ([svmqp](#) \*self)  
*Free the linear system solver work space for the svm qp formulation.*
- void [svmqp\\_A\\_times](#) ([svmqp](#) \*self, const [abip\\_float](#) \*x, [abip\\_float](#) \*y)  
*Customized matrix-vector multiplication for the svm qp formulation with A untransposed.*
- void [svmqp\\_AT\\_times](#) ([svmqp](#) \*self, const [abip\\_float](#) \*x, [abip\\_float](#) \*y)  
*Customized matrix-vector multiplication for the svm qp formulation with A transposed.*
- [abip\\_float svmqp\\_inner\\_conv\\_check](#) ([svmqp](#) \*self, [ABIPWork](#) \*w)  
*Check whether the inner loop of the svm qp formulation has converged.*

### 4.159.1 Typedef Documentation

#### 4.159.1.1 svmqp

```
typedef struct SVMqp svmqp
```

Definition at line 12 of file [svm\\_qp\\_config.h](#).

### 4.159.2 Function Documentation

#### 4.159.2.1 calc\_svmqp\_residuals()

```
void calc_svmqp_residuals (
    svmqp * self,
    ABIPWork * w,
    ABIPResiduals * r,
    abip_int ipm_iter,
    abip_int admm_iter )
```

Calculate the residuals of the svm qp formulation.

Definition at line 628 of file [svm\\_qp\\_config.c](#).

#### 4.159.2.2 `free_svmqp_linsys_work()`

```
void free_svmqp_linsys_work (
    svmqp * self )
```

Free the linear system solver work space for the svm qp formulation.

Definition at line 1002 of file [svm\\_qp\\_config.c](#).

#### 4.159.2.3 `init_svmqp()`

```
abip_int init_svmqp (
    svmqp ** gen_svm,
    ABIPData * d,
    ABIPSettings * stgs )
```

Initialize the svm qp formulation structure.

Definition at line 8 of file [svm\\_qp\\_config.c](#).

#### 4.159.2.4 `init_svmqp_linsys_work()`

```
abip_int init_svmqp_linsys_work (
    svmqp * self )
```

Initialize the linear system solver work space for the svm qp formulation.

Definition at line 868 of file [svm\\_qp\\_config.c](#).

#### 4.159.2.5 `scaling_svmqp_data()`

```
void scaling_svmqp_data (
    svmqp * self,
    ABIPCone * k )
```

Customized scaling procedure for the svm qp formulation.

Definition at line 196 of file [svm\\_qp\\_config.c](#).



#### 4.159.2.6 solve\_svmqp\_linsys()

```
abip_int solve_svmqp_linsys (
    svmqp * self,
    abip_float * b,
    abip_float * pcg_warm_start,
    abip_int iter,
    abip_float error_ratio )
```

Customized linear system solver for the svm qp formulation.

Definition at line 894 of file [svm\\_qp\\_config.c](#).

#### 4.159.2.7 svmqp\_A\_times()

```
void svmqp_A_times (
    svmqp * self,
    const abip_float * x,
    abip_float * y )
```

Customized matrix-vector multiplication for the svm qp formulation with A untransposed.

Definition at line 128 of file [svm\\_qp\\_config.c](#).

#### 4.159.2.8 svmqp\_AT\_times()

```
void svmqp_AT_times (
    svmqp * self,
    const abip_float * x,
    abip_float * y )
```

Customized matrix-vector multiplication for the svm qp formulation with A transposed.

Definition at line 141 of file [svm\\_qp\\_config.c](#).

#### 4.159.2.9 svmqp\_inner\_conv\_check()

```
abip_float svmqp_inner_conv_check (
    svmqp * self,
    ABIPWork * w )
```

Check whether the inner loop of the svm qp formulation has converged.

Definition at line 152 of file [svm\\_qp\\_config.c](#).

#### 4.159.2.10 un\_scaling\_svmqp\_sol()

```
void un_scaling_svmqp_sol (
    svmqp * self,
    ABIPSolution * sol )
```

Get the unscaled solution of the original svm problem.

Definition at line 597 of file [svm\\_qp\\_config.c](#).

### 4.160 svm\_qp\_config.h

[Go to the documentation of this file.](#)

```
00001 #ifndef SVM_QP_CONFIG_H_GUARD
00002 #define SVM_QP_CONFIG_H_GUARD
00003
00004 #ifdef __cplusplus
00005 extern "C" {
00006 #endif
00007
00008 #include "glbopts.h"
00009 #include "abip.h"
00010 #include "amatrix.h"
00011 #include "linsys.h"
00012 typedef struct SVMqp svmqp;
00013
00014 struct SVMqp{
00015
00016     /*-----common parts-----*/
00017     enum problem_type pro_type;
00018     abip_int m; //rows of input data A
00019     abip_int n; //cols of input data A
00020     abip_int p; //rows of ABIP constraint matrix A
00021     abip_int q; //cols of ABIP constraint matrix A
00022     ABIPLinSysWork *L;
00023     ABIPSettings *stgs;
00024     ABIPData *data; //original data
00025     abip_float sparsity;
00026
00027     abip_float *rho_dr; // non-identity DR scaling
00028
00029     /* scaled data */
00030     ABIPMatrix *A;
00031     ABIPMatrix *Q;
00032     abip_float *b;
00033     abip_float *c;
00034     /*-----*/
00035
00036     void (*scaling_data)(svmqp *self, ABIPConc *k);
00037     void (*un_scaling_sol)(svmqp *self, ABIPSolution *sol);
00038     void (*calc_residuals)(svmqp *self, ABIPWork *w, ABIPResiduals *r, abip_int ipm_iter, abip_int
00039 admm_iter);
00039     abip_int (*init_spe_linsys_work)(svmqp *self);
00040     abip_int (*solve_spe_linsys)(svmqp *self, abip_float *b, abip_float *pcg_warm_start, abip_int
00041 iter, abip_float error_ratio);
00041     void (*free_spe_linsys_work)(svmqp *self);
00042     void (*spe_A_times)(svmqp *self, const abip_float *x, abip_float *y); //y += Ax, where A is the
00043 reformulated constraint matrix of ABIP
00043     void (*spe_AT_times)(svmqp *self, const abip_float *x, abip_float *y); //y += A'x, where A is the
00044 reformulated constraint matrix of ABIP
00044     abip_float (*inner_conv_check)(svmqp *self, ABIPWork *w);
00045     /*-----*/
00046
00047     abip_float lambda;
00048     /*-----scaling-----*/
00049     abip_float *D;
00050     abip_float *E;
00051     abip_float *F;
00052     abip_float *H;
00053     abip_float sc_b;
00054     abip_float sc_c;
00055 };
00056
00057 abip_int init_svmqp(svmqp **gen_svm, ABIPData *d, ABIPSettings *stgs);
00058
00059 void scaling_svmqp_data(svmqp *self, ABIPConc *k);
```

```

00060 void un_scaling_svmqp_sol(svmqp *self, ABIPSolution *sol);
00061 void calc_svmqp_residuals(svmqp *self, ABIPWork *w, ABIPResiduals *r, abip_int ipm_iter, abip_int
    admm_iter);
00062 abip_int init_svmqp_linsys_work(svmqp *self);
00063 abip_int solve_svmqp_linsys(svmqp *self, abip_float *b, abip_float *pcg_warm_start, abip_int iter,
    abip_float error_ratio);
00064 void free_svmqp_linsys_work(svmqp *self);
00065 void svmqp_A_times(svmqp *self, const abip_float *x, abip_float *y); //y += Ax, where A is the
    reformulated constraint matrix of ABIP
00066 void svmqp_AT_times(svmqp *self, const abip_float *x, abip_float *y); //y += A'x, where A is the
    reformulated constraint matrix of ABIP
00067 abip_float svmqp_inner_conv_check(svmqp *self, ABIPWork *w);
00068
00069 #ifdef __cplusplus
00070 }
00071 #endif
00072 #endif

```

## 4.161 include/util.h File Reference

```

#include "abip.h"
#include <stdlib.h>
#include <stdio.h>
#include <time.h>

```

### Functions

- struct [ABIP](#) (timer)
- void [ABIP\(\)](#) [tic](#) ([ABIP](#)(timer) \*t)
- [abip\\_float](#) [ABIP\(\)](#) [toc](#) ([ABIP](#)(timer) \*t)
- [abip\\_float](#) [ABIP\(\)](#) [str\\_toc](#) (char \*str, [ABIP](#)(timer) \*t)
- [abip\\_float](#) [ABIP\(\)](#) [tocq](#) ([ABIP](#)(timer) \*t)
- void [ABIP\(\)](#) [print\\_data](#) (const [ABIPData](#) \*d)
- void [ABIP\(\)](#) [print\\_work](#) (const [ABIPWork](#) \*w)
- void [ABIP\(\)](#) [print\\_array](#) (const [abip\\_float](#) \*arr, [abip\\_int](#) n, const char \*name)
- void [ABIP\(\)](#) [set\\_default\\_settings](#) ([ABIPData](#) \*d)

*Default parameter settings.*

- void [ABIP\(\)](#) [free\\_info](#) ([ABIPInfo](#) \*info)
- void [ABIP\(\)](#) [free\\_sol](#) ([ABIPSolution](#) \*sol)
- void [ABIP\(\)](#) [free\\_data](#) ([ABIPData](#) \*d)
- void [ABIP\(\)](#) [free\\_cone](#) ([ABIPCone](#) \*k)

### 4.161.1 Function Documentation

#### 4.161.1.1 [ABIP\(\)](#)

```

ABIP (
    timer
)

```

Definition at line 1 of file [util.h](#).

#### 4.161.1.2 free\_cone()

```
void ABIP() free_cone (
    ABIPCone * k )
```

Definition at line 148 of file [util.c](#).

#### 4.161.1.3 free\_data()

```
void ABIP() free_data (
    ABIPData * d )
```

Definition at line 160 of file [util.c](#).

#### 4.161.1.4 free\_info()

```
void ABIP() free_info (
    ABIPInfo * info )
```

Definition at line 142 of file [util.c](#).

#### 4.161.1.5 free\_sol()

```
void ABIP() free_sol (
    ABIPSolution * sol )
```

Definition at line 182 of file [util.c](#).

#### 4.161.1.6 print\_array()

```
void ABIP() print_array (
    const abip_float * arr,
    abip_int n,
    const char * name )
```

Definition at line 119 of file [util.c](#).

#### 4.161.1.7 print\_data()

```
void ABIP() print_data (
    const ABIPData * d )
```

Definition at line 100 of file [util.c](#).

#### 4.161.1.8 print\_work()

```
void ABIP() print_work (
    const ABIPWork * w )
```

Definition at line 80 of file [util.c](#).

#### 4.161.1.9 set\_default\_settings()

```
void ABIP() set_default_settings (
    ABIPData * d )
```

Default parameter settings.

Definition at line 203 of file [util.c](#).

#### 4.161.1.10 str\_toc()

```
abip_float ABIP() str_toc (
    char * str,
    ABIP(timer) * t )
```

Definition at line 74 of file [util.c](#).

#### 4.161.1.11 tic()

```
void ABIP() tic (
    ABIP(timer) * t )
```

Definition at line 48 of file [util.c](#).

**4.161.1.12 toc()**

```
abip_float ABIP() toc (
    ABIP(timer) * t )
```

Definition at line 68 of file [util.c](#).

**4.161.1.13 tocq()**

```
abip_float ABIP() tocq (
    ABIP(timer) * t )
```

Definition at line 50 of file [util.c](#).

**4.162 util.h**

[Go to the documentation of this file.](#)

```
00001 #ifndef UTIL_H_GUARD
00002 #define UTIL_H_GUARD
00003
00004 #ifdef __cplusplus
00005 extern "C" {
00006 #endif
00007
00008 #include "abip.h"
00009 #include <stdlib.h>
00010 #include <stdio.h>
00011
00012 #if (defined NOTIMER)
00013 typedef void *ABIP(timer);
00014
00015 #elif (defined _WIN32 || defined _WIN64 || defined _WINDLL)
00016
00017 #include <windows.h>
00018 typedef struct ABIP(timer)
00019 {
00020     LARGE_INTEGER tic;
00021     LARGE_INTEGER toc;
00022     LARGE_INTEGER freq;
00023 } ABIP(timer);
00024
00025 #elif (defined __APPLE__)
00026
00027 #include <mach/mach_time.h>
00028 typedef struct ABIP(timer)
00029 {
00030     uint64_t tic;
00031     uint64_t toc;
00032     mach_timebase_info_data_t tinfo;
00033 } ABIP(timer);
00034
00035 #else
00036
00037 #include <time.h>
00038 typedef struct ABIP(timer)
00039 {
00040     struct timespec tic;
00041     struct timespec toc;
00042 } ABIP(timer);
00043
00044 #endif
00045
00046 #if EXTRA_VERBOSE > 1
00047 extern ABIP(timer) global_timer;
00048 #endif
00049
00050 void ABIP(tic)(ABIP(timer) *t);
00051 abip_float ABIP(toc)(ABIP(timer) *t);
00052 abip_float ABIP(str_toc)(char *str, ABIP(timer) *t);
```

```

00053 abip_float ABIP(tocq) (ABIP(timer) *t);
00054
00055 void ABIP(print_data) (const ABIPData *d);
00056 void ABIP(print_work) (const ABIPWork *w);
00057 void ABIP(print_array) (const abip_float *arr, abip_int n, const char *name);
00058 void ABIP(set_default_settings) (ABIPData *d);
00059 void ABIP(free_info) (ABIPInfo *info);
00060 void ABIP(free_sol) (ABIPSolution *sol);
00061 void ABIP(free_data) (ABIPData *d);
00062 void ABIP(free_cone) (ABIPCone *k);
00063 #ifdef __cplusplus
00064 }
00065 #endif
00066 #endif

```

## 4.163 make\_abip\_qcp.m File Reference

### Functions

- else [error](#) ('Unsupported platform.\n')
- if (debug==0) [debugcommand](#)
- [fprintf](#) ("%s\n", [mexcommand](#))
- [eval](#) (replace([mexcommand](#), "Program Files (x86)", "Program Files (x86)"))
- [addpath](#) (pmex)
- if ([mex\\_type](#)=="ml") [fprintf](#) ("Successfull compiled mex function abip\_ml\n")
- [fprintf](#) ("Usage:[sol,info] = abip\_ml(data,settings)\n")
- [fprintf](#) ("data struct contains X,y,lambda\n")
- [fprintf](#) ("data struct contains A,Q,c,rl,ru,lb,ub and L(optional)\n")
- [fprintf](#) ("data struct contains A,b,c\n")
- [fprintf](#) ("data struct contains A,Q,b,c\n")

### Variables

- [mex\\_type](#) = 'qcp'
- [debug](#) = 0
- [platform](#) = convertCharsToStrings(computer('arch'))
- [link](#) = ''
- [MKLROOT](#) = getenv('MKLROOT')
- [mkl\\_lib\\_path](#) = fullfile([MKLROOT](#), 'lib')
- [lib\\_path](#) = join(['-L', [mkl\\_lib\\_path](#)])
- If use [MKL](#)
- If use we suggest you set the environmental variables by [oneapi](#)
- If use we suggest you set the environmental variables by it is typically placed at opt intel [oneapi](#) setvars sh [alternatively](#)
- If use we suggest you set the environmental variables by it is typically placed at opt intel [oneapi](#) setvars sh you can set the [lib\\_path](#) to your [MKL](#) path by your [self](#)
- If use we suggest you set the environmental variables by it is typically placed at opt intel [oneapi](#) setvars sh you can set the [lib\\_path](#) to your [MKL](#) path by your For [example](#)
- If use we suggest you set the environmental variables by it is typically placed at opt intel [oneapi](#) setvars sh you can set the [lib\\_path](#) to your [MKL](#) path by your For in [linux](#)
- If use we suggest you set the environmental variables by it is typically placed at opt intel [oneapi](#) setvars sh you can set the [lib\\_path](#) to your [MKL](#) path by your For in you may find it at opt intel [oneapi](#) mkl lib [intel64](#)
- end [mexfname](#) = "abip\_" + [mex\\_type](#)
- [psrc](#) = fullfile('.', 'source')
- [pmex](#) = fullfile('.', 'mex')
- [mex\\_file](#) = fullfile([pmex](#), ['abip\_' [mex\\_type](#) '\_mex.c'])
- [src\\_files](#) = dir( [[psrc](#) '/\*.c'] )

- `srclist = []`
- `for i`
- `end src = fullfile(psrc, srclist)`
- `pcs = fullfile('.', 'csparse', 'Source')`
- `cs_files = dir( [pcs '/*.c'] )`
- `cslist = []`
- `end cs = fullfile(pcs, cslist)`
- `pdl = fullfile('.', 'qdldl', 'src')`
- `ldl_files = dir( [pdl '/*.c'] )`
- `ldllist = []`
- `end ldl = fullfile(pdl, ldllist)`
- `pamd = fullfile('.', 'amd')`
- `amd_files = dir( [pamd '/*.c'] )`
- `amdlist = []`
- `end amd = fullfile(pamd, amdlist)`
- `pinc = "include"`
- `cs_include = fullfile("csparse", "Include")`
- `ldl_include = fullfile("qdldl", "include")`
- `mkl_include = fullfile(mkl_path, "include")`
- `amd_include = "amd"`
- `inc = [pinc, mkl_include, cs_include, ldl_include, amd_include]`
- `else debugcommand = "-g "`
- `end mexcommand = "mex " + debugcommand + " -output " + join([mexfname, lib_path, src, inc, link])`

## 4.163.1 Function Documentation

### 4.163.1.1 `addpath()`

```
addpath (
    pmex )
```

### 4.163.1.2 `error()`

```
else error (
    'Unsupported platform.\n' )
```

### 4.163.1.3 `eval()`

```
eval (
    replace(mexcommand, "Program Files (x86)", "'Program Files (x86)'" )
```



**4.163.1.4 fprintf()** [1/6]

```
fprintf (
    "%s\n" ,
    mexcommand )
```

**4.163.1.5 fprintf()** [2/6]

```
fprintf (
    "data struct contains A,
    b ,
    c\n" )
```

**4.163.1.6 fprintf()** [3/6]

```
fprintf (
    "data struct contains A,
    Q ,
    b ,
    c\n" )
```

**4.163.1.7 fprintf()** [4/6]

```
fprintf (
    "data struct contains A,
    Q ,
    c ,
    rl ,
    ru ,
    lb ,
    ub and L(optional)\n" )
```

**4.163.1.8 fprintf()** [5/6]

```
fprintf (
    "data struct contains X,
    Y ,
    lambda\n" )
```

#### 4.163.1.9 fprintf() [6/6]

```
fprintf ( )
```

#### 4.163.1.10 if() [1/2]

```
if (
    debug == 0 )
```

#### 4.163.1.11 if() [2/2]

```
end if (
    mex_type == "ml" )
```

### 4.163.2 Variable Documentation

#### 4.163.2.1 alternatively

If use we suggest you set the environmental variables by it is typically placed at `opt intel oneapi` setvars sh alternatively

Definition at line 12 of file [make\\_abip\\_qcp.m](#).

#### 4.163.2.2 amd

```
end amd = fullfile(pamd, amdlist)
```

Definition at line 72 of file [make\\_abip\\_qcp.m](#).

#### 4.163.2.3 amd\_files

```
amd_files = dir( [pamd '/*.c'] )
```

Definition at line 67 of file [make\\_abip\\_qcp.m](#).

#### 4.163.2.4 amd\_include

```
amd_include = "amd"
```

Definition at line 84 of file [make\\_abip\\_qcp.m](#).

#### 4.163.2.5 amdlist

```
amdlist = []
```

Definition at line 68 of file [make\\_abip\\_qcp.m](#).

#### 4.163.2.6 cs

```
end cs = fullfile(pcs, cslist)
```

Definition at line 54 of file [make\\_abip\\_qcp.m](#).

#### 4.163.2.7 cs\_files

```
cs_files = dir( [pcs '/*.c'] )
```

Definition at line 49 of file [make\\_abip\\_qcp.m](#).

#### 4.163.2.8 cs\_include

```
cs_include = fullfile("csparse", "Include")
```

Definition at line 78 of file [make\\_abip\\_qcp.m](#).

#### 4.163.2.9 cslist

```
cslist = []
```

Definition at line 50 of file [make\\_abip\\_qcp.m](#).

**4.163.2.10 debug**

```
debug = 0
```

Definition at line 2 of file [make\\_abip\\_qcp.m](#).

**4.163.2.11 debugcommand**

```
else debugcommand = "-g "
```

Definition at line 92 of file [make\\_abip\\_qcp.m](#).

**4.163.2.12 example**

If use we suggest you set the environmental variables by it is typically placed at opt intel oneapi setvars sh you can set the [lib\\_path](#) to your [MKL](#) path by your For example

Definition at line 13 of file [make\\_abip\\_qcp.m](#).

**4.163.2.13 i**

```
for i
```

**Initial value:**

```
= 1:length(src_files)
    srclist = [srclist,convertCharsToStrings(src_files(i).name)]
```

Definition at line 40 of file [make\\_abip\\_qcp.m](#).

**4.163.2.14 inc**

```
inc = [pinc,mkl\_include,cs\_include,ldl\_include, amd\_include]
```

Definition at line 86 of file [make\\_abip\\_qcp.m](#).

**4.163.2.15 intel64**

If use we suggest you set the environmental variables by it is typically placed at opt intel oneapi setvars sh you can set the [lib\\_path](#) to your [MKL](#) path by your For in you may find it at opt intel oneapi mkl lib intel64

Definition at line 13 of file [make\\_abip\\_qcp.m](#).

#### 4.163.2.16 ldl

```
end ldl = fullfile(pldl, ldllist)
```

Definition at line 63 of file [make\\_abip\\_qcp.m](#).

#### 4.163.2.17 ldl\_files

```
ldl_files = dir( [pldl '/*.c'] )
```

Definition at line 58 of file [make\\_abip\\_qcp.m](#).

#### 4.163.2.18 ldl\_include

```
ldl_include = fullfile("qdldl", "include")
```

Definition at line 80 of file [make\\_abip\\_qcp.m](#).

#### 4.163.2.19 ldllist

```
ldllist = []
```

Definition at line 59 of file [make\\_abip\\_qcp.m](#).

#### 4.163.2.20 lib\_path

```
lib_path = join(['-L', mkl_lib_path])
```

Definition at line 8 of file [make\\_abip\\_qcp.m](#).

#### 4.163.2.21 link

```
link = ' '
```

Definition at line 5 of file [make\\_abip\\_qcp.m](#).

#### 4.163.2.22 linux

If use we suggest you set the environmental variables by it is typically placed at `opt intel oneapi` setvars sh you can set the `lib_path` to your `MKL` path by your `For` in linux

Definition at line 13 of file [make\\_abip\\_qcp.m](#).

#### 4.163.2.23 mex\_file

```
mex_file = fullfile(p mex, ['abip_' mex_type '_mex.c'])
```

Definition at line 36 of file [make\\_abip\\_qcp.m](#).

#### 4.163.2.24 mex\_type

```
mex_type = 'qcp'
```

Definition at line 1 of file [make\\_abip\\_qcp.m](#).

#### 4.163.2.25 mexcommand

```
end mexcommand = "mex " + debugcommand + " -output " + join([mexfname, lib_path, src, inc, link])
```

Definition at line 94 of file [make\\_abip\\_qcp.m](#).

#### 4.163.2.26 mexfname

```
end mexfname = "abip_" + mex_type
```

Definition at line 31 of file [make\\_abip\\_qcp.m](#).

#### 4.163.2.27 MKL

If use MKL

Definition at line 10 of file [make\\_abip\\_qcp.m](#).

#### 4.163.2.28 mkl\_include

```
mkl_include = fullfile(mkl_path, "include")
```

Definition at line 82 of file [make\\_abip\\_qcp.m](#).

#### 4.163.2.29 mkl\_lib\_path

```
mkl_lib_path = fullfile(MKLROOT, 'lib')
```

Definition at line 7 of file [make\\_abip\\_qcp.m](#).

#### 4.163.2.30 MKLROOT

```
MKLROOT = getenv('MKLROOT')
```

Definition at line 6 of file [make\\_abip\\_qcp.m](#).

#### 4.163.2.31 oneapi

If use we suggest you set the environmental variables by oneapi

Definition at line 10 of file [make\\_abip\\_qcp.m](#).

#### 4.163.2.32 pamd

```
pamd = fullfile('.', 'amd')
```

Definition at line 65 of file [make\\_abip\\_qcp.m](#).

#### 4.163.2.33 pcs

```
pcs = fullfile('.', 'csparse', 'Source')
```

Definition at line 47 of file [make\\_abip\\_qcp.m](#).

#### 4.163.2.34 pinc

```
pinc = "include"
```

Definition at line 76 of file [make\\_abip\\_qcp.m](#).

#### 4.163.2.35 platform

```
elseif platform = convertCharsToStrings(computer('arch'))
```

Definition at line 4 of file [make\\_abip\\_qcp.m](#).

#### 4.163.2.36 pldl

```
pldl = fullfile('.', 'qdldl', 'src')
```

Definition at line 56 of file [make\\_abip\\_qcp.m](#).

#### 4.163.2.37 pmex

```
pmex = fullfile('.', 'mex')
```

Definition at line 34 of file [make\\_abip\\_qcp.m](#).

#### 4.163.2.38 psrc

```
psrc = fullfile('.', 'source')
```

Definition at line 32 of file [make\\_abip\\_qcp.m](#).

#### 4.163.2.39 self

If use we suggest you set the environmental variables by it is typically placed at `opt intel oneapi` setvars sh you can set the `lib_path` to your `MKL` path by your self

Definition at line 12 of file [make\\_abip\\_qcp.m](#).



## 4.163.2.40 src

```
src = fullfile(psrc, srclist)
```

Definition at line 45 of file [make\\_abip\\_qcp.m](#).

## 4.163.2.41 src\_files

```
src_files = dir( [psrc '/*.c'] )
```

Definition at line 37 of file [make\\_abip\\_qcp.m](#).

## 4.163.2.42 srclist

```
srclist = []
```

Definition at line 38 of file [make\\_abip\\_qcp.m](#).

## 4.164 make\_abip\_qcp.m

[Go to the documentation of this file.](#)

```
00001 mex_type = 'qcp';
00002 debug = 0;
00003
00004 platform = convertCharsToStrings(computer('arch'));
00005 link = ' ';
00006 MKLROOT = getenv('MKLROOT');
00007 mkl_lib_path = fullfile(MKLROOT, 'lib');
00008 lib_path = join(['-L', mkl_lib_path]);
00009 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
00010 % If use MKL, we suggest you set the environmental variables by oneapi,
00011 % it is typically placed at /opt/intel/oneapi/setvars.sh
00012 % alternatively, you can set the lib_path to your MKL path by your self,
00013 % For example, in linux, you may find it at /opt/intel/oneapi/mkl/2021.2.0/lib/intel64,
00014 % then you may set therein.
00015 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
00016 if platform == "win64"
00017     fprintf('Linking MKL in Windows \n');
00018     link = [link, ' -lmkl_intel_lp64',...
00019           ' -lmkl_core', ' -lmkl_sequential '];
00020 elseif platform == "maci64"
00021     fprintf('Linking MKL in MacOS \n');
00022     link = [link, join(mkl_lib_path + ["/libmkl_intel_lp64.a", "/libmkl_core.a", "/libmkl_sequential.a
00023         "])];
00024 elseif platform == "glnxa64"
00025     fprintf('Linking MKL in Linux \n');
00026     link = [link, ' -lmkl_intel_ilp64',...
00027           ' -lmkl_core', ' -lmkl_sequential '];
00028 else
00029     error('Unsupported platform.\n');
00030 end
00031 mexfname = "abip_" + mex_type;
00032 psrc = fullfile('.', 'source');
00033
00034 pmex = fullfile('.', 'mex');
00035
00036 mex_file = fullfile(pmex, [mexfname mex_type '_mex.c']);
00037 src_files = dir( [psrc '/*.c'] );
00038 srclist = [];
00039
```

```

00040 for i = 1:length(src_files)
00041     srclist = [srclist,convertCharsToStrings(src_files(i).name)];
00042 end
00043
00044
00045 src = fullfile(psrc, srclist);
00046
00047 pcs = fullfile('.', 'csparse', 'Source');
00048
00049 cs_files = dir( [pcs '/*.c'] );
00050 cslist = [];
00051 for i = 1:length(cs_files)
00052     cslist = [cslist,convertCharsToStrings(cs_files(i).name)];
00053 end
00054 cs = fullfile(pcs, cslist);
00055
00056 pldl = fullfile('.', 'qdldl', 'src');
00057
00058 ldl_files = dir( [pldl '/*.c'] );
00059 ldllist = [];
00060 for i = 1:length(ldl_files)
00061     ldllist = [ldllist,convertCharsToStrings(ldl_files(i).name)];
00062 end
00063 ldl = fullfile(pldl, ldllist);
00064
00065 pamd = fullfile('.', 'amd');
00066
00067 amd_files = dir( [pamd '/*.c'] );
00068 amdlist = [];
00069 for i = 1:length(amd_files)
00070     amdlist = [amdlist,convertCharsToStrings(amd_files(i).name)];
00071 end
00072 amd = fullfile(pamd, amdlist);
00073
00074 src = [src,cs,ldl,mex_file,amd];
00075
00076 pinc = "include";
00077
00078 cs_include = fullfile("csparse", "Include");
00079
00080 ldl_include = fullfile("qdldl", "include");
00081
00082 mkl_include = fullfile(mkl_path, "include");
00083
00084 amd_include = "amd";
00085
00086 inc = [pinc,mkl_include,cs_include,ldl_include, amd_include];
00087 inc = join("-I" + inc);
00088
00089 if(debug == 0)
00090     debugcommand = "-O";
00091 else
00092     debugcommand = "-g ";
00093 end
00094 mexcommand = "mex " + debugcommand + " -output " + join([mexfname, lib_path, src, inc, link]);
00095 fprintf("%s\n",mexcommand);
00096 eval(replace(mexcommand, "Program Files (x86)", "'Program Files (x86)')");
00097
00098 addpath(pmex);
00099
00100 if(mex_type == "ml")
00101     fprintf("Successfull compiled mex function abip_ml\n");
00102     fprintf("Usage:[sol,info] = abip_ml(data,settings)\n ");
00103     fprintf("data struct contains X,y,lambda\n");
00104 end
00105 if(mex_type == "qp")
00106     fprintf("Successfull compiled mex function abip_qp\n");
00107     fprintf("Usage:[sol,info] = abip_qp(data,settings) \n ");
00108     fprintf("data struct contains A,Q,c,rl,ru,lb,ub and L(optional)\n");
00109 end
00110 if(mex_type == "socp")
00111     fprintf("Successfull compiled mex function abip_socp\n");
00112     fprintf("Usage:[sol,info] = abip_socp(data,cone,settings)\n ");
00113     fprintf("data struct contains A,b,c\n");
00114 end
00115 if(mex_type == "qcp")
00116     fprintf("Successfull compiled mex function abip_qcp\n");
00117     fprintf("Usage:[sol,info] = abip_qcp(data,cone,settings)\n ");
00118     fprintf("data struct contains A,Q,b,c\n");
00119 end

```

## 4.165 mex/abip\_ml\_mex.c File Reference

```
#include "abip.h"
#include "linsys.h"
#include "matrix.h"
#include "mex.h"
#include "cones.h"
```

### Functions

- void [mexFunction](#) (int nlhs, mxArray \*plhs[], int nrhs, const mxArray \*prhs[])

### 4.165.1 Function Documentation

#### 4.165.1.1 mexFunction()

```
void mexFunction (
    int nlhs,
    mxArray * plhs[],
    int nrhs,
    const mxArray * prhs[] )
```

Definition at line 93 of file [abip\\_ml\\_mex.c](#).

## 4.166 abip\_ml\_mex.c

[Go to the documentation of this file.](#)

```
00001 #include "abip.h"
00002 #include "linsys.h"
00003 #include "matrix.h"
00004 #include "mex.h"
00005 #include "cones.h"
00006
00007
00008 static void free_mex(ABIPData *d, ABIPConc *k) {
00009
00010     /* if (k) {
00011         abip_free(k);
00012     }*/
00013     if (d) {
00014
00015         if (d->stgs) {
00016             abip_free(d->stgs);
00017         }
00018
00019         if (d->b) {
00020             abip_free(d->b);
00021         }
00022         if (d->c) {
00023             abip_free(d->c);
00024         }
00025
00026         if (d->A) {
00027
00028             if (d->A->p) {
00029                 abip_free(d->A->p);
00030             }
00031         }
00032     }
00033 }
```

```

00031         if (d->A->i) {
00032             abip_free(d->A->i);
00033         }
00034         if (d->A->x) {
00035             abip_free(d->A->x);
00036         }
00037         abip_free(d->A);
00038     }
00039     abip_free(d);
00040 }
00041 }
00042
00043
00044 /* this memory must be freed */
00045 static abip_int *cast_to_abip_int_arr(mwIndex *arr, abip_int len) {
00046     abip_int i;
00047     abip_int *arr_out = (abip_int *)abip_malloc(sizeof(abip_int) * len);
00048     for (i = 0; i < len; i++) {
00049         arr_out[i] = (abip_int)arr[i];
00050     }
00051     return arr_out;
00052 }
00053
00054
00055 /* this memory must be freed */
00056 static abip_float *cast_to_abip_float_arr(double *arr, abip_int len) {
00057     abip_int i;
00058     abip_float *arr_out = (abip_float *)abip_malloc(sizeof(abip_float) * len);
00059     for (i = 0; i < len; i++) {
00060         arr_out[i] = (abip_float)arr[i];
00061     }
00062     return arr_out;
00063 }
00064
00065 static double *cast_to_double_arr(abip_float *arr, abip_int len) {
00066     abip_int i;
00067     double *arr_out = (double *)abip_malloc(sizeof(double) * len);
00068     for (i = 0; i < len; i++) {
00069         arr_out[i] = (double)arr[i];
00070     }
00071     return arr_out;
00072 }
00073
00074 static void set_output_field(mxArray **pout, abip_float *out, abip_int len) {
00075     *pout = mxCreateDoubleMatrix(0, 0, mxREAL);
00076     #if SFLOAT > 0
00077         mxSetPr(*pout, cast_to_double_arr(out, len));
00078         abip_free(out);
00079     #else
00080         mxSetPr(*pout, out);
00081     #endif
00082     mxSetM(*pout, len);
00083     mxSetN(*pout, 1);
00084 }
00085
00086
00087
00088
00089 /*the input of matlab is data, cone, settings, output is sol, info
00090 data is struct containing X,y,lambda;
00091 matlab usage: [sol,info] = abip(data,settings)
00092 */
00093 void mexFunction(int nlhs, mxArray* plhs[], int nrhs, const mxArray* prhs[])
00094 {
00095
00096     const mxArray *data;
00097     const mxArray *A_mex;
00098     const mxArray *b_mex;
00099     const mxArray *lambda_mex;
00100
00101
00102     const mxArray *settings;
00103     mxArray *tmp;
00104
00105     const mwSize one[1] = {1};
00106     const int num_info_fields = 14;
00107     const char *info_fields[] = {"ipm_iter", "admm_iter", "status", "pobj", "dobj", "res_pri",
"res_dual",
00108                                     "gap", "status_val", "setup_time", "solve_time", "runtime",
"lin_sys_time_per_iter",
00109                                     "avg_cg_iters"};
00110
00111     const int svm_num_sol_fields = 3;
00112     const char *svm_sol_fields[] = {"w", "b", "xi"};
00113
00114     const int lasso_num_sol_fields = 1;
00115     const char *lasso_sol_fields[] = {"x"};

```

```

00116
00117
00118     /* get data */
00119     ABIPData* d = (ABIPData*)abip_malloc(sizeof(ABIPData));
00120     data = prhs[0];
00121     A_mex = (mxArray *)mxGetField(data, 0, "X");
00122     if (A_mex == ABIP_NULL) {
00123         abip_free(d);
00124         mexErrMsgTxt("ABIPData struct must contain a 'X' entry.");
00125     }
00126     if (!mxIsSparse(A_mex)) {
00127         abip_free(d);
00128         mexErrMsgTxt("Input matrix X must be in sparse format (pass in sparse(X))");
00129     }
00130     b_mex = (mxArray *)mxGetField(data, 0, "y");
00131     if (b_mex == ABIP_NULL) {
00132         abip_free(d);
00133         mexErrMsgTxt("ABIPData struct must contain a 'y' entry.");
00134     }
00135     if (mxIsSparse(b_mex)) {
00136         abip_free(d);
00137         mexErrMsgTxt("Input vector y must be in dense format (pass in full(y))");
00138     }
00139     lambda_mex = (mxArray *)mxGetField(data, 0, "lambda");
00140     if (lambda_mex == ABIP_NULL) {
00141         abip_free(d);
00142         mexErrMsgTxt("ABIPData struct must contain a 'lambda' entry.");
00143     }
00144
00145
00146     d->lambda = (abip_float)*mxGetPr(lambda_mex);
00147     d->m = mxGetM(A_mex);
00148     d->n = mxGetN(A_mex);
00149     ABIPMatrix *A = (ABIPMatrix *)abip_malloc(sizeof(ABIPMatrix));
00150     A->m = d->m;
00151     A->n = d->n;
00152
00153     d->b = cast_to_abip_float_arr(mxGetPr(b_mex), d->m);
00154
00155     A->p = cast_to_abip_int_arr(mxGetJc(A_mex), A->n + 1);
00156     A->i = cast_to_abip_int_arr(mxGetIr(A_mex), A->p[A->n]);
00157     A->x = cast_to_abip_float_arr(mxGetPr(A_mex), A->p[A->n]);
00158
00159     d->A = A;
00160     d->c = ABIP_NULL;
00161
00162     /* get settings */
00163     settings = prhs[1];
00164     d->stgs = (ABIPSettings*)abip_malloc(sizeof(ABIPSettings));
00165     ABIP(set_default_settings)(d);
00166
00167     tmp = mxGetField(settings, 0, "alpha");
00168     if (tmp != ABIP_NULL) {
00169         d->stgs->alpha = (abip_float)*mxGetPr(tmp);
00170     }
00171
00172     tmp = mxGetField(settings, 0, "cg_rate");
00173     if (tmp != ABIP_NULL) {
00174         d->stgs->cg_rate = (abip_float)*mxGetPr(tmp);
00175     }
00176
00177     tmp = mxGetField(settings, 0, "eps");
00178     if (tmp != ABIP_NULL) {
00179         d->stgs->eps = (abip_float)*mxGetPr(tmp);
00180         d->stgs->eps_p = d->stgs->eps;
00181         d->stgs->eps_d = d->stgs->eps;
00182         d->stgs->eps_g = d->stgs->eps;
00183         d->stgs->eps_inf = d->stgs->eps;
00184         d->stgs->eps_unb = d->stgs->eps;
00185     }
00186     tmp = mxGetField(settings, 0, "eps_p");
00187     if (tmp != ABIP_NULL) {
00188         d->stgs->eps_p = (abip_float)*mxGetPr(tmp);
00189     }
00190     tmp = mxGetField(settings, 0, "eps_d");
00191     if (tmp != ABIP_NULL) {
00192         d->stgs->eps_d = (abip_float)*mxGetPr(tmp);
00193     }
00194     tmp = mxGetField(settings, 0, "eps_g");
00195     if (tmp != ABIP_NULL) {
00196         d->stgs->eps_g = (abip_float)*mxGetPr(tmp);
00197     }
00198     tmp = mxGetField(settings, 0, "eps_inf");
00199     if (tmp != ABIP_NULL) {
00200         d->stgs->eps_inf = (abip_float)*mxGetPr(tmp);
00201     }
00202     tmp = mxGetField(settings, 0, "eps_unb");
00203     if (tmp != ABIP_NULL) {

```

```

00203         d->stgs->eps_unb = (abip_float)*mxGetPr(tmp);
00204     }
00205
00206     tmp = mxGetField(settings, 0, "max_admm_iters");
00207     if (tmp != ABIP_NULL) {
00208         d->stgs->max_admm_iters = (abip_int)*mxGetPr(tmp);
00209     }
00210
00211     tmp = mxGetField(settings, 0, "max_ipm_iters");
00212     if (tmp != ABIP_NULL) {
00213         d->stgs->max_ipm_iters = (abip_int)*mxGetPr(tmp);
00214     }
00215
00216     tmp = mxGetField(settings, 0, "normalize");
00217     if (tmp != ABIP_NULL) {
00218         d->stgs->normalize = (abip_int)*mxGetPr(tmp);
00219     }
00220
00221     tmp = mxGetField(settings, 0, "rho_y");
00222     if (tmp != ABIP_NULL) {
00223         d->stgs->rho_y = (abip_float)*mxGetPr(tmp);
00224     }
00225
00226     tmp = mxGetField(settings, 0, "rho_x");
00227     if (tmp != ABIP_NULL) {
00228         d->stgs->rho_x = (abip_float)*mxGetPr(tmp);
00229     }
00230
00231     tmp = mxGetField(settings, 0, "rho_tau");
00232     if (tmp != ABIP_NULL) {
00233         d->stgs->rho_tau = (abip_float)*mxGetPr(tmp);
00234     }
00235
00236     tmp = mxGetField(settings, 0, "scale");
00237     if (tmp != ABIP_NULL) {
00238         d->stgs->scale = (abip_int)*mxGetPr(tmp);
00239     }
00240
00241     tmp = mxGetField(settings, 0, "scale_bc");
00242     if (tmp != ABIP_NULL) {
00243         d->stgs->scale_bc = (abip_int)*mxGetPr(tmp);
00244     }
00245
00246     tmp = mxGetField(settings, 0, "scale_E");
00247     if (tmp != ABIP_NULL) {
00248         d->stgs->scale_E = (abip_int)*mxGetPr(tmp);
00249     }
00250
00251     tmp = mxGetField(settings, 0, "use_indirect");
00252     if (tmp != ABIP_NULL) {
00253         d->stgs->use_indirect = (abip_int)*mxGetPr(tmp);
00254     }
00255
00256     tmp = mxGetField(settings, 0, "verbose");
00257     if (tmp != ABIP_NULL) {
00258         d->stgs->verbose = (abip_int)*mxGetPr(tmp);
00259     }
00260
00261     tmp = mxGetField(settings, 0, "linsys_solver");
00262     if (tmp != ABIP_NULL) {
00263         d->stgs->linsys_solver = (abip_int)*mxGetPr(tmp);
00264     }
00265
00266     tmp = mxGetField(settings, 0, "prob_type");
00267     if (tmp != ABIP_NULL) {
00268         d->stgs->prob_type = (abip_int)*mxGetPr(tmp);
00269         if (d->stgs->prob_type != LASSO && d->stgs->prob_type != SVM && d->stgs->prob_type != SVMQP) {
00270             mexErrMsgTxt("Invalid problem type");
00271         }
00272     }
00273     else{
00274         mexErrMsgTxt("Please input the machine learning problem type");
00275     }
00276
00277     tmp = mxGetField(settings, 0, "inner_check_period");
00278     if (tmp != ABIP_NULL) {
00279         d->stgs->inner_check_period = (abip_int)*mxGetPr(tmp);
00280     }
00281
00282     tmp = mxGetField(settings, 0, "outer_check_period");
00283     if (tmp != ABIP_NULL) {
00284         d->stgs->outer_check_period = (abip_int)*mxGetPr(tmp);
00285     }
00286
00287     tmp = mxGetField(settings, 0, "err_dif");
00288     if (tmp != ABIP_NULL) {
00289         d->stgs->err_dif = (abip_float)*mxGetPr(tmp);

```

```

00290     }
00291
00292     tmp = mxGetField(settings, 0, "time_limit");
00293     if (tmp != ABIP_NULL) {
00294         d->stgs->time_limit = (abip_float)*mxGetPr(tmp);
00295     }
00296     tmp = mxGetField(settings, 0, "psi");
00297     if (tmp != ABIP_NULL) {
00298         d->stgs->psi = (abip_float)*mxGetPr(tmp);
00299     }
00300
00301     tmp = mxGetField(settings, 0, "origin_scaling");
00302     if (tmp != ABIP_NULL) {
00303         d->stgs->origin_scaling = (abip_int)*mxGetPr(tmp);
00304     }
00305
00306     tmp = mxGetField(settings, 0, "ruiz_scaling");
00307     if (tmp != ABIP_NULL) {
00308         d->stgs->ruiz_scaling = (abip_int)*mxGetPr(tmp);
00309     }
00310
00311     tmp = mxGetField(settings, 0, "pc_scaling");
00312     if (tmp != ABIP_NULL) {
00313         d->stgs->pc_scaling = (abip_int)*mxGetPr(tmp);
00314     }
00315
00316     ABIPCone* k = (ABIPCone*)abip_malloc(sizeof(ABIPCone));
00317
00318     k->q = ABIP_NULL;
00319     k->qsize = 0;
00320
00321     k->rqsize = 1;
00322     k->rq = (abip_int *)abip_malloc(sizeof(abip_int));
00323
00324     k->f = 0;
00325     k->z = 0;
00326
00327     if(d->stgs->prob_type == LASSO) {
00328
00329         k->rq[0] = 2 + d->m;
00330         k->l = 2 * d->n;
00331     }
00332
00333     else if(d->stgs->prob_type == SVM) {
00334
00335         k->rq[0] = 2 + d->n;
00336         k->l = 2 + 2 * d->m + 2 * d->n;
00337     }
00338     else if(d->stgs->prob_type == SVMQP){
00339         k->rqsize = 0;
00340         k->rq = ABIP_NULL;
00341         k->f = d->n + 1;
00342         k->l = 2 * d->m;
00343     }
00344     else{
00345         mexErrMsgTxt("This type of machine learning problem is not supported yet");
00346     }
00347
00348     ABIPSolution* sol = (ABIPSolution*)abip_malloc(sizeof(ABIPSolution));
00349     ABIPInfo* info = (ABIPInfo*)abip_malloc(sizeof(ABIPInfo));
00350
00351     sol->x = ABIP_NULL;
00352     sol->y = ABIP_NULL;
00353     sol->s = ABIP_NULL;
00354
00355
00356
00357     abip_int status = abip(d,sol,info,k);
00358
00359     /* output sol */
00360
00361     //SVM
00362     if(d->stgs->prob_type == 2 || d->stgs->prob_type == 4){
00363
00364         plhs[0] = mxCreateStructArray(1, one, svm_num_sol_fields, svm_sol_fields);
00365
00366         set_output_field(&tmp, sol->x, d->n);
00367         mxSetField(plhs[0], 0, "w", tmp);
00368
00369         set_output_field(&tmp, sol->y, 1);
00370         mxSetField(plhs[0], 0, "b", tmp);
00371
00372         set_output_field(&tmp, sol->s, d->m);
00373         mxSetField(plhs[0], 0, "xi", tmp);
00374     }
00375     //LASSO
00376     else{

```

```

00377     plhs[0] = mxCreateStructArray(1, one, lasso_num_sol_fields, lasso_sol_fields);
00378
00379     set_output_field(&tmp, sol->x, d->n);
00380     mxSetField(plhs[0], 0, "x", tmp);
00381 }
00382
00383 /* output info */
00384 plhs[1] = mxCreateStructArray(1, one, num_info_fields, info_fields);
00385
00386 /* if you add/remove fields here update the info_fields above */
00387 mxSetField(plhs[1], 0, "status", mxCreateString(info->status));
00388
00389 tmp = mxCreateDoubleMatrix(1, 1, mxREAL);
00390 mxSetField(plhs[1], 0, "ipm_iter", tmp);
00391 *mxGetPr(tmp) = (abip_float)info->ipm_iter;
00392
00393 tmp = mxCreateDoubleMatrix(1, 1, mxREAL);
00394 mxSetField(plhs[1], 0, "admm_iter", tmp);
00395 *mxGetPr(tmp) = (abip_float)info->admm_iter;
00396
00397 tmp = mxCreateDoubleMatrix(1, 1, mxREAL);
00398 mxSetField(plhs[1], 0, "status_val", tmp);
00399 *mxGetPr(tmp) = (abip_float)info->status_val;
00400
00401 tmp = mxCreateDoubleMatrix(1, 1, mxREAL);
00402 mxSetField(plhs[1], 0, "pobj", tmp);
00403 *mxGetPr(tmp) = info->pobj;
00404
00405 tmp = mxCreateDoubleMatrix(1, 1, mxREAL);
00406 mxSetField(plhs[1], 0, "dobj", tmp);
00407 *mxGetPr(tmp) = info->dobj;
00408
00409 tmp = mxCreateDoubleMatrix(1, 1, mxREAL);
00410 mxSetField(plhs[1], 0, "res_pri", tmp);
00411 *mxGetPr(tmp) = info->res_pri;
00412
00413 tmp = mxCreateDoubleMatrix(1, 1, mxREAL);
00414 mxSetField(plhs[1], 0, "res_dual", tmp);
00415 *mxGetPr(tmp) = info->res_dual;
00416
00417 tmp = mxCreateDoubleMatrix(1, 1, mxREAL);
00418 mxSetField(plhs[1], 0, "gap", tmp);
00419 *mxGetPr(tmp) = info->rel_gap;
00420
00421
00422 /*return value in secs */
00423 tmp = mxCreateDoubleMatrix(1, 1, mxREAL);
00424 mxSetField(plhs[1], 0, "setup_time", tmp);
00425 *mxGetPr(tmp) = info->setup_time / 1e3;
00426
00427 /*return value in secs */
00428 tmp = mxCreateDoubleMatrix(1, 1, mxREAL);
00429 mxSetField(plhs[1], 0, "solve_time", tmp);
00430 *mxGetPr(tmp) = info->solve_time / 1e3;
00431
00432 /*return value in secs */
00433 tmp = mxCreateDoubleMatrix(1, 1, mxREAL);
00434 mxSetField(plhs[1], 0, "runtime", tmp);
00435 *mxGetPr(tmp) = (info->solve_time + info->setup_time) / 1e3;
00436
00437 /*return value in secs */
00438 tmp = mxCreateDoubleMatrix(1, 1, mxREAL);
00439 mxSetField(plhs[1], 0, "lin_sys_time_per_iter", tmp);
00440 *mxGetPr(tmp) = info->avg_linsys_time / 1e3;
00441
00442 //average cg iters per admm iter
00443 tmp = mxCreateDoubleMatrix(1, 1, mxREAL);
00444 mxSetField(plhs[1], 0, "avg_cg_iters", tmp);
00445 *mxGetPr(tmp) = info->avg_cg_iters;
00446
00447 free_mex(d, k);
00448 return;
00449 }

```

## 4.167 mex/abip\_qcp\_mex.c File Reference

```

#include "abip.h"
#include "linsys.h"
#include "matrix.h"
#include "mex.h"

```



```
#include "cones.h"
```

## Functions

- void [mexFunction](#) (int nlhs, mxArray \*plhs[], int nrhs, const mxArray \*prhs[])

### 4.167.1 Function Documentation

#### 4.167.1.1 mexFunction()

```
void mexFunction (
    int nlhs,
    mxArray * plhs[],
    int nrhs,
    const mxArray * prhs[] )
```

Definition at line 109 of file [abip\\_qcp\\_mex.c](#).

## 4.168 abip\_qcp\_mex.c

[Go to the documentation of this file.](#)

```
00001 #include "abip.h"
00002 #include "linsys.h"
00003 #include "matrix.h"
00004 #include "mex.h"
00005 #include "cones.h"
00006
00007 /*
00008 This is mex file for creating matlab routine:
00009
00010 function [results, info] = abip_qp(data, cones, params)
00011
00012     min      1/2x'Qx + c'x
00013 subject to  Ax = b
00014             x in K
00015
00016 where A \in R^m*n, b \in R^m, c \in R^n
00017
00018 data:
00019 A:      m*n matrix used as above
00020 b:      m dimensional column vector used as above
00021 c:      n dimensional column vector used as above
00022
00023 K:      covex cone, now SOC, RSOC, free cone, zero cone and positive orthant are supported
00024 */
00025
00026 static void free_mex(ABIPData *d, ABIPCone *k) {
00027
00028
00029     if (k) {
00030
00031         if (k->q) {
00032             abip_free(k->q);
00033         }
00034         if (k->rq) {
00035             abip_free(k->rq);
00036         }
00037         abip_free(k);
00038     }
00039     if (d) {
00040
```

```

00041         if (d->A) {
00042
00043             abip_free(d->A->x);
00044
00045             abip_free(d->A->i);
00046             abip_free(d->A->p);
00047
00048         }
00049
00050         if (d->b) {
00051             abip_free(d->b);
00052         }
00053         if (d->c) {
00054             abip_free(d->c);
00055         }
00056
00057         abip_free(d);
00058     }
00059 }
00060
00061
00062
00063 #if !(DLONG > 0)
00064 /* this memory must be freed */
00065 static abip_int *cast_to_abip_int_arr(mwIndex *arr, abip_int len) {
00066     abip_int i;
00067     abip_int *arr_out = (abip_int *)abip_malloc(sizeof(abip_int) * len);
00068     for (i = 0; i < len; i++) {
00069         arr_out[i] = (abip_int)arr[i];
00070     }
00071     return arr_out;
00072 }
00073 #endif
00074
00075
00076 /* this memory must be freed */
00077 static abip_float *cast_to_abip_float_arr(double *arr, abip_int len) {
00078     abip_int i;
00079     abip_float *arr_out = (abip_float *)abip_malloc(sizeof(abip_float) * len);
00080     for (i = 0; i < len; i++) {
00081         arr_out[i] = (abip_float)arr[i];
00082     }
00083     return arr_out;
00084 }
00085
00086 static double *cast_to_double_arr(abip_float *arr, abip_int len) {
00087     abip_int i;
00088     double *arr_out = (double *)abip_malloc(sizeof(double) * len);
00089     for (i = 0; i < len; i++) {
00090         arr_out[i] = (double)arr[i];
00091     }
00092     return arr_out;
00093 }
00094
00095
00096 static void set_output_field(mxArray **pout, abip_float *out, abip_int len) {
00097     *pout = mxCreateDoubleMatrix(0, 0, mxREAL);
00098     #if SFLOAT > 0
00099         mxSetPr(*pout, cast_to_double_arr(out, len));
00100         abip_free(out);
00101     #else
00102         mxSetPr(*pout, out);
00103     #endif
00104     mxSetM(*pout, len);
00105     mxSetN(*pout, 1);
00106 }
00107
00108
00109 void mexFunction(int nlhs, mxArray* plhs[], int nrhs, const mxArray* prhs[])
00110 {
00111
00112     const mxArray *data;
00113     const mxArray *A_mex;
00114     const mxArray *Q_mex;
00115     const mxArray *b_mex;
00116     const mxArray *c_mex;
00117     const mxArray *cone;
00118     const mxArray *settings;
00119     const mxArray *kq;
00120     const mxArray *krq;
00121     const mxArray *kf;
00122     const mxArray *kz;
00123     const mxArray *kl;
00124     const double *q_mex;
00125     const double *rq_mex;
00126     const size_t *q_dims;
00127     const size_t *rq_dims;

```

```

00128     abip_int ns;
00129
00130     mxArray *tmp;
00131
00132     const mwSize one[1] = {1};
00133     const int num_info_fields = 14;
00134     const char *info_fields[] = {"ipm_iter", "admm_iter", "status", "pobj", "dobj", "res_pri",
"res_dual",
00135     "gap", "status_val", "setup_time", "solve_time", "runtime",
"lin_sys_time_per_iter",
00136     "avg_cg_iters"};
00137
00138     const int num_sol_fields = 3;
00139     const char *sol_fields[] = {"x", "y", "s"};
00140
00141
00142     /* get data*/
00143     ABIPData* d = (ABIPData*)abip_malloc(sizeof(ABIPData));
00144     data = prhs[0];
00145
00146
00147     b_mex = (mxArray *)mxGetField(data, 0, "b");
00148     if (b_mex == ABIP_NULL) {
00149         d->b = ABIP_NULL;
00150         printf("ABIPData doesn't contain 'b'\n");
00151     }
00152     else if (mxIsSparse(b_mex)) {
00153         abip_free(d);
00154         mexErrMsgTxt("Input vector b must be in dense format (pass in full(b))");
00155     }
00156     else{
00157         d->b = cast_to_abip_float_arr(mxGetPr(b_mex), MAX(mxGetM(b_mex), mxGetN(b_mex)));
00158     }
00159
00160
00161
00162     c_mex = (mxArray *)mxGetField(data, 0, "c");
00163     if (c_mex == ABIP_NULL) {
00164         abip_free(d);
00165         mexErrMsgTxt("ABIPData struct must contain a 'c' entry.");
00166     }
00167     else if (mxIsSparse(c_mex)) {
00168         abip_free(d);
00169         mexErrMsgTxt("Input vector c must be in dense format (pass in full(c))");
00170     }
00171     else{
00172         d->c = cast_to_abip_float_arr(mxGetPr(c_mex), MAX(mxGetM(c_mex), mxGetN(c_mex)));
00173     }
00174
00175
00176
00177     A_mex = (mxArray *)mxGetField(data, 0, "A");
00178     if (A_mex == ABIP_NULL) {
00179         d->A = ABIP_NULL;
00180         d->m = 0;
00181         d->n = MAX(mxGetM(c_mex), mxGetN(c_mex));
00182
00183         printf("ABIPData doesn't contain 'A'\n");
00184     }
00185     else if (!mxIsSparse(A_mex)) {
00186         abip_free(d);
00187         mexErrMsgTxt("Input matrix A must be in sparse format (pass in sparse(A))");
00188     }
00189     else{
00190         d->A = (ABIPMatrix*)abip_malloc(sizeof(ABIPMatrix));
00191         ABIPMatrix *A = d->A;
00192
00193         A->m = mxGetM(A_mex);
00194         A->n = mxGetN(A_mex);
00195
00196         d->m = A->m;
00197         d->n = A->n;
00198
00199
00200         A->p = cast_to_abip_int_arr(mxGetJc(A_mex), A->n + 1);
00201         A->i = cast_to_abip_int_arr(mxGetIr(A_mex), A->p[A->n]);
00202
00203         A->x = cast_to_abip_float_arr(mxGetPr(A_mex), A->p[A->n]);
00204     }
00205
00206
00207     Q_mex = (mxArray *)mxGetField(data, 0, "Q");
00208     if (Q_mex == ABIP_NULL) {
00209         d->Q = ABIP_NULL;
00210         printf("ABIPData doesn't contain 'Q'\n");
00211     }
00212     else if (!mxIsSparse(Q_mex)) {

```

```

00213         abip_free(d);
00214         mexErrMsgTxt("Input matrix Q must be in sparse format (pass in sparse(Q))");
00215     }
00216     else{
00217         d->Q = (ABIPMatrix*)abip_malloc(sizeof(ABIPMatrix));
00218         ABIPMatrix *Q = d->Q;
00219
00220         Q->m = mxGetM(Q_mex);
00221         Q->n = mxGetN(Q_mex);
00222
00223
00224         Q->p = cast_to_abip_int_arr(mxGetJc(Q_mex), Q->n + 1);
00225         Q->i = cast_to_abip_int_arr(mxGetIr(Q_mex), Q->p[Q->n]);
00226         Q->x = cast_to_abip_float_arr(mxGetPr(Q_mex), Q->p[Q->n]);
00227
00228     }
00229
00230     /*get cone*/
00231     cone = prhs[1];
00232
00233     ABIPCone *k = (ABIPCone*)abip_malloc(sizeof(ABIPCone));
00234
00235     kq = mxGetField(cone, 0, "q");
00236     if (kq && !mxIsEmpty(kq)) {
00237         q_mex = mxGetPr(kq);
00238         ns = (abip_int)mxGetNumberOfDimensions(kq);
00239         q_dims = mxGetDimensions(kq);
00240         k->qsize = (abip_int)q_dims[0];
00241         if (ns > 1 && q_dims[0] == 1) {
00242             k->qsize = (abip_int)q_dims[1];
00243         }
00244         k->q = (abip_int *)mxMalloc(sizeof(abip_int) * k->qsize);
00245         for (abip_int i = 0; i < k->qsize; i++) {
00246             k->q[i] = (abip_int)q_mex[i];
00247         }
00248     } else {
00249         k->qsize = 0;
00250         k->q = ABIP_NULL;
00251     }
00252
00253     krq = mxGetField(cone, 0, "rq");
00254     if (krq && !mxIsEmpty(krq)) {
00255         rq_mex = mxGetPr(krq);
00256         ns = (abip_int)mxGetNumberOfDimensions(krq);
00257         rq_dims = mxGetDimensions(krq);
00258         k->rsize = (abip_int)rq_dims[0];
00259         if (ns > 1 && rq_dims[0] == 1) {
00260             k->rsize = (abip_int)rq_dims[1];
00261         }
00262         k->r = (abip_int *)mxMalloc(sizeof(abip_int) * k->rsize);
00263         for (abip_int i = 0; i < k->rsize; i++) {
00264             k->r[i] = (abip_int)rq_mex[i];
00265         }
00266     } else {
00267         k->rsize = 0;
00268         k->r = ABIP_NULL;
00269     }
00270
00271     kf = mxGetField(cone, 0, "f");
00272     if (kf && !mxIsEmpty(kf)) {
00273         k->f = (abip_int)*mxGetPr(kf);
00274     } else {
00275         k->f = 0;
00276     }
00277
00278     kz = mxGetField(cone, 0, "z");
00279     if (kz && !mxIsEmpty(kz)) {
00280         k->z = (abip_int)*mxGetPr(kz);
00281     } else {
00282         k->z = 0;
00283     }
00284
00285     kl = mxGetField(cone, 0, "l");
00286     if (kl && !mxIsEmpty(kl)) {
00287         k->l = (abip_int)*mxGetPr(kl);
00288     } else {
00289         k->l = 0;
00290     }
00291
00292     /*get settings*/
00293     settings = prhs[2];
00294     d->stgs = (ABIPSettings*)abip_malloc(sizeof(ABIPSettings));
00295     ABIP(set_default_settings)(d);
00296
00297     tmp = mxGetField(settings, 0, "alpha");
00298     if (tmp != ABIP_NULL) {
00299         d->stgs->alpha = (abip_float)*mxGetPr(tmp);

```

```

00300     }
00301
00302     tmp = mxGetField(settings, 0, "cg_rate");
00303     if (tmp != ABIP_NULL) {
00304         d->stgs->cg_rate = (abip_float)*mxGetPr(tmp);
00305     }
00306
00307     tmp = mxGetField(settings, 0, "eps");
00308     if (tmp != ABIP_NULL) {
00309         d->stgs->eps = (abip_float)*mxGetPr(tmp);
00310         d->stgs->eps_p = d->stgs->eps;
00311         d->stgs->eps_d = d->stgs->eps;
00312         d->stgs->eps_g = d->stgs->eps;
00313         d->stgs->eps_inf = d->stgs->eps;
00314         d->stgs->eps_unb = d->stgs->eps;
00315     }
00316     tmp = mxGetField(settings, 0, "eps_p");
00317     if (tmp != ABIP_NULL) {
00318         d->stgs->eps_p = (abip_float)*mxGetPr(tmp);
00319     }
00320     tmp = mxGetField(settings, 0, "eps_d");
00321     if (tmp != ABIP_NULL) {
00322         d->stgs->eps_d = (abip_float)*mxGetPr(tmp);
00323     }
00324     tmp = mxGetField(settings, 0, "eps_g");
00325     if (tmp != ABIP_NULL) {
00326         d->stgs->eps_g = (abip_float)*mxGetPr(tmp);
00327     }
00328     tmp = mxGetField(settings, 0, "eps_inf");
00329     if (tmp != ABIP_NULL) {
00330         d->stgs->eps_inf = (abip_float)*mxGetPr(tmp);
00331         tmp = mxGetField(settings, 0, "eps_unb");
00332         if (tmp != ABIP_NULL) {
00333             d->stgs->eps_unb = (abip_float)*mxGetPr(tmp);
00334         }
00335     }
00336     tmp = mxGetField(settings, 0, "max_admm_iters");
00337     if (tmp != ABIP_NULL) {
00338         d->stgs->max_admm_iters = (abip_int)*mxGetPr(tmp);
00339     }
00340
00341     tmp = mxGetField(settings, 0, "max_ipm_iters");
00342     if (tmp != ABIP_NULL) {
00343         d->stgs->max_ipm_iters = (abip_int)*mxGetPr(tmp);
00344     }
00345
00346     tmp = mxGetField(settings, 0, "normalize");
00347     if (tmp != ABIP_NULL) {
00348         d->stgs->normalize = (abip_int)*mxGetPr(tmp);
00349     }
00350
00351     tmp = mxGetField(settings, 0, "rho_y");
00352     if (tmp != ABIP_NULL) {
00353         d->stgs->rho_y = (abip_float)*mxGetPr(tmp);
00354     }
00355
00356     tmp = mxGetField(settings, 0, "rho_x");
00357     if (tmp != ABIP_NULL) {
00358         d->stgs->rho_x = (abip_float)*mxGetPr(tmp);
00359     }
00360
00361     tmp = mxGetField(settings, 0, "rho_tau");
00362     if (tmp != ABIP_NULL) {
00363         d->stgs->rho_tau = (abip_float)*mxGetPr(tmp);
00364     }
00365
00366     tmp = mxGetField(settings, 0, "scale");
00367     if (tmp != ABIP_NULL) {
00368         d->stgs->scale = (abip_int)*mxGetPr(tmp);
00369     }
00370
00371     tmp = mxGetField(settings, 0, "scale_bc");
00372     if (tmp != ABIP_NULL) {
00373         d->stgs->scale_bc = (abip_int)*mxGetPr(tmp);
00374     }
00375
00376     tmp = mxGetField(settings, 0, "scale_E");
00377     if (tmp != ABIP_NULL) {
00378         d->stgs->scale_E = (abip_int)*mxGetPr(tmp);
00379     }
00380
00381     tmp = mxGetField(settings, 0, "use_indirect");
00382     if (tmp != ABIP_NULL) {
00383         d->stgs->use_indirect = (abip_int)*mxGetPr(tmp);
00384     }
00385
00386     tmp = mxGetField(settings, 0, "verbose");

```

```

00387     if (tmp != ABIP_NULL) {
00388         d->stgs->verbose = (abip_int)*mxGetPr(tmp);
00389     }
00390
00391     tmp = mxGetField(settings, 0, "linsys_solver");
00392     if (tmp != ABIP_NULL) {
00393         d->stgs->linsys_solver = (abip_int)*mxGetPr(tmp);
00394     }
00395
00396     tmp = mxGetField(settings, 0, "inner_check_period");
00397     if (tmp != ABIP_NULL) {
00398         d->stgs->inner_check_period = (abip_int)*mxGetPr(tmp);
00399     }
00400
00401     tmp = mxGetField(settings, 0, "outer_check_period");
00402     if (tmp != ABIP_NULL) {
00403         d->stgs->outer_check_period = (abip_int)*mxGetPr(tmp);
00404     }
00405
00406     tmp = mxGetField(settings, 0, "err_dif");
00407     if (tmp != ABIP_NULL) {
00408         d->stgs->err_dif = (abip_float)*mxGetPr(tmp);
00409     }
00410
00411     tmp = mxGetField(settings, 0, "time_limit");
00412     if (tmp != ABIP_NULL) {
00413         d->stgs->time_limit = (abip_float)*mxGetPr(tmp);
00414     }
00415
00416     tmp = mxGetField(settings, 0, "psi");
00417     if (tmp != ABIP_NULL) {
00418         d->stgs->psi = (abip_float)*mxGetPr(tmp);
00419     }
00420
00421     tmp = mxGetField(settings, 0, "origin_scaling");
00422     if (tmp != ABIP_NULL) {
00423         d->stgs->origin_scaling = (abip_int)*mxGetPr(tmp);
00424     }
00425
00426     tmp = mxGetField(settings, 0, "ruiz_scaling");
00427     if (tmp != ABIP_NULL) {
00428         d->stgs->ruiz_scaling = (abip_int)*mxGetPr(tmp);
00429     }
00430
00431     tmp = mxGetField(settings, 0, "pc_scaling");
00432     if (tmp != ABIP_NULL) {
00433         d->stgs->pc_scaling = (abip_int)*mxGetPr(tmp);
00434     }
00435
00436     d->stgs->prob_type = QCP; //QCP
00437
00438     ABIPSolution* sol = (ABIPSolution*)abip_malloc(sizeof(ABIPSolution));
00439     ABIPInfo* info = (ABIPInfo*)abip_malloc(sizeof(ABIPInfo));
00440
00441     sol->x = ABIP_NULL;
00442     sol->y = ABIP_NULL;
00443     sol->s = ABIP_NULL;
00444
00445     abip_int status = abip(d,sol,info,k);
00446
00447     /* output sol */
00448     plhs[0] = mxCreateStructArray(1, one, num_sol_fields, sol_fields);
00449
00450     set_output_field(&tmp, sol->x, d->n);
00451     mxSetField(plhs[0], 0, "x", tmp);
00452
00453     set_output_field(&tmp, sol->y, d->m);
00454     mxSetField(plhs[0], 0, "y", tmp);
00455
00456     set_output_field(&tmp, sol->s, d->n);
00457     mxSetField(plhs[0], 0, "s", tmp);
00458
00459     /* output info */
00460     plhs[1] = mxCreateStructArray(1, one, num_info_fields, info_fields);
00461
00462     /* if you add/remove fields here update the info_fields above */
00463     mxSetField(plhs[1], 0, "status", mxCreateString(info->status));
00464
00465     tmp = mxCreateDoubleMatrix(1, 1, mxREAL);
00466     mxSetField(plhs[1], 0, "ipm_iter", tmp);
00467     *mxGetPr(tmp) = (abip_float)info->ipm_iter;
00468
00469     tmp = mxCreateDoubleMatrix(1, 1, mxREAL);
00470     mxSetField(plhs[1], 0, "admm_iter", tmp);
00471     *mxGetPr(tmp) = (abip_float)info->admm_iter;
00472
00473     tmp = mxCreateDoubleMatrix(1, 1, mxREAL);

```

```

00474     mxSetField(plhs[1], 0, "status_val", tmp);
00475     *mxGetPr(tmp) = (abip_float)info->status_val;
00476
00477     tmp = mxCreateDoubleMatrix(1, 1, mxREAL);
00478     mxSetField(plhs[1], 0, "pobj", tmp);
00479     *mxGetPr(tmp) = info->pobj;
00480
00481     tmp = mxCreateDoubleMatrix(1, 1, mxREAL);
00482     mxSetField(plhs[1], 0, "dobj", tmp);
00483     *mxGetPr(tmp) = info->dobj;
00484
00485     tmp = mxCreateDoubleMatrix(1, 1, mxREAL);
00486     mxSetField(plhs[1], 0, "res_pri", tmp);
00487     *mxGetPr(tmp) = info->res_pri;
00488
00489     tmp = mxCreateDoubleMatrix(1, 1, mxREAL);
00490     mxSetField(plhs[1], 0, "res_dual", tmp);
00491     *mxGetPr(tmp) = info->res_dual;
00492
00493     tmp = mxCreateDoubleMatrix(1, 1, mxREAL);
00494     mxSetField(plhs[1], 0, "gap", tmp);
00495     *mxGetPr(tmp) = info->rel_gap;
00496
00497     /*return value in secs */
00498     tmp = mxCreateDoubleMatrix(1, 1, mxREAL);
00499     mxSetField(plhs[1], 0, "setup_time", tmp);
00500     *mxGetPr(tmp) = info->setup_time / 1e3;
00501
00502     /*return value in secs */
00503     tmp = mxCreateDoubleMatrix(1, 1, mxREAL);
00504     mxSetField(plhs[1], 0, "solve_time", tmp);
00505     *mxGetPr(tmp) = info->solve_time / 1e3;
00506
00507     /*return value in secs */
00508     tmp = mxCreateDoubleMatrix(1, 1, mxREAL);
00509     mxSetField(plhs[1], 0, "runtime", tmp);
00510     *mxGetPr(tmp) = (info->setup_time + info->solve_time) / 1e3;
00511
00512     /*return value in secs */
00513     tmp = mxCreateDoubleMatrix(1, 1, mxREAL);
00514     mxSetField(plhs[1], 0, "lin_sys_time_per_iter", tmp);
00515     *mxGetPr(tmp) = info->avg_linsys_time / 1e3;
00516
00517     //average cg iters per admm iter
00518     tmp = mxCreateDoubleMatrix(1, 1, mxREAL);
00519     mxSetField(plhs[1], 0, "avg_cg_iters", tmp);
00520     *mxGetPr(tmp) = info->avg_cg_iters;
00521
00522
00523     free_mex(d, k);
00524     return;
00525 }

```

## 4.169 qdldl/include/qdldl.h File Reference

```
#include "qdldl_types.h"
```

### Functions

- [QDLDL\\_int QDLDL\\_etree](#) (const [QDLDL\\_int](#) n, const [QDLDL\\_int](#) \*Ap, const [QDLDL\\_int](#) \*Ai, [QDLDL\\_int](#) \*work, [QDLDL\\_int](#) \*Lnz, [QDLDL\\_int](#) \*etree)
- [QDLDL\\_int QDLDL\\_factor](#) (const [QDLDL\\_int](#) n, const [QDLDL\\_int](#) \*Ap, const [QDLDL\\_int](#) \*Ai, const [QDLDL\\_float](#) \*Ax, [QDLDL\\_int](#) \*Lp, [QDLDL\\_int](#) \*Li, [QDLDL\\_float](#) \*Lx, [QDLDL\\_float](#) \*D, [QDLDL\\_float](#) \*Dinv, const [QDLDL\\_int](#) \*Lnz, const [QDLDL\\_int](#) \*etree, [QDLDL\\_bool](#) \*bwork, [QDLDL\\_int](#) \*iwork, [QDLDL\\_float](#) \*fwork)
- void [QDLDL\\_solve](#) (const [QDLDL\\_int](#) n, const [QDLDL\\_int](#) \*Lp, const [QDLDL\\_int](#) \*Li, const [QDLDL\\_float](#) \*Lx, const [QDLDL\\_float](#) \*Dinv, [QDLDL\\_float](#) \*x)
- void [QDLDL\\_Lsolve](#) (const [QDLDL\\_int](#) n, const [QDLDL\\_int](#) \*Lp, const [QDLDL\\_int](#) \*Li, const [QDLDL\\_float](#) \*Lx, [QDLDL\\_float](#) \*x)
- void [QDLDL\\_Ltsolve](#) (const [QDLDL\\_int](#) n, const [QDLDL\\_int](#) \*Lp, const [QDLDL\\_int](#) \*Li, const [QDLDL\\_float](#) \*Lx, [QDLDL\\_float](#) \*x)

## 4.169.1 Function Documentation

### 4.169.1.1 QDLDL\_etree()

```
QDLDL_int QDLDL_etree (
    const QDLDL_int n,
    const QDLDL_int * Ap,
    const QDLDL_int * Ai,
    QDLDL_int * work,
    QDLDL_int * Lnz,
    QDLDL_int * etree )
```

Compute the elimination tree for a quasidefinite matrix in compressed sparse column form, where the input matrix is assumed to contain data for the upper triangular part of A only, and there are no duplicate indices.

Returns an elimination tree for the factorization  $A = LDL^T$  and a count of the nonzeros in each column of L that are strictly below the diagonal.

Does not use MALLOC. It is assumed that the arrays work, Lnz, and etree will be allocated with a number of elements equal to n.

The data in (n,Ap,Ai) are from a square matrix A in CSC format, and should include the upper triangular part of A only.

This function is only intended for factorisation of QD matrices specified by their upper triangular part. An error is returned if any column has data below the diagonal or is completely empty.

For matrices with a non-empty column but a zero on the corresponding diagonal, this function will *not* return an error, as it may still be possible to factor such a matrix in LDL form. No promises are made in this case though...

#### Parameters

<i>n</i>	number of columns in CSC matrix A (assumed square)
<i>Ap</i>	column pointers (size n+1) for columns of A
<i>Ai</i>	row indices of A. Has Ap[n] elements
<i>work</i>	work vector (size n) (no meaning on return)
<i>Lnz</i>	count of nonzeros in each column of L (size n) below diagonal
<i>etree</i>	elimination tree (size n)

#### Returns

total sum of Lnz (i.e. total nonzeros in L below diagonal). Returns -1 if the input is not triu or has an empty column. Returns -2 if the return value overflows QDLDL\_int.

Definition at line 11 of file [qddl.c](#).



## 4.169.1.2 QDLDL\_factor()

```

QDLDL_int QDLDL_factor (
    const QDLDL_int n,
    const QDLDL_int * Ap,
    const QDLDL_int * Ai,
    const QDLDL_float * Ax,
    QDLDL_int * Lp,
    QDLDL_int * Li,
    QDLDL_float * Lx,
    QDLDL_float * D,
    QDLDL_float * Dinv,
    const QDLDL_int * Lnz,
    const QDLDL_int * etree,
    QDLDL_bool * bwork,
    QDLDL_int * iwork,
    QDLDL_float * fwork )

```

Compute an LDL decomposition for a quasidefinite matrix in compressed sparse column form, where the input matrix is assumed to contain data for the upper triangular part of A only, and there are no duplicate indices.

Returns factors L, D and  $D_{inv} = 1./D$ .

Does not use MALLOC. It is assumed that L will be a compressed sparse column matrix with data (n,Lp,Li,Lx) with sufficient space allocated, with a number of nonzeros equal to the count given as a return value by QDLDL\_etree

## Parameters

<i>n</i>	number of columns in L and A (both square)
<i>Ap</i>	column pointers (size n+1) for columns of A (not modified)
<i>Ai</i>	row indices of A. Has Ap[n] elements (not modified)
<i>Ax</i>	data of A. Has Ap[n] elements (not modified)
<i>Lp</i>	column pointers (size n+1) for columns of L
<i>Li</i>	row indices of L. Has Lp[n] elements
<i>Lx</i>	data of L. Has Lp[n] elements
<i>D</i>	vectorized factor D. Length is n
<i>Dinv</i>	reciprocal of D. Length is n
<i>Lnz</i>	count of nonzeros in each column of L below diagonal, as given by QDLDL_etree (not modified)
<i>etree</i>	elimination tree as as given by QDLDL_etree (not modified)
<i>bwork</i>	working array of bools. Length is n
<i>iwork</i>	working array of integers. Length is 3*n
<i>fwork</i>	working array of floats. Length is n

## Returns

Returns a count of the number of positive elements in D. Returns -1 and exits immediately if any element of D evaluates exactly to zero (matrix is not quasidefinite or otherwise LDL factorisable)

Definition at line 72 of file [qdldl.c](#).

#### 4.169.1.3 QDLDL\_Lsolve()

```
void QDLDL_Lsolve (
    const QDLDL_int n,
    const QDLDL_int * Lp,
    const QDLDL_int * Li,
    const QDLDL_float * Lx,
    QDLDL_float * x )
```

Solves  $(L+I)x = b$

It is assumed that L will be a compressed sparse column matrix with data (n,Lp,Li,Lx).

##### Parameters

<i>n</i>	number of columns in L
<i>Lp</i>	column pointers (size n+1) for columns of L
<i>Li</i>	row indices of L. Has Lp[n] elements
<i>Lx</i>	data of L. Has Lp[n] elements
<i>x</i>	initialized to b. Equal to x on return

Definition at line 236 of file [qddl.c](#).

#### 4.169.1.4 QDLDL\_Ltsolve()

```
void QDLDL_Ltsolve (
    const QDLDL_int n,
    const QDLDL_int * Lp,
    const QDLDL_int * Li,
    const QDLDL_float * Lx,
    QDLDL_float * x )
```

Solves  $(L+I)'x = b$

It is assumed that L will be a compressed sparse column matrix with data (n,Lp,Li,Lx).

##### Parameters

<i>n</i>	number of columns in L
<i>Lp</i>	column pointers (size n+1) for columns of L
<i>Li</i>	row indices of L. Has Lp[n] elements
<i>Lx</i>	data of L. Has Lp[n] elements
<i>x</i>	initialized to b. Equal to x on return

Definition at line 252 of file [qddl.c](#).

## 4.169.1.5 QDLDL\_solve()

```
void QDLDL_solve (
    const QDLDL_int n,
    const QDLDL_int * Lp,
    const QDLDL_int * Li,
    const QDLDL_float * Lx,
    const QDLDL_float * Dinv,
    QDLDL_float * x )
```

Solves  $LDL^T x = b$

It is assumed that L will be a compressed sparse column matrix with data (n,Lp,Li,Lx).

## Parameters

<i>n</i>	number of columns in L
<i>Lp</i>	column pointers (size n+1) for columns of L
<i>Li</i>	row indices of L. Has Lp[n] elements
<i>Lx</i>	data of L. Has Lp[n] elements
<i>Dinv</i>	reciprocal of D. Length is n
<i>x</i>	initialized to b. Equal to x on return

Definition at line 269 of file [qdldl.c](#).

## 4.170 qdldl.h

[Go to the documentation of this file.](#)

```
00001 #ifndef QDLDL_H
00002 #define QDLDL_H
00003
00004 // Include qdldl type options
00005 #include "qdldl_types.h"
00006
00007 # ifdef __cplusplus
00008 extern "C" {
00009 # endif // ifdef __cplusplus
00010
00046 QDLDL_int QDLDL_etree(const QDLDL_int n,
00047                      const QDLDL_int* Ap,
00048                      const QDLDL_int* Ai,
00049                      QDLDL_int* work,
00050                      QDLDL_int* Lnz,
00051                      QDLDL_int* etree);
00052
00053
00088 QDLDL_int QDLDL_factor(const QDLDL_int n,
00089                      const QDLDL_int* Ap,
00090                      const QDLDL_int* Ai,
00091                      const QDLDL_float* Ax,
00092                      QDLDL_int* Lp,
00093                      QDLDL_int* Li,
00094                      QDLDL_float* Lx,
00095                      QDLDL_float* D,
00096                      QDLDL_float* Dinv,
00097                      const QDLDL_int* Lnz,
00098                      const QDLDL_int* etree,
00099                      QDLDL_bool* bwork,
00100                      QDLDL_int* iwork,
00101                      QDLDL_float* fwork);
00102
00103
00118 void QDLDL_solve(const QDLDL_int n,
00119                  const QDLDL_int* Lp,
00120                  const QDLDL_int* Li,
```

```

00121             const QDLDL_float* Lx,
00122             const QDLDL_float* Dinvs,
00123             QDLDL_float* x);
00124
00125
00139 void QDLDL_Lsolve(const QDLDL_int    n,
00140                  const QDLDL_int*    Lp,
00141                  const QDLDL_int*    Li,
00142                  const QDLDL_float*  Lx,
00143                  QDLDL_float*  x);
00144
00145
00159 void QDLDL_Ltsolve(const QDLDL_int    n,
00160                   const QDLDL_int*    Lp,
00161                   const QDLDL_int*    Li,
00162                   const QDLDL_float*  Lx,
00163                   QDLDL_float*  x);
00164
00165 # ifdef __cplusplus
00166 }
00167 # endif // ifdef __cplusplus
00168
00169 #endif // ifndef QDLDL_H

```

## 4.171 qdldl/include/qdldl\_types.h File Reference

```

#include <limits.h>
#include "glbopts.h"

```

### Macros

- `#define QDLDL_INT_MAX INT_MAX`

### Typedefs

- typedef `abip_int` `QDLDL_int`
- typedef `abip_float` `QDLDL_float`
- typedef `abip_int` `QDLDL_bool`

## 4.171.1 Macro Definition Documentation

### 4.171.1.1 QDLDL\_INT\_MAX

```
#define QDLDL_INT_MAX INT_MAX
```

Definition at line 18 of file `qdldl_types.h`.

### 4.171.2 Typedef Documentation

#### 4.171.2.1 QDDL\_bool

```
typedef abip_int QDDL_bool
```

Definition at line 15 of file [qddl\\_types.h](#).

#### 4.171.2.2 QDDL\_float

```
typedef abip_float QDDL_float
```

Definition at line 14 of file [qddl\\_types.h](#).

#### 4.171.2.3 QDDL\_int

```
typedef abip_int QDDL_int
```

Definition at line 13 of file [qddl\\_types.h](#).

### 4.172 qddl\_types.h

[Go to the documentation of this file.](#)

```
00001 #ifndef QDDL_TYPES_H
00002 # define QDDL_TYPES_H
00003
00004 # ifdef __cplusplus
00005 extern "C" {
00006 # endif /* ifdef __cplusplus */
00007
00008 #include <limits.h> //for the QDDL_INT_TYPE_MAX
00009 #include "glbopts.h"
00010
00011 // QDDL integer and float types
00012
00013 typedef abip_int    QDDL_int; /* for indices */
00014 typedef abip_float  QDDL_float; /* for numerical values */
00015 typedef abip_int    QDDL_bool; /* for boolean values */
00016
00017 //Maximum value of the signed type QDDL_int.
00018 #define QDDL_INT_MAX INT_MAX
00019
00020 # ifdef __cplusplus
00021 }
00022 # endif /* ifdef __cplusplus */
00023
00024 #endif /* ifndef QDDL_TYPES_H */
```

### 4.173 qddl/src/qddl.c File Reference

```
#include "qddl.h"
```

## Macros

- `#define QDLDL_UNKNOWN (-1)`
- `#define QDLDL_USED (1)`
- `#define QDLDL_UNUSED (0)`

## Functions

- `QDLDL_int QDLDL_etree` (const `QDLDL_int` n, const `QDLDL_int` \*Ap, const `QDLDL_int` \*Ai, `QDLDL_int` \*work, `QDLDL_int` \*Lnz, `QDLDL_int` \*etree)
- `QDLDL_int QDLDL_factor` (const `QDLDL_int` n, const `QDLDL_int` \*Ap, const `QDLDL_int` \*Ai, const `QDLDL_float` \*Ax, `QDLDL_int` \*Lp, `QDLDL_int` \*Li, `QDLDL_float` \*Lx, `QDLDL_float` \*D, `QDLDL_float` \*Dinv, const `QDLDL_int` \*Lnz, const `QDLDL_int` \*etree, `QDLDL_bool` \*bwork, `QDLDL_int` \*iwork, `QDLDL_float` \*fwork)
- `void QDLDL_Lsolve` (const `QDLDL_int` n, const `QDLDL_int` \*Lp, const `QDLDL_int` \*Li, const `QDLDL_float` \*Lx, `QDLDL_float` \*x)
- `void QDLDL_Ltsolve` (const `QDLDL_int` n, const `QDLDL_int` \*Lp, const `QDLDL_int` \*Li, const `QDLDL_float` \*Lx, `QDLDL_float` \*x)
- `void QDLDL_solve` (const `QDLDL_int` n, const `QDLDL_int` \*Lp, const `QDLDL_int` \*Li, const `QDLDL_float` \*Lx, const `QDLDL_float` \*Dinv, `QDLDL_float` \*x)

### 4.173.1 Macro Definition Documentation

#### 4.173.1.1 QDLDL\_UNKNOWN

```
#define QDLDL_UNKNOWN (-1)
```

Definition at line 3 of file [qdlldl.c](#).

#### 4.173.1.2 QDLDL\_UNUSED

```
#define QDLDL_UNUSED (0)
```

Definition at line 5 of file [qdlldl.c](#).

#### 4.173.1.3 QDLDL\_USED

```
#define QDLDL_USED (1)
```

Definition at line 4 of file [qdlldl.c](#).

## 4.173.2 Function Documentation

### 4.173.2.1 QDLDL\_etree()

```
QDLDL_int QDLDL_etree (
    const QDLDL_int n,
    const QDLDL_int * Ap,
    const QDLDL_int * Ai,
    QDLDL_int * work,
    QDLDL_int * Lnz,
    QDLDL_int * etree )
```

Compute the elimination tree for a quasidefinite matrix in compressed sparse column form, where the input matrix is assumed to contain data for the upper triangular part of A only, and there are no duplicate indices.

Returns an elimination tree for the factorization  $A = LDL^T$  and a count of the nonzeros in each column of L that are strictly below the diagonal.

Does not use MALLOC. It is assumed that the arrays work, Lnz, and etree will be allocated with a number of elements equal to n.

The data in (n,Ap,Ai) are from a square matrix A in CSC format, and should include the upper triangular part of A only.

This function is only intended for factorisation of QD matrices specified by their upper triangular part. An error is returned if any column has data below the diagonal or is completely empty.

For matrices with a non-empty column but a zero on the corresponding diagonal, this function will *not* return an error, as it may still be possible to factor such a matrix in LDL form. No promises are made in this case though...

#### Parameters

<i>n</i>	number of columns in CSC matrix A (assumed square)
<i>Ap</i>	column pointers (size n+1) for columns of A
<i>Ai</i>	row indices of A. Has Ap[n] elements
<i>work</i>	work vector (size n) (no meaning on return)
<i>Lnz</i>	count of nonzeros in each column of L (size n) below diagonal
<i>etree</i>	elimination tree (size n)

#### Returns

total sum of Lnz (i.e. total nonzeros in L below diagonal). Returns -1 if the input is not triu or has an empty column. Returns -2 if the return value overflows QDLDL\_int.

Definition at line 11 of file [qdldl.c](#).

#### 4.173.2.2 QDLDL\_factor()

```
QDLDL_int QDLDL_factor (
    const QDLDL_int n,
    const QDLDL_int * Ap,
    const QDLDL_int * Ai,
    const QDLDL_float * Ax,
    QDLDL_int * Lp,
    QDLDL_int * Li,
    QDLDL_float * Lx,
    QDLDL_float * D,
    QDLDL_float * Dinv,
    const QDLDL_int * Lnz,
    const QDLDL_int * etree,
    QDLDL_bool * bwork,
    QDLDL_int * iwork,
    QDLDL_float * fwork )
```

Compute an LDL decomposition for a quasidefinite matrix in compressed sparse column form, where the input matrix is assumed to contain data for the upper triangular part of A only, and there are no duplicate indices.

Returns factors L, D and  $D_{inv} = 1./D$ .

Does not use MALLOC. It is assumed that L will be a compressed sparse column matrix with data (n,Lp,Li,Lx) with sufficient space allocated, with a number of nonzeros equal to the count given as a return value by QDLDL\_etree

##### Parameters

<i>n</i>	number of columns in L and A (both square)
<i>Ap</i>	column pointers (size n+1) for columns of A (not modified)
<i>Ai</i>	row indices of A. Has Ap[n] elements (not modified)
<i>Ax</i>	data of A. Has Ap[n] elements (not modified)
<i>Lp</i>	column pointers (size n+1) for columns of L
<i>Li</i>	row indices of L. Has Lp[n] elements
<i>Lx</i>	data of L. Has Lp[n] elements
<i>D</i>	vectorized factor D. Length is n
<i>Dinv</i>	reciprocal of D. Length is n
<i>Lnz</i>	count of nonzeros in each column of L below diagonal, as given by QDLDL_etree (not modified)
<i>etree</i>	elimination tree as as given by QDLDL_etree (not modified)
<i>bwork</i>	working array of bools. Length is n
<i>iwork</i>	working array of integers. Length is 3*n
<i>fwork</i>	working array of floats. Length is n

##### Returns

Returns a count of the number of positive elements in D. Returns -1 and exits immediately if any element of D evaluates exactly to zero (matrix is not quasidefinite or otherwise LDL factorisable)

Definition at line 72 of file [qddl.c](#).



### 4.173.2.3 QDLDL\_Lsolve()

```
void QDLDL_Lsolve (
    const QDLDL_int n,
    const QDLDL_int * Lp,
    const QDLDL_int * Li,
    const QDLDL_float * Lx,
    QDLDL_float * x )
```

Solves  $(L+I)x = b$

It is assumed that L will be a compressed sparse column matrix with data (n,Lp,Li,Lx).

#### Parameters

<i>n</i>	number of columns in L
<i>Lp</i>	column pointers (size n+1) for columns of L
<i>Li</i>	row indices of L. Has Lp[n] elements
<i>Lx</i>	data of L. Has Lp[n] elements
<i>x</i>	initialized to b. Equal to x on return

Definition at line 236 of file [qdldl.c](#).

### 4.173.2.4 QDLDL\_Ltsolve()

```
void QDLDL_Ltsolve (
    const QDLDL_int n,
    const QDLDL_int * Lp,
    const QDLDL_int * Li,
    const QDLDL_float * Lx,
    QDLDL_float * x )
```

Solves  $(L+I)'x = b$

It is assumed that L will be a compressed sparse column matrix with data (n,Lp,Li,Lx).

#### Parameters

<i>n</i>	number of columns in L
<i>Lp</i>	column pointers (size n+1) for columns of L
<i>Li</i>	row indices of L. Has Lp[n] elements
<i>Lx</i>	data of L. Has Lp[n] elements
<i>x</i>	initialized to b. Equal to x on return

Definition at line 252 of file [qdldl.c](#).

#### 4.173.2.5 QDLDL\_solve()

```
void QDLDL_solve (
    const QDLDL_int n,
    const QDLDL_int * Lp,
    const QDLDL_int * Li,
    const QDLDL_float * Lx,
    const QDLDL_float * Dinv,
    QDLDL_float * x )
```

Solves  $LDL^T x = b$

It is assumed that L will be a compressed sparse column matrix with data (n,Lp,Li,Lx).

##### Parameters

<i>n</i>	number of columns in L
<i>Lp</i>	column pointers (size n+1) for columns of L
<i>Li</i>	row indices of L. Has Lp[n] elements
<i>Lx</i>	data of L. Has Lp[n] elements
<i>Dinv</i>	reciprocal of D. Length is n
<i>x</i>	initialized to b. Equal to x on return

Definition at line 269 of file [qddl.c](#).

## 4.174 qddl.c

[Go to the documentation of this file.](#)

```
00001 #include "qddl.h"
00002
00003 #define QDLDL_UNKNOWN (-1)
00004 #define QDLDL_USED (1)
00005 #define QDLDL_UNUSED (0)
00006
00007 /* Compute the elimination tree for a quasidefinite matrix
00008    in compressed sparse column form.
00009 */
00010
00011 QDLDL_int QDLDL_etree(const QDLDL_int n,
00012                      const QDLDL_int* Ap,
00013                      const QDLDL_int* Ai,
00014                      QDLDL_int* work,
00015                      QDLDL_int* Lnz,
00016                      QDLDL_int* etree){
00017
00018     QDLDL_int sumLnz;
00019     QDLDL_int i,j,p;
00020
00021
00022     for(i = 0; i < n; i++){
00023         // zero out Lnz and work. Set all etree values to unknown
00024         work[i] = 0;
00025         Lnz[i] = 0;
00026         etree[i] = QDLDL_UNKNOWN;
00027
00028         //Abort if A doesn't have at least one entry
00029         //one entry in every column
00030         if(Ap[i] == Ap[i+1]){
00031             return -1;
00032         }
00033     }
00034
00035     for(j = 0; j < n; j++){
00036         work[j] = j;
```

```

00038     for(p = Ap[j]; p < Ap[j+1]; p++){
00039         i = Ai[p];
00040         if(i > j){return -1;}; //abort if entries on lower triangle
00041         while(work[i] != j){
00042             if(etree[i] == QDLDL_UNKNOWN){
00043                 etree[i] = j;
00044             }
00045             Lnz[i]++;           //nonzeros in this column
00046             work[i] = j;
00047             i = etree[i];
00048         }
00049     }
00050 }
00051
00052 //compute the total nonzeros in L. This much
00053 //space is required to store Li and Lx. Return
00054 //error code -2 if the nonzero count will overflow
00055 //its unteger type.
00056 sumLnz = 0;
00057 for(i = 0; i < n; i++){
00058     if(sumLnz > QDLDL_INT_MAX - Lnz[i]){
00059         sumLnz = -2;
00060         break;
00061     }
00062     else{
00063         sumLnz += Lnz[i];
00064     }
00065 }
00066
00067 return sumLnz;
00068 }
00069
00070
00071
00072 QDLDL_int QDLDL_factor(const QDLDL_int    n,
00073                       const QDLDL_int*    Ap,
00074                       const QDLDL_int*    Ai,
00075                       const QDLDL_float*  Ax,
00076                       QDLDL_int*          Lp,
00077                       QDLDL_int*          Li,
00078                       QDLDL_float*        Lx,
00079                       QDLDL_float*        D,
00080                       QDLDL_float*        Dinv,
00081                       const QDLDL_int*    Lnz,
00082                       const QDLDL_int*    etree,
00083                       QDLDL_bool*         bwork,
00084                       QDLDL_int*          iwork,
00085                       QDLDL_float*        fwork){
00086
00087     QDLDL_int i,j,k,nnzY, bidx, cidx, nextIdx, nnzE, tmpIdx;
00088     QDLDL_int *yIdx, *elimBuffer, *LNextSpaceInCol;
00089     QDLDL_float *yVals;
00090     QDLDL_float yVals_cidx;
00091     QDLDL_bool *yMarkers;
00092     QDLDL_int positiveValuesInD = 0;
00093
00094     //partition working memory into pieces
00095     yMarkers      = bwork;
00096     yIdx          = iwork;
00097     elimBuffer    = iwork + n;
00098     LNextSpaceInCol = iwork + n*2;
00099     yVals         = fwork;
00100
00101
00102     Lp[0] = 0; //first column starts at index zero
00103
00104     for(i = 0; i < n; i++){
00105
00106         //compute L column indices
00107         Lp[i+1] = Lp[i] + Lnz[i]; //cumsum, total at the end
00108
00109         // set all Yidx to be 'unused' initially
00110         //in each column of L, the next available space
00111         //to start is just the first space in the column
00112         yMarkers[i] = QDLDL_UNUSED;
00113         yVals[i]    = 0.0;
00114         D[i]        = 0.0;
00115         LNextSpaceInCol[i] = Lp[i];
00116     }
00117
00118     // First element of the diagonal D.
00119     D[0] = Ax[0];
00120     if(D[0] == 0.0){return -1;}
00121     if(D[0] > 0.0){positiveValuesInD++;}
00122     Dinv[0] = 1/D[0];
00123
00124     //Start from 1 here. The upper LH corner is trivially 0

```

```

00125 //in L b/c we are only computing the subdiagonal elements
00126 for(k = 1; k < n; k++){
00127
00128     //NB : For each k, we compute a solution to
00129     //y = L(0:(k-1),0:k-1)\b, where b is the kth
00130     //column of A that sits above the diagonal.
00131     //The solution y is then the kth row of L,
00132     //with an implied '1' at the diagonal entry.
00133
00134     //number of nonzeros in this row of L
00135     nnzY = 0; //number of elements in this row
00136
00137     //This loop determines where nonzeros
00138     //will go in the kth row of L, but doesn't
00139     //compute the actual values
00140     tmpIdx = Ap[k+1];
00141
00142     for(i = Ap[k]; i < tmpIdx; i++){
00143
00144         bidx = Ai[i]; // we are working on this element of b
00145
00146         //Initialize D[k] as the element of this column
00147         //corresponding to the diagonal place. Don't use
00148         //this element as part of the elimination step
00149         //that computes the k^th row of L
00150         if(bidx == k){
00151             D[k] = Ax[i];
00152             continue;
00153         }
00154
00155         yVals[bidx] = Ax[i]; // initialise y(bidx) = b(bidx)
00156
00157         // use the forward elimination tree to figure
00158         // out which elements must be eliminated after
00159         // this element of b
00160         nextIdx = bidx;
00161
00162         if(yMarkers[nextIdx] == QDLDL_UNUSED){ //this y term not already visited
00163
00164             yMarkers[nextIdx] = QDLDL_USED; //I touched this one
00165             elimBuffer[0] = nextIdx; // It goes at the start of the current list
00166             nnzE = 1; //length of unvisited elimination path from here
00167
00168             nextIdx = etree[bidx];
00169
00170             while(nextIdx != QDLDL_UNKNOWN && nextIdx < k){
00171                 if(yMarkers[nextIdx] == QDLDL_USED) break;
00172
00173                 yMarkers[nextIdx] = QDLDL_USED; //I touched this one
00174                 elimBuffer[nnzE] = nextIdx; //It goes in the current list
00175                 nnzE++; //the list is one longer than before
00176                 nextIdx = etree[nextIdx]; //one step further along tree
00177
00178             } //end while
00179
00180             // now I put the buffered elimination list into
00181             // my current ordering in reverse order
00182             while(nnzE){
00183                 yIdx[nnzY++] = elimBuffer[--nnzE];
00184             } //end while
00185         } //end if
00186     } //end for i
00187
00188 //This for loop places nonzeros values in the k^th row
00189 for(i = (nnzY-1); i >=0; i--){
00190
00191     //which column are we working on?
00192     cidx = yIdx[i];
00193
00194
00195     // loop along the elements in this
00196     // column of L and subtract to solve to y
00197     tmpIdx = LNextSpaceInCol[cidx];
00198     yVals_cidx = yVals[cidx];
00199     for(j = Lp[cidx]; j < tmpIdx; j++){
00200         yVals[Li[j]] -= Lx[j]*yVals_cidx;
00201     }
00202
00203     //Now I have the cidx^th element of y = L\b.
00204     //so compute the corresponding element of
00205     //this row of L and put it into the right place
00206     Li[tmpIdx] = k;
00207     Lx[tmpIdx] = yVals_cidx *Dinv[cidx];
00208
00209     //D[k] -= yVals[cidx]*yVals[cidx]*Dinv[cidx];
00210     D[k] -= yVals_cidx*Lx[tmpIdx];
00211     LNextSpaceInCol[cidx]++;

```

```

00212
00213     //reset the yvalues and indices back to zero and QDLDL_UNUSED
00214     //once I'm done with them
00215     yVals[cidx] = 0.0;
00216     yMarkers[cidx] = QDLDL_UNUSED;
00217
00218     } //end for i
00219
00220     //Maintain a count of the positive entries
00221     //in D. If we hit a zero, we can't factor
00222     //this matrix, so abort
00223     if(D[k] == 0.0){return -1;}
00224     if(D[k] > 0.0){positiveValuesInD++;}
00225
00226     //compute the inverse of the diagonal
00227     Dinv[k]= 1/D[k];
00228
00229     } //end for k
00230
00231     return positiveValuesInD;
00232 }
00233 }
00234
00235 // Solves (L+I)x = b
00236 void QDLDL_Lsolve(const QDLDL_int n,
00237                  const QDLDL_int* Lp,
00238                  const QDLDL_int* Li,
00239                  const QDLDL_float* Lx,
00240                  QDLDL_float* x){
00241
00242     QDLDL_int i,j;
00243     for(i = 0; i < n; i++){
00244         QDLDL_float val = x[i];
00245         for(j = Lp[i]; j < Lp[i+1]; j++){
00246             x[Li[j]] -= Lx[j]*val;
00247         }
00248     }
00249 }
00250
00251 // Solves (L+I)'x = b
00252 void QDLDL_Ltsolve(const QDLDL_int n,
00253                  const QDLDL_int* Lp,
00254                  const QDLDL_int* Li,
00255                  const QDLDL_float* Lx,
00256                  QDLDL_float* x){
00257
00258     QDLDL_int i,j;
00259     for(i = n-1; i>=0; i--){
00260         QDLDL_float val = x[i];
00261         for(j = Lp[i]; j < Lp[i+1]; j++){
00262             val -= Lx[j]*x[Li[j]];
00263         }
00264         x[i] = val;
00265     }
00266 }
00267
00268 // Solves Ax = b where A has given LDL factors
00269 void QDLDL_solve(const QDLDL_int n,
00270                 const QDLDL_int* Lp,
00271                 const QDLDL_int* Li,
00272                 const QDLDL_float* Lx,
00273                 const QDLDL_float* Dinv,
00274                 QDLDL_float* x){
00275
00276     QDLDL_int i;
00277
00278     QDLDL_Lsolve(n,Lp,Li,Lx,x);
00279     for(i = 0; i < n; i++) x[i] *= Dinv[i];
00280     QDLDL_Ltsolve(n,Lp,Li,Lx,x);
00281 }

```

## 4.175 source/abip.c File Reference

```

#include "abip.h"
#include <stdio.h>
#include "cones.h"
#include "cs.h"
#include "ctrlc.h"
#include "glbopts.h"

```

```
#include "lasso_config.h"
#include "linalg.h"
#include "linsys.h"
#include "mkl.h"
#include "mkl_lapacke.h"
#include "qcp_config.h"
#include "svm_config.h"
#include "svm_qp_config.h"
#include "util.h"
```

## Macros

- `#define _CRT_SECURE_NO_WARNINGS`

## Functions

- `ABIP` (timer)
- `abip_int update_work` (const `ABIPData` \*d, `ABIPWork` \*w, const `ABIPSolution` \*sol, const `ABIPConc` \*c, `spe_problem` \*spe)
- `abip_float adjust_barrier` (`ABIPWork` \*w, `ABIPResiduals` \*r, `spe_problem` \*spe)
- `abip_int ABIP()` `solve` (`ABIPWork` \*w, const `ABIPData` \*d, `ABIPSolution` \*sol, `ABIPInfo` \*info, `ABIPConc` \*c, `spe_problem` \*s)
  - Main solve iteration of the solver.*
- void `ABIP()` `finish` (`ABIPWork` \*w, `spe_problem` \*spe)
  - Free the memory allocated for the solver.*
- `ABIPWork` \*`ABIP()` `init` (const `ABIPData` \*d, `ABIPInfo` \*info, `spe_problem` \*s, `ABIPConc` \*c)
  - Initialize and allocate memory for the solver.*
- `abip_int ABIP()` `init_problem` (`spe_problem` \*\*s, `ABIPData` \*d, `ABIPSettings` \*stgs, enum `problem_type` `special_problem`)
  - Specify the problem type and initialize the problem.*
- `abip_int abip` (const `ABIPData` \*d, `ABIPSolution` \*sol, `ABIPInfo` \*info, `ABIPConc` \*K)
  - the main function to call the abip conic solver*

## 4.175.1 Macro Definition Documentation

### 4.175.1.1 \_CRT\_SECURE\_NO\_WARNINGS

```
#define _CRT_SECURE_NO_WARNINGS
```

Definition at line 1 of file `abip.c`.

## 4.175.2 Function Documentation

#### 4.175.2.1 abip()

```
abip_int abip (
    const ABIPData * d,
    ABIPSolution * sol,
    ABIPInfo * info,
    ABIPCone * K )
```

the main function to call the abip conic solver

Definition at line 1335 of file [abip.c](#).

#### 4.175.2.2 ABIP()

```
ABIP (
    timer )
```

Definition at line 20 of file [abip.c](#).

#### 4.175.2.3 adjust\_barrier()

```
abip_float adjust_barrier (
    ABIPWork * w,
    ABIPResiduals * r,
    spe_problem * spe )
```

Definition at line 994 of file [abip.c](#).

#### 4.175.2.4 finish()

```
void ABIP() finish (
    ABIPWork * w,
    spe_problem * spe )
```

Free the memory allocated for the solver.

Definition at line 1254 of file [abip.c](#).

#### 4.175.2.5 init()

```
ABIPWork *ABIP() init (
    const ABIPData * d,
    ABIPInfo * info,
    spe_problem * s,
    ABIPCone * c )
```

Initialize and allocate memory for the solver.

Definition at line 1271 of file [abip.c](#).

#### 4.175.2.6 init\_problem()

```
abip_int ABIP() init_problem (
    spe_problem ** s,
    ABIPData * d,
    ABIPSettings * stgs,
    enum problem_type special_problem )
```

Specify the problem type and initialize the problem.

Definition at line 1316 of file [abip.c](#).

#### 4.175.2.7 solve()

```
abip_int ABIP() solve (
    ABIPWork * w,
    const ABIPData * d,
    ABIPSolution * sol,
    ABIPInfo * info,
    ABIPCone * c,
    spe_problem * s )
```

Main solve iteration of the solver.

Definition at line 1076 of file [abip.c](#).

#### 4.175.2.8 update\_work()

```
abip_int update_work (
    const ABIPData * d,
    ABIPWork * w,
    const ABIPSolution * sol,
    const ABIPCone * c,
    spe_problem * spe )
```

Definition at line 912 of file [abip.c](#).



## 4.176 abip.c

[Go to the documentation of this file.](#)

```

00001 #define _CRT_SECURE_NO_WARNINGS
00002 #include "abip.h"
00003
00004 #include <stdio.h>
00005
00006 #include "cones.h"
00007 #include "cs.h"
00008 #include "ctrlc.h"
00009 #include "glbopts.h"
00010 #include "lasso_config.h"
00011 #include "linalg.h"
00012 #include "linsys.h"
00013 #include "mkl.h"
00014 #include "mkl_lapacke.h"
00015 #include "qcp_config.h"
00016 #include "svm_config.h"
00017 #include "svm_qp_config.h"
00018 #include "util.h"
00019
00020 ABIP(timer) global_timer;
00021
00022 /* printing header */
00023 static const char *HEADER[] = {
00024     " ipm iter ", " admm iter ", "      mu ", " pri res ", " dua res ",
00025     " rel gap ", " pri obj ", " dua obj ", " kap/tau ", " time (s)",
00026 };
00027
00028 static const abip_int HSPACE = 9;
00029 static const abip_int HEADER_LEN = 10;
00030 static const abip_int LINE_LEN = 150;
00031
00032 static abip_int abip_isnan(abip_float x) {
00033     DEBUG_FUNC
00034     RETURN(x == NAN || x != x);
00035 }
00036
00037 static void free_work(ABIPWork *w) {
00038     DEBUG_FUNC
00039
00040     if (!w) {
00041         RETURN;
00042     }
00043
00044     if (w->u) {
00045         abip_free(w->u);
00046     }
00047
00048     if (w->v) {
00049         abip_free(w->v);
00050     }
00051
00052     if (w->u_t) {
00053         abip_free(w->u_t);
00054     }
00055
00056     if (w->rel_ut) {
00057         abip_free(w->rel_ut);
00058     }
00059
00060     if (w->A) {
00061         ABIP(free_A_matrix)(w->A);
00062     }
00063
00064     abip_free(w);
00065
00066     RETURN;
00067 }
00068
00069 static void print_init_header(spe_problem *spe) {
00070     DEBUG_FUNC
00071
00072     abip_int i;
00073     ABIPSettings *stgs = spe->stgs;
00074     char *lin_sys_method = ABIP(get_lin_sys_method)(spe);
00075
00076     for (i = 0; i < LINE_LEN; ++i) {
00077         abip_printf("-");
00078     }
00079
00080     abip_printf(
00081         "\n\tABIP v%s - First-Order Interior-Point Solver for Conic "
00082         "Programming\n\t(c) Jinsong Liu & LEAVES Group, 2021-2024\n",

```

```

00083     ABIP(version)();
00084
00085     for (i = 0; i < LINE_LEN; ++i) {
00086         abip_printf("-");
00087     }
00088
00089     abip_printf("\n");
00090
00091     if (lin_sys_method) {
00092         abip_printf("Lin-sys: %s\n", lin_sys_method);
00093         abip_free(lin_sys_method);
00094     }
00095
00096     if (stgs->normalize) {
00097         abip_printf(
00098             "eps_p = %.2e, eps_d = %.2e, eps_g = %.2e, alpha = %.2f, max_ipm_iters "
00099             "= %i, max_admm_iters = %i, normalize = %i\n"
00100             "rho_y = %.2e\n",
00101             stgs->eps_p, stgs->eps_d, stgs->eps_g, stgs->alpha,
00102             (int)stgs->max_ipm_iters, (int)stgs->max_admm_iters,
00103             (int)stgs->normalize, stgs->rho_y);
00104     } else {
00105         abip_printf(
00106             "eps_p = %.2e, eps_d = %.2e, eps_g = %.2e, alpha = %.2f, max_ipm_iters "
00107             "= %i, max_admm_iters = %i, normalize = %i\n"
00108             "rho_y = %.2e\n",
00109             stgs->eps_p, stgs->eps_d, stgs->eps_g, stgs->alpha,
00110             (int)stgs->max_ipm_iters, (int)stgs->max_admm_iters,
00111             (int)stgs->normalize, stgs->rho_y);
00112     }
00113
00114     abip_printf("constraints m = %i, Variables n = %i\n", (int)spe->p,
00115                 (int)spe->q);
00116
00117 #ifdef MATLAB_MEX_FILE
00118     mexEvalString("drawnow;");
00119 #endif
00120
00121     RETURN;
00122 }
00123
00124 static void populate_on_failure(abip_int m, abip_int n, ABIPSolution *sol,
00125                               ABIPInfo *info, abip_int status_val,
00126                               const char *msg) {
00127     DEBUG_FUNC
00128
00129     if (info) {
00130         info->res_pri = NAN;
00131         info->res_dual = NAN;
00132         info->rel_gap = NAN;
00133         info->res_infeas = NAN;
00134         info->res_unbdd = NAN;
00135
00136         info->pobj = NAN;
00137         info->dobj = NAN;
00138
00139         info->ipm_iter = -1;
00140         info->admm_iter = -1;
00141         info->status_val = status_val;
00142         info->solve_time = NAN;
00143         strcpy(info->status, msg);
00144     }
00145
00146     if (sol) {
00147         if (n > 0) {
00148             if (!sol->x) {
00149                 sol->x = (abip_float *)abip_malloc(sizeof(abip_float) * n);
00150             }
00151             ABIP(scale_array)(sol->x, NAN, n);
00152
00153             if (!sol->s) {
00154                 sol->s = (abip_float *)abip_malloc(sizeof(abip_float) * n);
00155             }
00156             ABIP(scale_array)(sol->s, NAN, m);
00157         }
00158
00159         if (m > 0) {
00160             if (!sol->y) {
00161                 sol->y = (abip_float *)abip_malloc(sizeof(abip_float) * m);
00162             }
00163             ABIP(scale_array)(sol->y, NAN, m);
00164         }
00165     }
00166
00167     RETURN;
00168 }
00169
00170 RETURN;

```

```

00170 }
00171
00172 static abip_int failure(ABIPWork *w, abip_int m, abip_int n, ABIPSolution *sol,
00173                        ABIPInfo *info, abip_int stint, const char *msg,
00174                        const char *ststr) {
00175     DEBUG_FUNC
00176
00177     abip_int status = stint;
00178     populate_on_failure(m, n, sol, info, status, ststr);
00179
00180     abip_printf("Failure:%s\n", msg);
00181     abip_end_interrupt_listener();
00182
00183     RETURN status;
00184 }
00185
00186 static abip_int projection(ABIPWork *w, abip_int iter, spe_problem *spe,
00187                           ABIPResiduals *r) {
00188     DEBUG_FUNC
00189
00190     abip_int status;
00191
00192     if (spe->Q != ABIP_NULL || spe->stgs->linsys_solver != 3) {
00193         abip_int n = w->n;
00194         abip_int m = w->m;
00195         abip_int l = n + m + 1;
00196         abip_float a = w->a;
00197
00198         abip_float *mu = (abip_float *)abip_malloc((m + n) * sizeof(abip_float));
00199         memcpy(mu, w->u, (m + n) * sizeof(abip_float));
00200         ABIP(add_scaled_array)(mu, w->v, m + n, 1);
00201         ABIP(c_dot)(mu, spe->rho_dr, m + n);
00202         abip_float eta = spe->rho_dr[m + n] * (w->u[m + n] + w->v[m + n]);
00203
00204         abip_float *warm_start =
00205             (abip_float *)abip_malloc((m + n) * sizeof(abip_float));
00206         memcpy(warm_start, w->u, (m + n) * sizeof(abip_float));
00207
00208         abip_float *tem = (abip_float *)abip_malloc((m + n) * sizeof(abip_float));
00209
00210         abip_float pcg_tol = 0;
00211         if (spe->stgs->linsys_solver == 3) { // create warm start for pcg
00212             ABIP(add_scaled_array)(warm_start, w->r, m + n, w->u[m + n]);
00213
00214             pcg_tol = MIN(r->Ax_b_norm, r->Qx_ATy_c_s_norm);
00215             pcg_tol = 0.2 * MIN(pcg_tol, ABIP(norm_inf)(warm_start, n) /
00216                               POWF((abip_float)iter + 1, 1.5));
00217
00218             pcg_tol = MAX(pcg_tol, 1e-12);
00219         }
00220
00221         abip_float *p = (abip_float *)abip_malloc((m + n) * sizeof(abip_float));
00222         memcpy(p, mu, (m + n) * sizeof(abip_float));
00223
00224         status = spe->solve_spe_linsys(spe, p, warm_start, iter, pcg_tol);
00225
00226         memcpy(tem, p, (m + n) * sizeof(abip_float));
00227         ABIP(c_dot)(tem, spe->rho_dr, m + n);
00228
00229         abip_float b =
00230             ABIP(dot)(w->r, mu, m + n) - 2 * ABIP(dot)(w->r, tem, m + n) - eta;
00231
00232         abip_float *Qp = (abip_float *)abip_malloc(n * sizeof(abip_float));
00233         memset(Qp, 0, n * sizeof(abip_float));
00234
00235         if (spe->Q != ABIP_NULL) {
00236             ABIP(accum_by_A)(spe->Q, &p[m], Qp);
00237         }
00238
00239         abip_float c = -ABIP(dot)(&p[m], Qp, n);
00240
00241         if (iter > 0) {
00242             w->u_t[m + n] = (-b + SQRTF(MAX(0, b * b - 4 * a * c))) / (2 * a);
00243         } else {
00244             w->u_t[m + n] = 1;
00245         }
00246
00247         memcpy(w->u_t, p, (m + n) * sizeof(abip_float));
00248         ABIP(add_scaled_array)(w->u_t, w->r, m + n, -w->u_t[m + n]);
00249
00250         abip_free(mu);
00251         abip_free(warm_start);
00252         abip_free(p);
00253         abip_free(tem);
00254         abip_free(Qp);
00255     } else {
00256         abip_int n = w->n;

```

```

00257     abip_int m = w->m;
00258     abip_int l = n + m + 1;
00259     abip_float a = w->a;
00260
00261     abip_float *mu = (abip_float *)abip_malloc((m + n) * sizeof(abip_float));
00262     memcpy(mu, w->u, (m + n) * sizeof(abip_float));
00263     ABIP(add_scaled_array)(mu, w->v, m + n, 1);
00264     ABIP(c_dot)(mu, spe->rho_dr, m + n);
00265
00266     abip_float eta = spe->rho_dr[m + n] * (w->u[m + n] + w->v[m + n]);
00267
00268     abip_float *w3 = (abip_float *)abip_malloc((m + n) * sizeof(abip_float));
00269     memcpy(w3, mu, (m + n) * sizeof(abip_float));
00270     ABIP(add_scaled_array)(w3, spe->b, m, eta);
00271     ABIP(add_scaled_array)(&w3[m], spe->c, n, -eta);
00272
00273     abip_float *warm_start =
00274         (abip_float *)abip_malloc((m + n) * sizeof(abip_float));
00275     memcpy(warm_start, w->u, (m + n) * sizeof(abip_float));
00276
00277     abip_float pcg_tol = 0;
00278     if (spe->stgs->linsys_solver == 3) { // create warm start for pcg
00279
00280         ABIP(add_scaled_array)(warm_start, w->r, m + n, w->u[m + n]);
00281
00282         pcg_tol = MIN(r->Ax_b_norm, r->Qx_ATy_c_s_norm);
00283         pcg_tol = 0.2 * MIN(pcg_tol, ABIP(norm_inf)(warm_start, n) /
00284                             POWF((abip_float)iter + 1, 1.5));
00285
00286         pcg_tol = MAX(pcg_tol, 1e-12);
00287     }
00288
00289     memset(warm_start, 0, (m + n) * sizeof(abip_float));
00290
00291     status = spe->solve_spe_linsys(spe, w3, warm_start, iter, r->error_ratio);
00292
00293     abip_float coef =
00294         -(-ABIP(dot)(spe->b, w3, m) + ABIP(dot)(spe->c, &w3[m], n)) /
00295         (1 - ABIP(dot)(spe->b, w->r, m) + ABIP(dot)(spe->c, &w->r[m], n));
00296
00297     memcpy(w->u_t, w3, (m + n) * sizeof(abip_float));
00298     ABIP(add_scaled_array)(w->u_t, w->r, m + n, coef);
00299     if (iter > 0) {
00300         w->u_t[m + n] =
00301             eta - ABIP(dot)(spe->b, w->u_t, m) + ABIP(dot)(spe->c, &w->u_t[m], n);
00302     } else {
00303         w->u_t[m + n] = 1;
00304     }
00305
00306     abip_free(mu);
00307     abip_free(w3);
00308     abip_free(warm_start);
00309 }
00310
00311 RETURN status;
00312 }
00313
00314 static void update_dual_vars(ABIPWork *w) {
00315     DEBUG_FUNC
00316
00317     abip_int l = w->m + w->n + 1;
00318
00319     memcpy(w->v, w->u, l * sizeof(abip_float));
00320
00321     ABIP(add_scaled_array)(w->v, w->rel_ut, l, -1);
00322
00323     RETURN;
00324 }
00325
00326 static void solve_barrier_subproblem(ABIPWork *w, const ABIPConc *c,
00327                                     spe_problem *spe) {
00328     DEBUG_FUNC
00329
00330     abip_int n = w->n;
00331     abip_int m = w->m;
00332     abip_int l = m + n + 1;
00333     abip_float lambda = w->mu / w->beta;
00334
00335     // over relaxation: rel_ut = alpha * ut + (1 - alpha) * u
00336     memcpy(w->rel_ut, w->u_t, l * sizeof(abip_float));
00337     ABIP(scale_array)(w->rel_ut, spe->stgs->alpha, l);
00338     ABIP(add_scaled_array)(w->rel_ut, w->u, l, 1 - spe->stgs->alpha);
00339
00340     abip_float *tmp = (abip_float *)abip_malloc(l * sizeof(abip_float));
00341
00342     ABIP(add_scaled_array)(w->rel_ut, w->v, l, -1);
00343     memcpy(tmp, w->rel_ut, l * sizeof(abip_float));

```

```

00344
00345     abip_float *y = (abip_float *)abip_malloc(m * sizeof(abip_float));
00346     memcpy(y, tmp, m * sizeof(abip_float));
00347
00348     abip_float tau = (tmp[l - 1] + SQRTF(tmp[l - 1] * tmp[l - 1] +
00349                                     4 * lambda / spe->rho_dr[l - 1])) /
00350                     2;
00351
00352     memcpy(w->u, y, m * sizeof(abip_float));
00353     w->u[l - 1] = tau;
00354
00355     abip_int count = 0;
00356     abip_int i;
00357
00358     /*soc*/
00359     if (c->qsize && c->q) {
00360         for (i = 0; i < c->qsize; ++i) {
00361             if (c->q[i] == 0) {
00362                 continue;
00363             }
00364             if (c->q[i] == 1) {
00365                 ABIP(positive_orthant_barrier_subproblem)
00366                 (&w->u[m + count], &tmp[m + count], lambda / spe->rho_dr[m + count], 1);
00367             } else {
00368                 ABIP(soc_barrier_subproblem)
00369                 (&w->u[m + count], &tmp[m + count], lambda / spe->rho_dr[m + count],
00370                  c->q[i]);
00371             }
00372             count += c->q[i];
00373         }
00374     }
00375
00376     /*rsoc*/
00377     if (c->rsize && c->r) {
00378         for (i = 0; i < c->rsize; ++i) {
00379             if (c->r[i] < 3) {
00380                 continue;
00381             } else {
00382                 ABIP(rsoc_barrier_subproblem)
00383                 (&w->u[m + count], &tmp[m + count], lambda / spe->rho_dr[m + count],
00384                  c->r[i]);
00385             }
00386             count += c->r[i];
00387         }
00388     }
00389
00390     /*free cone*/
00391     if (c->f) {
00392         ABIP(free_barrier_subproblem)
00393         (&w->u[m + count], &tmp[m + count], lambda / spe->rho_dr[m + count], c->f);
00394         count += c->f;
00395     }
00396
00397     /*zero cone*/
00398     if (c->z) {
00399         ABIP(zero_barrier_subproblem)
00400         (&w->u[m + count], &tmp[m + count], lambda / spe->rho_dr[m + count], c->z);
00401         count += c->z;
00402     }
00403
00404     /*positive orthant*/
00405     if (c->l) {
00406         ABIP(positive_orthant_barrier_subproblem)
00407         (&w->u[m + count], &tmp[m + count], lambda / spe->rho_dr[m + count], c->l);
00408         count += c->l;
00409     }
00410
00411     abip_free(tmp);
00412     abip_free(y);
00413 }
00414
00415 static abip_int indeterminate(ABIPWork *w, ABIPSolution *sol, ABIPInfo *info) {
00416     DEBUG_FUNC
00417
00418     strcpy(info->status, "Indeterminate");
00419
00420     ABIP(scale_array)(sol->x, NAN, w->n);
00421     ABIP(scale_array)(sol->y, NAN, w->m);
00422     ABIP(scale_array)(sol->s, NAN, w->n);
00423
00424     RETURN ABIP_INDETERMINATE;
00425 }
00426
00427 static abip_int solved(ABIPWork *w, ABIPSolution *sol, ABIPInfo *info,
00428                       abip_float tau) {
00429     DEBUG_FUNC
00430

```

```

00431 ABIP(scale_array)(sol->x, SAFEDIV_POS(1.0, tau), w->n);
00432 ABIP(scale_array)(sol->y, SAFEDIV_POS(1.0, tau), w->m);
00433 ABIP(scale_array)(sol->s, SAFEDIV_POS(1.0, tau), w->n);
00434
00435 if ((info->status_val == 0) || (info->status_val == 2)) {
00436     strcpy(info->status, "Solved/Inaccurate");
00437     RETURN ABIP_SOLVED_INACCURATE;
00438 }
00439
00440 strcpy(info->status, "Solved");
00441
00442 RETURN ABIP_SOLVED;
00443 }
00444 static void sety(ABIPWork *w, ABIPSolution *sol) {
00445     DEBUG_FUNC
00446
00447     if (!sol->y) {
00448         sol->y = (abip_float *)abip_malloc(sizeof(abip_float) * w->m);
00449     }
00450
00451     memcpy(sol->y, w->u, w->m * sizeof(abip_float));
00452
00453     RETURN;
00454 }
00455
00456 static void setx(ABIPWork *w, ABIPSolution *sol) {
00457     DEBUG_FUNC
00458
00459     if (!sol->x) {
00460         sol->x = (abip_float *)abip_malloc(sizeof(abip_float) * w->n);
00461     }
00462
00463     memcpy(sol->x, &(w->u[w->m]), w->n * sizeof(abip_float));
00464
00465     RETURN;
00466 }
00467
00468 static void sets(ABIPWork *w, ABIPSolution *sol) {
00469     DEBUG_FUNC
00470
00471     if (!sol->s) {
00472         sol->s = (abip_float *)abip_malloc(sizeof(abip_float) * w->n);
00473     }
00474
00475     memcpy(sol->s, &(w->v[w->m]), w->n * sizeof(abip_float));
00476
00477     RETURN;
00478 }
00479
00480 static abip_int infeasible(ABIPWork *w, ABIPSolution *sol, ABIPInfo *info,
00481                             abip_float bt_y) {
00482     DEBUG_FUNC
00483
00484     ABIP(scale_array)(sol->y, 1 / bt_y, w->m);
00485     ABIP(scale_array)(sol->s, 1 / bt_y, w->n);
00486     ABIP(scale_array)(sol->x, NAN, w->n);
00487
00488     if (info->status_val == 0) {
00489         strcpy(info->status, "Infeasible/Inaccurate");
00490         RETURN ABIP_INFEASIBLE_INACCURATE;
00491     }
00492
00493     strcpy(info->status, "Infeasible");
00494     RETURN ABIP_INFEASIBLE;
00495 }
00496
00497 static abip_int unbounded(ABIPWork *w, ABIPSolution *sol, ABIPInfo *info,
00498                             abip_float ct_x) {
00499     DEBUG_FUNC
00500
00501     ABIP(scale_array)(sol->x, -1 / ct_x, w->n);
00502     ABIP(scale_array)(sol->y, NAN, w->m);
00503     ABIP(scale_array)(sol->s, NAN, w->n);
00504
00505     if (info->status_val == 0) {
00506         strcpy(info->status, "Unbounded/Inaccurate");
00507         RETURN ABIP_UNBOUNDED_INACCURATE;
00508     }
00509
00510     strcpy(info->status, "Unbounded");
00511     RETURN ABIP_UNBOUNDED;
00512 }
00513
00514 static abip_int is_solved_status(abip_int status) {
00515     RETURN status == ABIP_SOLVED || status == ABIP_SOLVED_INACCURATE;
00516 }
00517

```

```

00518 static abip_int is_infeasible_status(abip_int status) {
00519     RETURN status == ABIP_INFEASIBLE || status == ABIP_INFEASIBLE_INACCURATE;
00520 }
00521
00522 static abip_int is_unbounded_status(abip_int status) {
00523     RETURN status == ABIP_UNBOUNDED || status == ABIP_UNBOUNDED_INACCURATE;
00524 }
00525
00526 static void get_info(ABIPWork *w, ABIPSolution *sol, ABIPInfo *info,
00527                     ABIPResiduals *r, abip_int ipm_iter, abip_int admm_iter) {
00528     DEBUG_FUNC
00529
00530     info->ipm_iter = ipm_iter + 1;
00531     info->admm_iter = admm_iter;
00532
00533     info->res_infeas = r->res_infeas;
00534     info->res_unbdd = r->res_unbdd;
00535
00536     if (is_solved_status(info->status_val)) {
00537         info->rel_gap = r->rel_gap;
00538         info->res_pri = r->res_pri;
00539         info->res_dual = r->res_dual;
00540         info->pobj = r->pobj;
00541         info->dobj = r->dobj;
00542     } else if (is_unbounded_status(info->status_val)) {
00543         info->rel_gap = NAN;
00544         info->res_pri = NAN;
00545         info->res_dual = NAN;
00546         info->pobj = -INFINITY;
00547         info->dobj = -INFINITY;
00548     } else if (is_infeasible_status(info->status_val)) {
00549         info->rel_gap = NAN;
00550         info->res_pri = NAN;
00551         info->res_dual = NAN;
00552         info->pobj = INFINITY;
00553         info->dobj = INFINITY;
00554     }
00555
00556     RETURN;
00557 }
00558
00559 static void get_solution(ABIPWork *w, spe_problem *spe, ABIPSolution *sol,
00560                          ABIPInfo *info, ABIPResiduals *r, abip_int ipm_iter,
00561                          abip_int admm_iter) {
00562     DEBUG_FUNC
00563
00564     abip_int l = w->m + w->n + 1;
00565
00566     setx(w, sol);
00567     sety(w, sol);
00568     sets(w, sol);
00569
00570     if (info->status_val == ABIP_UNFINISHED) {
00571         info->status_val = solved(w, sol, info, r->tau);
00572     } else if (is_solved_status(info->status_val)) {
00573         info->status_val = solved(w, sol, info, r->tau);
00574     } else if (is_infeasible_status(info->status_val)) {
00575         info->status_val = infeasible(w, sol, info, r->dobj * r->tau);
00576     } else {
00577         info->status_val = unbounded(w, sol, info, r->pobj * r->tau);
00578     }
00579
00580     if (spe->stgs->normalize) {
00581         spe->un_scaling_sol(spe, sol);
00582     }
00583
00584     get_info(w, sol, info, r, ipm_iter, admm_iter);
00585
00586     RETURN;
00587 }
00588
00589 static void print_summary(ABIPWork *w, spe_problem *spe, abip_int i, abip_int j,
00590                          abip_int k, ABIPResiduals *r,
00591                          ABIP(timer) * solve_timer) {
00592     DEBUG_FUNC
00593
00594     if (!spe->stgs->verbose) {
00595         RETURN;
00596     }
00597
00598     abip_printf("%*i|", (int)strlen(HEADER[0]), (int)i + 1);
00599     abip_printf("%*i|", (int)strlen(HEADER[1]), (int)j + 1);
00600     abip_printf("%*.2e|", (int)strlen(HEADER[2]), w->mu);
00601
00602     abip_printf("%*.2e|", (int)HSPACE, r->res_pri / spe->stgs->eps_p);
00603     abip_printf("%*.2e|", (int)HSPACE, r->res_dual / spe->stgs->eps_d);
00604     abip_printf("%*.2e|", (int)HSPACE, r->rel_gap / spe->stgs->eps_g);

```

```

00605     abip_printf("%*.2e|", (int)HSPACE, r->pobj);
00606     abip_printf("%*.2e|", (int)HSPACE, r->dobj);
00607     abip_printf("%*.2e|", (int)HSPACE, r->tau);
00608
00609     abip_printf("%*.2e ", (int)HSPACE, ABIP(tocq)(solve_timer) / 1e3);
00610     abip_printf("\n");
00611
00612     #if EXTRA_VERBOSE > 0
00613
00614     abip_printf("Norm u = %4f, ", ABIP(norm)(w->u, w->n + w->m + 2));
00615     abip_printf("Norm u_t = %4f, ", ABIP(norm)(w->u_t, w->n + w->m + 2));
00616     abip_printf("Norm v = %4f, ", ABIP(norm)(w->v, w->n + w->m + 2));
00617     abip_printf("tau = %4f, ", r->tau);
00618     abip_printf("kappa = %4f, ", r->kap);
00619     abip_printf("|u - u_t| = %1.2e, ",
00620                 ABIP(norm_diff)(w->u, w->u_t, w->n + w->m + 2));
00621     abip_printf("res_infeas = %1.2e, ", r->res_infeas);
00622     abip_printf("res_unbdd = %1.2e\n", r->res_unbdd);
00623
00624     #endif
00625
00626     #ifdef MATLAB_MEX_FILE
00627         mexEvalString("drawnow;");
00628     #endif
00629
00630     #endif
00631
00632     RETURN;
00633 }
00634
00635 static void print_header(ABIPWork *w) {
00636     DEBUG_FUNC
00637
00638     abip_int i;
00639
00640     for (i = 0; i < LINE_LEN; ++i) {
00641         abip_printf("-");
00642     }
00643     abip_printf("\n");
00644
00645     for (i = 0; i < HEADER_LEN - 1; ++i) {
00646         abip_printf("%s|", HEADER[i]);
00647     }
00648     abip_printf("%s\n", HEADER[HEADER_LEN - 1]);
00649
00650     for (i = 0; i < LINE_LEN; ++i) {
00651         abip_printf("-");
00652     }
00653     abip_printf("\n");
00654
00655     #ifdef MATLAB_MEX_FILE
00656         mexEvalString("drawnow;");
00657     #endif
00658
00659     #endif
00660
00661     RETURN;
00662 }
00663
00664 static void print_footer(const ABIPData *d, spe_problem *spe, ABIPSolution *sol,
00665                         ABIPWork *w, ABIPInfo *info, abip_int k) {
00666     DEBUG_FUNC
00667
00668     abip_int i;
00669
00670     char *lin_sys_str = ABIP(get_lin_sys_summary)(spe, info);
00671
00672     for (i = 0; i < LINE_LEN; ++i) {
00673         abip_printf("-");
00674     }
00675
00676     abip_printf("\n");
00677
00678     abip_printf("Status: %s\n", info->status);
00679
00680     if (info->ipm_iter + 1 == spe->stgs->max_ipm_iters) {
00681         abip_printf("Hit max_ipm_iters, solution may be inaccurate\n");
00682     }
00683
00684     if (info->admm_iter + 1 >= spe->stgs->max_admm_iters) {
00685         abip_printf("Hit max_admm_iters, solution may be inaccurate\n");
00686     }
00687
00688     abip_printf("Timing: Solve time: %1.2es\n", info->solve_time / 1e3);
00689     abip_printf("        per iter: %1.2es\n", info->solve_time / (k * 1e3));
00690     abip_printf("        Total time: %1.2es\n",
00691                 (info->solve_time + info->setup_time) / 1e3);

```



```

00692
00693     if (lin_sys_str) {
00694         abip_printf("%s", lin_sys_str);
00695         abip_free(lin_sys_str);
00696     }
00697
00698     for (i = 0; i < LINE_LEN; ++i) {
00699         abip_printf("-");
00700     }
00701
00702     abip_printf("\n");
00703
00704     if (is_infeasible_status(info->status_val)) {
00705         abip_printf("Certificate of primal infeasibility:\n");
00706         abip_printf("|A'y + s|_2 * |b|_2 = %.4e\n", info->res_infeas);
00707         abip_printf("b'y = %.4f\n", ABIP(dot)(d->b, sol->y, d->m));
00708     } else if (is_unbounded_status(info->status_val)) {
00709         abip_printf("Certificate of dual infeasibility:\n");
00710         abip_printf("|Ax|_2 * |c|_2 = %.4e\n", info->res_unbdd);
00711         abip_printf("c'x = %.4f\n", ABIP(dot)(d->c, sol->x, d->n));
00712     } else {
00713         abip_printf("Error metrics:\n");
00714         abip_printf(
00715             "primal res: |Ax - b|_inf / (1 + max(|Ax|_inf, |b|_inf)) = %.4e\n",
00716             info->res_pri);
00717         abip_printf(
00718             "dual res: |Qx - A'y + c - s|_inf / (1 + max(|Qx|_inf + |c|_inf)) = "
00719             "%.4e\n",
00720             info->res_dual);
00721         abip_printf(
00722             "rel gap: |x'Qx + c'x - b'y| / (1 + max(|x'Qx| + |c'x| + |b'y|)) = "
00723             "%.4e\n",
00724             info->rel_gap);
00725
00726         for (i = 0; i < LINE_LEN; ++i) {
00727             abip_printf("-");
00728         }
00729
00730         abip_printf("\n");
00731         abip_printf("1/2x'Qx + c'x = %.4e, -1/2x'Qx + b'y = %.4e\n", info->pobj,
00732             info->dobj);
00733     }
00734
00735     for (i = 0; i < LINE_LEN; ++i) {
00736         abip_printf("=");
00737     }
00738
00739     abip_printf("\n");
00740
00741 #ifdef MATLAB_MEX_FILE
00742     mexEvalString("drawnow;");
00743 #endif
00744     RETURN;
00745 }
00746
00747 static abip_int has_converged(ABIPWork *w, spe_problem *spe, ABIPResiduals *r,
00748     abip_int ipm_iter, abip_int admm_iter) {
00749     DEBUG_FUNC
00750
00751     abip_float eps_p = spe->stgs->eps_p;
00752     abip_float eps_d = spe->stgs->eps_d;
00753     abip_float eps_g = spe->stgs->eps_g;
00754     abip_float eps_inf = spe->stgs->eps_inf;
00755     abip_float eps_unb = spe->stgs->eps_unb;
00756
00757     if (r->res_pri < eps_p && r->res_dual < eps_d && r->rel_gap < eps_g) {
00758         RETURN ABIP_SOLVED;
00759     }
00760
00761     if (r->res_dif < spe->stgs->err_dif * MAX(MAX(eps_p, eps_d), eps_g)) {
00762         RETURN ABIP_SOLVED_INACCURATE;
00763     }
00764
00765     if (r->res_unbdd < eps_unb && ipm_iter > 0 && admm_iter > 0) {
00766         RETURN ABIP_UNBOUNDED;
00767     }
00768
00769     if (r->res_infeas < eps_inf && ipm_iter > 0 && admm_iter > 0) {
00770         RETURN ABIP_INFEASIBLE;
00771     }
00772
00773     RETURN 0;
00774 }
00775
00776
00777
00778

```

```

00779 static abip_int validate(const ABIPData *d, const ABIPConc *k,
00780                          spe_problem *spe) {
00781     DEBUG_FUNC
00782
00783     ABIPSettings *stgs = d->stgs;
00784
00785     if (d->n <= 0) {
00786         abip_printf("n must be greater than 0; n = %li\n", (long)d->n);
00787         RETURN - 1;
00788     }
00789
00790     if (spe->p > spe->q) {
00791         abip_printf("WARN: m larger than n, problem likely degenerate\n");
00792         RETURN - 1;
00793     }
00794
00795     if (ABIP(validate_lin_sys)(d->A) < 0) {
00796         abip_printf("invalid linear system input data\n");
00797         RETURN - 1;
00798     }
00799
00800     if (ABIP(validate_cones)(spe, k) < 0) {
00801         abip_printf("cone validation error\n");
00802         return -1;
00803     }
00804
00805     if (stgs->max_ipm_iters <= 0) {
00806         abip_printf("max_ipm_iters must be positive\n");
00807         RETURN - 1;
00808     }
00809
00810     if (stgs->max_admm_iters <= 0) {
00811         abip_printf("max_admm_iters must be positive\n");
00812         RETURN - 1;
00813     }
00814
00815     if (stgs->eps_p <= 0 || stgs->eps_d <= 0 || stgs->eps_g <= 0 ||
00816         stgs->eps_inf <= 0 || stgs->eps_unb <= 0) {
00817         abip_printf("eps tolerance must be positive\n");
00818         RETURN - 1;
00819     }
00820
00821     if (stgs->alpha <= 0 || stgs->alpha >= 2) {
00822         abip_printf("alpha must be in (0,2)\n");
00823         RETURN - 1;
00824     }
00825
00826     if (stgs->rho_y <= 0) {
00827         abip_printf("rho_y must be positive (1e-3 works well).\n");
00828         RETURN - 1;
00829     }
00830
00831     RETURN 0;
00832 }
00833
00834 static ABIPWork *init_work(spe_problem *s, ABIPConc *k) {
00835     DEBUG_FUNC
00836
00837     ABIPWork *w = (ABIPWork *)abip_calloc(1, sizeof(ABIPWork));
00838     abip_int l = s->p + s->q + 1;
00839
00840     if (s->stgs->verbose) {
00841         print_init_header(s);
00842     }
00843
00844     if (!w) {
00845         abip_printf("ERROR: allocating work failure\n");
00846         RETURN ABIP_NULL;
00847     }
00848
00849     w->sigma = SIGMA;
00850     w->gamma = GAMMA;
00851
00852     w->mu = 1.0;
00853     w->beta = 1.0;
00854
00855     w->m = s->p;
00856     w->n = s->q;
00857
00858     w->u = (abip_float *)abip_malloc(1 * sizeof(abip_float));
00859     w->v = (abip_float *)abip_malloc(1 * sizeof(abip_float));
00860     memset(w->u, 0, 1 * sizeof(abip_float));
00861     memset(w->v, 0, 1 * sizeof(abip_float));
00862
00863     w->v_origin = (abip_float *)abip_malloc(1 * sizeof(abip_float));
00864     w->u_t = (abip_float *)abip_malloc(1 * sizeof(abip_float));
00865     w->rel_ut = (abip_float *)abip_malloc(1 * sizeof(abip_float));

```

```

00866     w->r = (abip_float *)abip_malloc((w->n + w->m) * sizeof(abip_float));
00867
00868     if (!w->u || !w->v || !w->u_t || !w->rel_ut || !w->v_origin || !w->r) {
00869         abip_printf("ERROR: work memory allocation failure\n");
00870         RETURN ABIP_NULL;
00871     }
00872
00873     w->nm_inf_b = ABIP(norm_inf)(s->data->b, s->p);
00874     w->nm_inf_c = ABIP(norm_inf)(s->data->c, s->q);
00875
00876     s->scaling_data(s, k);
00877
00878     if (s->init_spe_linsys_work(s)) {
00879         abip_printf("ERROR: init_lin_sys_work failure\n");
00880         RETURN ABIP_NULL;
00881     }
00882
00883     RETURN w;
00884 }
00885
00886 static abip_int pre_calculate(ABIPWork *w, spe_problem *spe) {
00887     DEBUG_FUNC
00888
00889     abip_int m = w->m;
00890     abip_int n = w->n;
00891
00892     abip_float *zeros = (abip_float *)abip_malloc((m + n) * sizeof(abip_float));
00893     memset(zeros, 0, (m + n) * sizeof(abip_float));
00894
00895     memcpy(w->r, spe->b, m * sizeof(abip_float));
00896     ABIP(scale_array)(w->r, -1, m);
00897     memcpy(&w->r[m], spe->c, n * sizeof(abip_float));
00898
00899     spe->solve_spe_linsys(spe, w->r, ABIP_NULL, -1, 1e-12);
00900
00901     abip_float *tem = (abip_float *)abip_malloc((m + n) * sizeof(abip_float));
00902     memcpy(tem, w->r, (m + n) * sizeof(abip_float));
00903     ABIP(c_dot)(tem, spe->rho_dr, m + n);
00904     w->a = spe->rho_dr[m + n] + ABIP(dot)(tem, w->r, m + n);
00905
00906     abip_free(tem);
00907     abip_free(zeros);
00908
00909     RETURN 0;
00910 }
00911
00912 abip_int update_work(const ABIPData *d, ABIPWork *w, const ABIPSolution *sol,
00913                     const ABIPConc *c, spe_problem *spe)
00914 {
00915     DEBUG_FUNC
00916
00917     abip_int n = w->n;
00918     abip_int m = w->m;
00919
00920     // initialize x,y
00921     abip_float *x = (abip_float *)abip_malloc(n * sizeof(abip_float));
00922     abip_float *y = (abip_float *)abip_malloc(m * sizeof(abip_float));
00923     memset(y, 0, m * sizeof(abip_float));
00924     /*-----initialize x-----*/
00925     abip_int i;
00926     abip_int count = 0;
00927
00928     // soc
00929     if (c->qsize && c->q) {
00930         for (i = 0; i < c->qsize; ++i) {
00931             if (c->q[i] == 0) {
00932                 continue;
00933             }
00934             memset(&x[count], 0, c->q[i] * sizeof(abip_float));
00935             x[count] = 1;
00936             count += c->q[i];
00937         }
00938     }
00939
00940     // rsoc
00941     if (c->rqsize && c->rq) {
00942         for (i = 0; i < c->rqsize; ++i) {
00943             if (c->rq[i] < 3) {
00944                 continue;
00945             }
00946             memset(&x[count], 0, c->rq[i] * sizeof(abip_float));
00947             x[count] = 1;
00948             x[count + 1] = 1;
00949             count += c->rq[i];
00950         }
00951     }
00952 }

```

```

00953
00954 // free cone
00955 if (c->f) {
00956     for (i = count; i < count + c->f; i++) {
00957         x[i] = 0;
00958     }
00959     count += c->f;
00960 }
00961
00962 // zero cone
00963 if (c->z) {
00964     for (i = count; i < count + c->z; i++) {
00965         x[i] = 0;
00966     }
00967     count += c->z;
00968 }
00969
00970 // positive orthant
00971 if (c->l) {
00972     for (i = count; i < count + c->l; i++) {
00973         x[i] = 1;
00974     }
00975     count += c->l;
00976 }
00977
00978 // initialize u,v
00979 memcpy(w->u, y, m * sizeof(abip_float));
00980 memcpy(&w->u[m], x, n * sizeof(abip_float));
00981 w->u[m + n] = 1.0;
00982 // w->u[m+n+1] = 1;
00983 memcpy(w->v, y, m * sizeof(abip_float));
00984 memcpy(&w->v[m], x, n * sizeof(abip_float));
00985 w->v[m + n] = 1.0;
00986
00987 pre_calculate(w, spe);
00988
00989 abip_free(x);
00990 abip_free(y);
00991 return 0;
00992 }
00993
00994 abip_float adjust_barrier(ABIPWork *w, ABIPResiduals *r, spe_problem *spe) {
00995     abip_float error_ratio = r->error_ratio;
00996
00997     abip_float sigma = 0.8;
00998     abip_float ratio =
00999         w->mu / MIN(MIN(spe->stgs->eps_p, spe->stgs->eps_d), spe->stgs->eps_g);
01000     abip_float gamma;
01001
01002     if (ratio > 50 && ratio <= 100) {
01003         gamma = 1.5;
01004     } else if (ratio > 10 && ratio <= 50) {
01005         gamma = 1.3;
01006     } else if (ratio > 5 && ratio <= 10) {
01007         gamma = 1.2;
01008     } else if (ratio > 1 && ratio <= 5) {
01009         gamma = 1.1;
01010     } else if (ratio > 0.5 && ratio <= 1) {
01011         gamma = 1;
01012     } else if (ratio > 0.1 && ratio <= 0.5) {
01013         gamma = 0.9;
01014     } else if (ratio > 0.05 && ratio <= 0.1) {
01015         gamma = 0.9;
01016     } else if (ratio > 0.01 && ratio <= 0.05) {
01017         gamma = 0.8;
01018     } else if (ratio > 0.005 && ratio <= 0.01) {
01019         gamma = 0.8;
01020     } else if (ratio > 0.001 && ratio <= 0.005) {
01021         gamma = 0.7;
01022     } else if (ratio > 0.0005 && ratio <= 0.001) {
01023         gamma = 0.7;
01024     } else if (ratio > 0.0001 && ratio <= 0.0005) {
01025         gamma = 0.6;
01026     } else if (ratio > 0.00005 && ratio <= 0.0001) {
01027         gamma = 0.6;
01028     } else {
01029         gamma = 0.5;
01030     }
01031
01032     abip_float mix_ratio = error_ratio;
01033
01034     if (mix_ratio > 22) {
01035         gamma = gamma * 4.4;
01036     } else if (mix_ratio > 18 && mix_ratio <= 22) {
01037         gamma = gamma * 4.2;
01038     } else if (mix_ratio > 15 && mix_ratio <= 18) {
01039         gamma = gamma * 4;

```

```

01040     } else if (mix_ratio > 12 && mix_ratio <= 15) {
01041         gamma = gamma * 3.8;
01042     } else if (mix_ratio > 8 && mix_ratio <= 12) {
01043         gamma = gamma * 3.6;
01044     } else if (mix_ratio > 6 && mix_ratio <= 8) {
01045         sigma = 0.81;
01046         gamma = gamma * 3.4;
01047     } else if (mix_ratio > 4 && mix_ratio <= 6) {
01048         sigma = 0.82;
01049         gamma = gamma * 3.4;
01050     } else if (mix_ratio > 3 && mix_ratio <= 4) {
01051         sigma = 0.83;
01052         gamma = gamma * 3.2;
01053     } else if (mix_ratio > 3 && mix_ratio <= 4) {
01054         sigma = 0.84;
01055         gamma = gamma * 3;
01056     } else if (mix_ratio > 2 && mix_ratio <= 3) {
01057         sigma = 0.85;
01058         gamma = gamma * 2.8;
01059     } else if (mix_ratio > 1.5 && mix_ratio <= 2) {
01060         sigma = 0.85;
01061         gamma = gamma * 2.6;
01062     } else if (mix_ratio < 1.5) {
01063         sigma = 0.85;
01064         gamma = gamma * 2.4;
01065     }
01066
01067     sigma = sigma * 0.2;
01068
01069     w->mu = sigma * w->mu;
01070     return gamma * POWF(w->mu, spe->stgs->psi);
01071 }
01072
01073 abip_int ABIP(solve)(ABIPWork *w, const ABIPData *d, ABIPSolution *sol,
01074                     ABIPInfo *info, ABIPCone *c, spe_problem *s) {
01075     DEBUG_FUNC
01076
01077     abip_int i;
01078     abip_int j;
01079     abip_int k;
01080     ABIP(timer) solve_timer;
01081     ABIP(timer) lin_timer;
01082     ABIP(timer) barrier_timer;
01083     ABIP(timer) res_timer;
01084     ABIP(timer) P_timer;
01085     ABIP(timer) uw_timer;
01086     abip_float lin_time = 0;
01087     abip_float barrier_time = 0;
01088     abip_float res_time = 0;
01089     abip_float P_time = 0;
01090     abip_float uw_time = 0;
01091
01092     abip_float time_limit_left = 1e3 * s->stgs->time_limit - info->setup_time;
01093
01094     ABIPResiduals *r = (ABIPResiduals *)abip_malloc(sizeof(ABIPResiduals));
01095
01096     abip_int l = w->m + w->n + 1;
01097
01098     if (!d || !sol || !info || !w) {
01099         abip_printf("ERROR: ABIP_NULL input\n");
01100         RETURN ABIP_FAILED;
01101     }
01102
01103     abip_start_interrupt_listener();
01104     ABIP(tic)(&solve_timer);
01105
01106     info->status_val = ABIP_UNFINISHED;
01107     r->last_ipm_iter = -1;
01108     r->last_admm_iter = -1;
01109     r->res_pri = 1e8;
01110     r->res_dual = 1e8;
01111     r->rel_gap = 1e8;
01112     r->error_ratio = 1e8;
01113
01114     abip_float tol_inner = 4 * POWF(w->mu, s->stgs->psi);
01115
01116     ABIP(tic)(&uw_timer);
01117     update_work(d, w, sol, c, s);
01118     uw_time += ABIP(tocq)(&uw_timer) / 1e3;
01119
01120     if (s->stgs->verbose) {
01121         print_header(w);
01122     }
01123
01124     k = 0;
01125
01126     for (i = 0; i < s->stgs->max_ipm_iters; ++i) {

```

```

01130     for (j = 0; j < s->stgs->max_admm_iters; ++j) {
01131         ABIP(tic)(&lin_timer);
01132         if (projection(w, k, s, r) < 0) {
01133             RETURN failure(w, w->m, w->n, sol, info, ABIP_FAILED,
01134                 "error in project_lin_sys", "Failure");
01135         }
01136         lin_time += ABIP(tocq)(&lin_timer) / 1e3;
01137         ABIP(tic)(&barrier_timer);
01138         solve_barrier_subproblem(w, c, s);
01139         barrier_time += ABIP(tocq)(&barrier_timer) / 1e3;
01140
01141         update_dual_vars(w);
01142
01143         memcpy(w->v_origin, w->v, l * sizeof(abip_float));
01144         ABIP(c_dot)(w->v_origin, s->rho_dr, l);
01145
01146         k += 1;
01147
01148         ABIP(tic)(&P_timer);
01149
01150         abip_float err_inner = s->inner_conv_check(s, w);
01151
01152         if (err_inner < tol_inner || ABIP(tocq)(&solve_timer) > time_limit_left) {
01153             P_time += ABIP(tocq)(&P_timer) / 1e3;
01154
01155             break;
01156         }
01157         P_time += ABIP(tocq)(&P_timer) / 1e3;
01158
01159         if (abip_is_interrupted()) {
01160             RETURN failure(w, w->m, w->n, sol, info, ABIP_SIGINT, "Interrupted",
01161                 "Interrupted");
01162         }
01163
01164         #if EXTRA_VERBOSE > 0
01165         abip_printf("primal error: %.4f, dual error: %.4f, gap: %.4f\n",
01166             r.err_pri / w->eps_p, r.err_dual / w->eps_d,
01167             r.gap / w->eps_g);
01168         #endif
01169
01170         if ((j + 1) % s->stgs->inner_check_period == 0 || r->error_ratio <= 8) {
01171             ABIP(tic)(&res_timer);
01172             s->calc_residuals(s, w, r, i, k);
01173             res_time += ABIP(tocq)(&res_timer) / 1e3;
01174
01175             if ((j + 1) % s->stgs->inner_check_period == 0) {
01176                 print_summary(w, s, i, j, k, r, &solve_timer);
01177             }
01178
01179             if ((info->status_val = has_converged(w, s, r, i, k)) != 0 ||
01180                 k + 1 >= s->stgs->max_admm_iters * s->stgs->max_ipm_iters ||
01181                 i + 1 >= s->stgs->max_ipm_iters ||
01182                 ABIP(tocq)(&solve_timer) >
01183                     time_limit_left) // max running time is time_limit s
01184             {
01185                 if (s->stgs->verbose && k > 0) {
01186                     printf("\nin last admm iter:\n");
01187                     print_summary(w, s, i, j, k, r, &solve_timer);
01188                     abip_printf("total admm iter is %i\n", k);
01189                 }
01190
01191                 get_solution(w, s, sol, info, r, i, k);
01192                 info->solve_time = ABIP(tocq)(&solve_timer);
01193
01194                 if (s->stgs->verbose) {
01195                     print_footer(d, s, sol, w, info, k);
01196                     printf(
01197                         "\ntotal time of project_lin_sys: %.2es\ntotal time of "
01198                         "solve_barrier_subproblem: %.2es\ntotal time of calculate res: "
01199                         "%.2es\ntotal time of calculate err_inner: %.2es\ntotal time "
01200                         "of updating work: %.2es\n",
01201                         lin_time, barrier_time, res_time, P_time, uw_time);
01202                 }
01203
01204                 abip_end_interrupt_listener();
01205
01206                 RETURN info->status_val;
01207             }
01208         }
01209     } // inner for
01210
01211     if (s->sparsity || (i + 1) % s->stgs->outer_check_period == 0) {
01212         ABIP(tic)(&res_timer);
01213         s->calc_residuals(s, w, r, i, k);
01214         res_time += ABIP(tocq)(&res_timer) / 1e3;
01215         print_summary(w, s, i, j, k, r, &solve_timer);
01216     }

```

```

01217     if ((info->status_val = has_converged(w, s, r, i, k)) != 0 ||
01218         k + 1 >= s->stgs->max_admm_iters * s->stgs->max_ipm_iters ||
01219         i + 1 >= s->stgs->max_ipm_iters ||
01220         ABIP(tocq)(&solve_timer) > time_limit_left) {
01221     if (s->stgs->verbose && k > 0) {
01222         printf("\nin last admm iter:\n");
01223         print_summary(w, s, i, j, k, r, &solve_timer);
01224         abip_printf("total admm iter is %i\n", k);
01225     }
01226
01227     get_solution(w, s, sol, info, r, i, k);
01228     info->solve_time = ABIP(tocq)(&solve_timer);
01229
01230     if (s->stgs->verbose) {
01231         print_footer(d, s, sol, w, info, k);
01232         printf(
01233             "\ntotal time of project_lin_sys: %.2es\ntotal time of "
01234             "solve_barrier_subproblem: %.2es\ntotal time of calculate res: "
01235             "%.2es\ntotal time of calculate err_inner: %.2es\n",
01236             lin_time, barrier_time, res_time, P_time);
01237     }
01238
01239     abip_end_interrupt_listener();
01240
01241     RETURN info->status_val;
01242 }
01243 }
01244
01245     tol_inner = adjust_barrier(w, r, s);
01246 }
01247
01248     RETURN info->status_val;
01249 }
01250
01254 void ABIP(finish) (ABIPWork *w, spe_problem *spe) {
01255     DEBUG_FUNC
01256
01257     if (w) {
01258         free_work(w);
01259     }
01260
01261     if (spe->L) {
01262         spe->free_spe_linsys_work(spe);
01263     }
01264
01265     RETURN;
01266 }
01267
01271 ABIPWork *ABIP(init) (const ABIPData *d, ABIPInfo *info, spe_problem *s,
01272                        ABIPConc *c) {
01273     DEBUG_FUNC
01274
01275     #if EXTRA_VERBOSE > 1
01276         ABIP(tic)(&global_timer);
01277     #endif
01278
01279     ABIPWork *w;
01280     ABIP(timer) init_timer;
01281     abip_start_interrupt_listener();
01282
01283     if (!d || !info) {
01284         abip_printf("ERROR: Missing ABIPData or ABIPInfo input\n");
01285         RETURN ABIP_NULL;
01286     }
01287
01288     #if EXTRA_VERBOSE > 0
01289         ABIP(print_data) (d);
01290     #endif
01291
01292     #ifndef NOVALIDATE
01293     if (validate(d, c, s) < 0) {
01294         abip_printf("ERROR: Validation returned failure\n");
01295         RETURN ABIP_NULL;
01296     }
01297     #endif
01298
01299     ABIP(tic)(&init_timer);
01300
01301     w = init_work(s, c);
01302     info->setup_time = ABIP(tocq)(&init_timer);
01303
01304     if (d->stgs->verbose) {
01305         abip_printf("Setup time: %1.2es\n", info->setup_time / 1e3);
01306     }
01307
01308     abip_end_interrupt_listener();
01309

```

```

01310     RETURN w;
01311 }
01312
01316 abip_int ABIP(init_problem)(spe_problem **s, ABIPData *d, ABIPSettings *stgs,
01317                             enum problem_type special_problem) {
01318     switch (special_problem) {
01319         case LASSO:
01320             return init_lasso((lasso **)s, d, stgs);
01321         case SVM:
01322             return init_svm((svm **)s, d, stgs);
01323         case QCP:
01324             return init_qcp((qcp **)s, d, stgs);
01325         case SVMQP:
01326             return init_svmqp((svmqp **)s, d, stgs);
01327         default:
01328             return init_qcp((qcp **)s, d, stgs);
01329     }
01330 }
01331
01335 abip_int abip(const ABIPData *d, ABIPSolution *sol, ABIPInfo *info,
01336               ABIPCone *K) {
01337     DEBUG_FUNC
01338
01339     spe_problem *s = (spe_problem *)abip_malloc(sizeof(spe_problem));
01340     enum problem_type prob_type;
01341     if (d->stgs->prob_type == 0)
01342         prob_type = LASSO;
01343     else if (d->stgs->prob_type == 1)
01344         prob_type = SVM;
01345     else if (d->stgs->prob_type == 2)
01346         prob_type = QCP;
01347     else if (d->stgs->prob_type == 3)
01348         prob_type = SVMQP;
01349
01350     ABIP(init_problem)(&s, d, d->stgs, prob_type);
01351     abip_int status;
01352     ABIPWork *w = ABIP(init)(d, info, s,
01353                             K); // call init_work() to call init_lin_sys_work()
01354                                // to perform LDL' factorization
01355
01356     #if EXTRA_VERBOSE > 0
01357         abip_printf("size of abip_int = %lu, size of abip_float = %lu\n",
01358                     sizeof(abip_int), sizeof(abip_float));
01359     #endif
01360
01361     if (w) {
01362         ABIP(solve)(w, d, sol, info, K, s);
01363         status = info->status_val;
01364         ABIP(finish)(w, s);
01365     } else {
01366         status = failure(ABIP_NULL, d ? d->m : -1, d ? d->n : -1, sol, info,
01367                         ABIP_FAILED, "could not initialize work", "Failure");
01368     }
01369
01370     RETURN status;
01371 }

```

## 4.177 source/abip\_version.c File Reference

```
#include "glbopts.h"
```

### Functions

- const char \*ABIP() version (void)

#### 4.177.1 Function Documentation



## 4.177.1.1 version()

```
const char *ABIP() version (
    void )
```

Definition at line 3 of file [abip\\_version.c](#).

## 4.178 abip\_version.c

[Go to the documentation of this file.](#)

```
00001 #include "glbopts.h"
00002
00003 const char *ABIP(version) (void)
00004 {
00005     return ABIP_VERSION;
00006 }
```

## 4.179 source/cones.c File Reference

```
#include "cones.h"
```

## Macros

- `#define _CRT_SECURE_NO_WARNINGS`

## Functions

- `abip_int ABIP() get_cone_dims` (const `ABIPCone` \*k)  
*Calculate the total number of dimensions of all the cones.*
- `abip_int ABIP() validate_cones` (`spe_problem` \*spe, const `ABIPCone` \*k)  
*Check if the cone dimensions are valid.*
- `char *ABIP() get_cone_header` (const `ABIPCone` \*k)  
*Get the number of variables and blocks of each cone.*
- `void ABIP() soc_barrier_subproblem` (`abip_float` \*x, `abip_float` \*tmp, `abip_float` lambda, `abip_int` n)  
*Barrier subproblem for the second order cone.*
- `void ABIP() rsoc_barrier_subproblem` (`abip_float` \*x, `abip_float` \*tmp, `abip_float` lambda, `abip_int` n)  
*Barrier subproblem for the rotated second order cone.*
- `void ABIP() free_barrier_subproblem` (`abip_float` \*x, `abip_float` \*tmp, `abip_float` lambda, `abip_int` n)  
*Barrier subproblem for the free cone.*
- `void ABIP() zero_barrier_subproblem` (`abip_float` \*x, `abip_float` \*tmp, `abip_float` lambda, `abip_int` n)  
*Barrier subproblem for the zero cone.*
- `void ABIP() positive_orthant_barrier_subproblem` (`abip_float` \*x, `abip_float` \*tmp, `abip_float` lambda, `abip_int` n)  
*Barrier subproblem for the positive orthant cone.*

## 4.179.1 Macro Definition Documentation

#### 4.179.1.1 `_CRT_SECURE_NO_WARNINGS`

```
#define _CRT_SECURE_NO_WARNINGS
```

Definition at line 1 of file [cones.c](#).

### 4.179.2 Function Documentation

#### 4.179.2.1 `free_barrier_subproblem()`

```
void ABIP() free_barrier_subproblem (
    abip_float * x,
    abip_float * tmp,
    abip_float lambda,
    abip_int n )
```

Barrier subproblem for the free cone.

Definition at line 255 of file [cones.c](#).

#### 4.179.2.2 `get_cone_dims()`

```
abip_int ABIP() get_cone_dims (
    const ABIPConc * k )
```

Calculate the total number of dimensions of all the cones.

Definition at line 8 of file [cones.c](#).

#### 4.179.2.3 `get_cone_header()`

```
char *ABIP() get_cone_header (
    const ABIPConc * k )
```

Get the number of variables and blocks of each cone.

Definition at line 87 of file [cones.c](#).

#### 4.179.2.4 positive\_orthant\_barrier\_subproblem()

```
void ABIP() positive_orthant_barrier_subproblem (
    abip_float * x,
    abip_float * tmp,
    abip_float lambda,
    abip_int n )
```

Barrier subproblem for the positive orthant cone.

Definition at line 279 of file [cones.c](#).

#### 4.179.2.5 rsoc\_barrier\_subproblem()

```
void ABIP() rsoc_barrier_subproblem (
    abip_float * x,
    abip_float * tmp,
    abip_float lambda,
    abip_int n )
```

Barrier subproblem for the rotated second order cone.

Definition at line 169 of file [cones.c](#).

#### 4.179.2.6 soc\_barrier\_subproblem()

```
void ABIP() soc_barrier_subproblem (
    abip_float * x,
    abip_float * tmp,
    abip_float lambda,
    abip_int n )
```

Barrier subproblem for the second order cone.

Definition at line 130 of file [cones.c](#).

#### 4.179.2.7 validate\_cones()

```
abip_int ABIP() validate_cones (
    spe_problem * spe,
    const ABIPCones * k )
```

Check if the cone dimensions are valid.

Definition at line 37 of file [cones.c](#).

#### 4.179.2.8 zero\_barrier\_subproblem()

```
void ABIP() zero_barrier_subproblem (
    abip_float * x,
    abip_float * tmp,
    abip_float lambda,
    abip_int n )
```

Barrier subproblem for the zero cone.

Definition at line 267 of file [cones.c](#).

### 4.180 cones.c

[Go to the documentation of this file.](#)

```
00001 #define _CRT_SECURE_NO_WARNINGS
00002 #include "cones.h"
00003
00007 /* c = l + f + q[i] + s[i] */
00008 abip_int ABIP(get_cone_dims) (const ABIPCone* k) {
00009     abip_int i, c = 0;
00010
00011     if (k->l) {
00012         c += k->l;
00013     }
00014     if (k->z) {
00015         c += k->z;
00016     }
00017     if (k->f) {
00018         c += k->f;
00019     }
00020     if (k->qsize && k->q) {
00021         for (i = 0; i < k->qsize; ++i) {
00022             c += k->q[i];
00023         }
00024     }
00025     if (k->rsize && k->r) {
00026         for (i = 0; i < k->rsize; ++i) {
00027             c += k->r[i];
00028         }
00029     }
00030
00031     return c;
00032 }
00033
00037 abip_int ABIP(validate_cones) (spe_problem* spe, const ABIPCone* k) {
00038     abip_int i;
00039     if (ABIP(get_cone_dims) (k) != spe->q) {
00040         abip_printf("cone dimensions %li not equal to num rows in A = n = %li\n",
00041             (long)ABIP(get_cone_dims) (k), (long)spe->q);
00042         return -1;
00043     }
00044     if (k->l && k->l < 0) {
00045         abip_printf("lp cone error\n");
00046         return -1;
00047     }
00048     if (k->f && k->f < 0) {
00049         abip_printf("free cone error\n");
00050         return -1;
00051     }
00052     if (k->z && k->z < 0) {
00053         abip_printf("zero cone error\n");
00054         return -1;
00055     }
00056     if (k->qsize && k->q) {
00057         if (k->qsize < 0) {
00058             abip_printf("soc cone error\n");
00059             return -1;
00060         }
00061         for (i = 0; i < k->qsize; ++i) {
00062             if (k->q[i] < 0) {
00063                 abip_printf("soc cone error\n");
00064                 return -1;
00065             }
00066         }
00067     }
```

```

00067     }
00068     if (k->rsize && k->rq) {
00069         if (k->rsize < 0) {
00070             abip_printf("rsoc cone error\n");
00071             return -1;
00072         }
00073         for (i = 0; i < k->rsize; ++i) {
00074             if (k->rq[i] < 0) {
00075                 abip_printf("rsoc cone error\n");
00076                 return -1;
00077             }
00078         }
00079     }
00080     return 0;
00081 }
00082
00083 char* ABIP(get_cone_header)(const ABIPCone* k) {
00084     abip_int i, soc_vars, rsoc_vars;
00085     char* tmp = (char*)abip_malloc(sizeof(char) * 512);
00086     sprintf(tmp, "Cones:");
00087
00088     soc_vars = 0;
00089     if (k->qsize && k->q) {
00090         for (i = 0; i < k->qsize; i++) {
00091             soc_vars += k->q[i];
00092         }
00093         sprintf(tmp + strlen(tmp), "\tsoc vars: %li, soc blks: %li\n",
00094             (long)soc_vars, (long)k->qsize);
00095     }
00096
00097     rsoc_vars = 0;
00098     if (k->rsize && k->rq) {
00099         for (i = 0; i < k->rsize; i++) {
00100             rsoc_vars += k->rq[i];
00101         }
00102         sprintf(tmp + strlen(tmp), "\trsoc vars: %li, rsoc blks: %li\n",
00103             (long)rsoc_vars, (long)k->rsize);
00104     }
00105
00106     if (k->f) {
00107         sprintf(tmp + strlen(tmp), "\tfree vars: %li\n", (long)k->f);
00108     }
00109
00110     if (k->z) {
00111         sprintf(tmp + strlen(tmp), "\tzero vars: %li\n", (long)k->z);
00112     }
00113
00114     if (k->l) {
00115         sprintf(tmp + strlen(tmp), "\tlinear vars: %li\n", (long)k->l);
00116     }
00117
00118     return tmp;
00119 }
00120
00121 /* x in second order cone
00122    K = {(t,x) | t>||x||}
00123 */
00124 void ABIP(soc_barrier_subproblem)(abip_float* x, abip_float* tmp,
00125     abip_float lambda, abip_int n) {
00126     abip_float a = tmp[0];
00127     abip_float tol = 1e-9;
00128     abip_float* b = (abip_float*)abip_malloc((n - 1) * sizeof(abip_float));
00129     memcpy(b, &tmp[1], (n - 1) * sizeof(abip_float));
00130     abip_float b_norm_sq = ABIP(norm_sq)(b, n - 1);
00131     if (ABS(a) <= tol) {
00132         x[0] = SQRTF(2 * lambda + b_norm_sq / 4);
00133         memcpy(&x[1], b, (n - 1) * sizeof(abip_float));
00134         ABIP(scale_array)(&x[1], 0.5, (n - 1));
00135     } else {
00136         abip_float coef1 = -(POWF(a, 2) - ABIP(norm_sq)(b, n - 1)) / lambda;
00137         abip_float coef2 =
00138             -(4 * POWF(a, 2) + 4 * ABIP(norm_sq)(b, n - 1) + 16 * lambda) / lambda;
00139
00140         abip_float r = 16 * a * a /
00141             (8 * lambda - a * a + b_norm_sq +
00142             SQRTF(POWF((8 * lambda - a * a + b_norm_sq), 2) +
00143             32 * a * a * lambda));
00144         abip_float s1 = (r - SQRTF(r * (r + 8))) / 2;
00145         abip_float s2 = (r + SQRTF(r * (r + 8))) / 2;
00146
00147         abip_float s = a > 0 ? s2 : s1;
00148         abip_float eta = (s + 2) * a / s;
00149         ABIP(scale_array)(b, (s + 2) / (s + 4), n - 1);
00150
00151         x[0] = eta;
00152         memcpy(&x[1], b, (n - 1) * sizeof(abip_float));
00153     }

```

```

00160     abip_free(b);
00161 }
00162
00166 /* K = {(t1,t2,x) | 2*t1*t2>||x||}
00167     n = len(t1,t2,x)
00168 */
00169 void ABIP(rsoc_barrier_subproblem)(abip_float* x, abip_float* tmp,
00170                                   abip_float lambda, abip_int n) {
00171     abip_int nx = n - 2;
00172     abip_float tol = 1e-9;
00173
00174     abip_float zeta_eta = tmp[0];
00175     abip_float zeta_nu = tmp[1];
00176
00177     abip_float* zeta_x = (abip_float*)abip_malloc(nx * sizeof(abip_float));
00178     memcpy(zeta_x, &tmp[2], nx * sizeof(abip_float));
00179     abip_float zeta_x_norm_sq = ABIP(norm_sq)(zeta_x, nx);
00180
00181     if (zeta_eta + zeta_nu == 0) {
00182         x[1] =
00183             (-zeta_eta + SQRTF(zeta_eta * zeta_eta + 4 * lambda + zeta_x_norm_sq)) /
00184             2;
00185         x[0] = x[0] + zeta_eta;
00186         memcpy(&x[2], zeta_x, nx * sizeof(abip_float));
00187         ABIP(scale_array)(&x[2], 0.5, nx);
00188     } else {
00189         abip_float w;
00190         abip_float s;
00191         if (2 * zeta_eta * zeta_nu - zeta_x_norm_sq < 0) {
00192             w = (2 * POWF(zeta_eta + zeta_nu, 2) / lambda) /
00193                 (-2 * zeta_eta * zeta_nu - zeta_x_norm_sq) / (2 * lambda)) /
00194                 (1 + 4 / (-2 * zeta_eta * zeta_nu - zeta_x_norm_sq) / (2 * lambda)) +
00195                 SQRTF(
00196                     1 +
00197                     (4 * (zeta_eta * zeta_eta + zeta_nu * zeta_nu + zeta_x_norm_sq) /
00198                      lambda +
00199                     16) /
00200                     (-2 * zeta_eta * zeta_nu - zeta_x_norm_sq) / (2 * lambda)) /
00201                     (-2 * zeta_eta * zeta_nu - zeta_x_norm_sq) /
00202                     (2 * lambda)))));
00203         } else {
00204             w = (2 * zeta_eta * zeta_nu - zeta_x_norm_sq) / (2 * lambda) *
00205                 (1 - 4 / ((2 * zeta_eta * zeta_nu - zeta_x_norm_sq) / (2 * lambda)) +
00206                 SQRTF(
00207                     1 +
00208                     (4 * (zeta_eta * zeta_eta + zeta_nu * zeta_nu + zeta_x_norm_sq) /
00209                      lambda +
00210                     16) /
00211                     ((2 * zeta_eta * zeta_nu - zeta_x_norm_sq) / (2 * lambda)) /
00212                     ((2 * zeta_eta * zeta_nu - zeta_x_norm_sq) /
00213                     (2 * lambda)))))) /
00214                 2;
00215         }
00216         if (zeta_eta + zeta_nu > 0) {
00217             s = (w + SQRTF(w * (w + 4))) / 2;
00218             x[0] = (zeta_eta * POWF(s + 1, 2) + zeta_nu * (s + 1)) / (s * (s + 2));
00219             x[1] = (zeta_nu * POWF(s + 1, 2) + zeta_eta * (s + 1)) / (s * (s + 2));
00220             memcpy(&x[2], zeta_x, nx * sizeof(abip_float));
00221             ABIP(scale_array)(&x[2], (s + 1) / (s + 2), nx);
00222         } else {
00223             if (w > 10) {
00224                 s = 2 / (w + 2 + SQRTF(w * (w + 4)));
00225                 x[0] = (zeta_eta * POWF(s, 2) + zeta_nu * s) / ((s - 1) * (s + 1));
00226                 x[1] = (zeta_nu * POWF(s, 2) + zeta_eta * s) / ((s - 1) * (s + 1));
00227                 memcpy(&x[2], zeta_x, nx * sizeof(abip_float));
00228                 ABIP(scale_array)(&x[2], s / (s + 1), nx);
00229             } else {
00230                 s = (w - SQRTF(w * (w + 4))) / 2;
00231                 x[0] = (zeta_eta * POWF(s + 1, 2) + zeta_nu * (s + 1)) / (s * (s + 2));
00232                 x[1] = (zeta_nu * POWF(s + 1, 2) + zeta_eta * (s + 1)) / (s * (s + 2));
00233                 memcpy(&x[2], zeta_x, nx * sizeof(abip_float));
00234                 ABIP(scale_array)(&x[2], (s + 1) / (s + 2), nx);
00235             }
00236         }
00237     }
00238
00239     for (int i = nx + 2; i < n; i++) {
00240         if (tmp[i] >= 0) {
00241             x[i] = (tmp[i] + SQRTF(tmp[i] * tmp[i] + 4 * lambda)) / 2;
00242         } else {
00243             x[i] = 2 * lambda /
00244                 (-tmp[i] * (1 + SQRTF(1 + 4 * lambda / POWF(tmp[i], 2))));
00245         }
00246     }
00247     abip_free(zeta_x);
00248 }
00249

```

```

00253 /* K = {x | x free}
00254 */
00255 void ABIP(free_barrier_subproblem)(abip_float* x, abip_float* tmp,
00256                                  abip_float lambda, abip_int n) {
00257     for (int i = 0; i < n; i++) {
00258         x[i] = tmp[i];
00259     }
00260 }
00261
00265 /* K = {x | x = 0}
00266 */
00267 void ABIP(zero_barrier_subproblem)(abip_float* x, abip_float* tmp,
00268                                   abip_float lambda, abip_int n) {
00269     for (int i = 0; i < n; i++) {
00270         x[i] = 0;
00271     }
00272 }
00273
00277 /* K = {x | x >= 0}
00278 */
00279 void ABIP(positive_orthant_barrier_subproblem)(abip_float* x, abip_float* tmp,
00280                                                abip_float lambda, abip_int n) {
00281     for (int i = 0; i < n; i++) {
00282         if (tmp[i] >= 0) {
00283             x[i] = (tmp[i] + SQRTF(tmp[i] * tmp[i] + 4 * lambda)) / 2;
00284         } else {
00285             x[i] = 2 * lambda /
00286                 (-tmp[i] * (1 + SQRTF(1 + 4 * lambda / POWF(tmp[i], 2))));
00287         }
00288     }
00289 }

```

## 4.181 source/ctrlc.c File Reference

```
#include "ctrlc.h"
```

## 4.182 ctrlc.c

[Go to the documentation of this file.](#)

```

00001 /*
00002  * Implements signal handling (ctrl-c) for ABIP.
00003  *
00004  * Under Windows, we use SetConsoleCtrlHandler.
00005  * Under Unix systems, we use sigaction.
00006  * For Mex files, we use utSetInterruptEnabled/utIsInterruptPending.
00007  *
00008  */
00009
00010 #include "ctrlc.h"
00011
00012 #if CTRLC > 0
00013
00014 #ifndef MATLAB_MEX_FILE
00015
00016 static int ystate;
00017 void abip_start_interrupt_listener(void)
00018 {
00019     ystate = utSetInterruptEnabled(1);
00020 }
00021
00022 void abip_end_interrupt_listener(void)
00023 {
00024     utSetInterruptEnabled(ystate);
00025 }
00026
00027 int abip_is_interrupted(void)
00028 {
00029     return utIsInterruptPending();
00030 }
00031
00032 #elif defined _WIN32 || _WIN64 || defined _WINDLL
00033
00034 static int int_detected;
00035 static BOOL WINAPI abip_handle_ctrlc(DWORD dwCtrlType)

```

```

00036 {
00037     if (dwCtrlType != CTRL_C_EVENT)
00038     {
00039         return FALSE;
00040     }
00041
00042     int_detected = 1;
00043     return TRUE;
00044 }
00045
00046 void abip_start_interrupt_listener(void)
00047 {
00048     int_detected = 0;
00049     SetConsoleCtrlHandler(abip_handle_ctrlc, TRUE);
00050 }
00051
00052 void abip_end_interrupt_listener(void)
00053 {
00054     SetConsoleCtrlHandler(abip_handle_ctrlc, FALSE);
00055 }
00056
00057 int abip_is_interrupted(void)
00058 {
00059     return int_detected;
00060 }
00061
00062 #else /* Unix */
00063
00064 #include <signal.h>
00065 static int int_detected;
00066 struct sigaction oact;
00067 static void abip_handle_ctrlc(int dummy)
00068 {
00069     int_detected = dummy ? dummy : -1;
00070 }
00071
00072 void abip_start_interrupt_listener(void)
00073 {
00074     struct sigaction act;
00075     int_detected = 0;
00076
00077     act.sa_flags = 0;
00078     sigemptyset(&act.sa_mask);
00079
00080     act.sa_handler = abip_handle_ctrlc;
00081     sigaction(SIGINT, &act, &oact);
00082 }
00083
00084 void abip_end_interrupt_listener(void)
00085 {
00086     struct sigaction act;
00087     sigaction(SIGINT, &oact, &act);
00088 }
00089
00090 int abip_is_interrupted(void)
00091 {
00092     return int_detected;
00093 }
00094
00095 #endif /* END IF MATLAB_MEX_FILE / WIN32 */
00096
00097 #endif /* END IF CTRLC > 0 */

```

## 4.183 source/lasso\_config.c File Reference

```
#include "lasso_config.h"
```

### Macros

- #define [MIN\\_SCALE](#) (1e-3)
- #define [MAX\\_SCALE](#) (1e3)



## Functions

- [abip\\_int init\\_lasso](#) ([lasso](#) \*\*self, [ABIPData](#) \*d, [ABIPSettings](#) \*stgs)  
*Initialize the lasso problem structure.*
- void [lasso\\_A\\_times](#) ([lasso](#) \*self, const [abip\\_float](#) \*x, [abip\\_float](#) \*y)  
*Customized matrix-vector multiplication for the lasso problem with A untransposed.*
- void [lasso\\_AT\\_times](#) ([lasso](#) \*self, const [abip\\_float](#) \*x, [abip\\_float](#) \*y)  
*Customized matrix-vector multiplication for the lasso problem with A transposed.*
- void [scaling\\_lasso\\_data](#) ([lasso](#) \*self, [ABIPConc](#) \*k)  
*Customized scaling procedure for the lasso problem.*
- void [get\\_unscaled\\_x](#) ([lasso](#) \*self, [abip\\_float](#) \*x, [abip\\_float](#) \*us\_x)  
*Get the unscaled solution x of the lasso qcp problem.*
- void [get\\_unscaled\\_y](#) ([lasso](#) \*self, [abip\\_float](#) \*y, [abip\\_float](#) \*us\_y)  
*Get the unscaled solution y of the lasso qcp problem.*
- void [get\\_unscaled\\_s](#) ([lasso](#) \*self, [abip\\_float](#) \*s, [abip\\_float](#) \*us\_s)  
*Get the unscaled solution s of the lasso qcp problem.*
- void [un\\_scaling\\_lasso\\_sol](#) ([lasso](#) \*self, [ABIPSolution](#) \*sol)  
*Get the unscaled solution of the original lasso problem.*
- [abip\\_float](#) [lasso\\_inner\\_conv\\_check](#) ([lasso](#) \*self, [ABIPWork](#) \*w)  
*Check whether the inner loop of the lasso problem has converged.*
- void [calc\\_lasso\\_residuals](#) ([lasso](#) \*self, [ABIPWork](#) \*w, [ABIPResiduals](#) \*r, [abip\\_int](#) ipm\_iter, [abip\\_int](#) admm↔\_iter)  
*Calculate the residuals of the lasso qcp problem.*
- [cs](#) \* [form\\_lasso\\_kkt](#) ([lasso](#) \*self)  
*Formulate the qcp KKT matrix of the lasso problem.*
- void [init\\_lasso\\_precon](#) ([lasso](#) \*self)  
*Initialize the preconditioner of conjugate gradient method for the lasso problem.*
- [abip\\_float](#) [get\\_lasso\\_pcg\\_tol](#) ([abip\\_int](#) k, [abip\\_float](#) error\_ratio, [abip\\_float](#) norm\_p)  
*Get the tolerance of the conjugate gradient method for the lasso problem.*
- [abip\\_int](#) [init\\_lasso\\_linsys\\_work](#) ([lasso](#) \*self)  
*Initialize the linear system solver work space for the lasso problem.*
- [abip\\_int](#) [solve\\_lasso\\_linsys](#) ([lasso](#) \*self, [abip\\_float](#) \*b, [abip\\_float](#) \*pcg\_warm\_start, [abip\\_int](#) iter, [abip\\_float](#) error\_ratio)  
*Customized linear system solver for the lasso problem.*
- void [free\\_lasso\\_linsys\\_work](#) ([lasso](#) \*self)  
*Free the linear system solver work space for the lasso problem.*

### 4.183.1 Macro Definition Documentation

#### 4.183.1.1 MAX\_SCALE

```
#define MAX_SCALE (1e3)
```

Definition at line 3 of file [lasso\\_config.c](#).

#### 4.183.1.2 MIN\_SCALE

```
#define MIN_SCALE (1e-3)
```

Definition at line 2 of file [lasso\\_config.c](#).

### 4.183.2 Function Documentation

#### 4.183.2.1 calc\_lasso\_residuals()

```
void calc_lasso_residuals (  
    lasso * self,  
    ABIPWork * w,  
    ABIPResiduals * r,  
    abip\_int ipm_iter,  
    abip\_int admm_iter )
```

Calculate the residuals of the lasso qcp problem.

Definition at line 367 of file [lasso\\_config.c](#).

#### 4.183.2.2 form\_lasso\_kkt()

```
cs * form_lasso_kkt (  
    lasso * self )
```

Formulate the qcp KKT matrix of the lasso problem.

Definition at line 507 of file [lasso\\_config.c](#).

#### 4.183.2.3 free\_lasso\_linsys\_work()

```
void free_lasso_linsys_work (  
    lasso * self )
```

Free the linear system solver work space for the lasso problem.

Definition at line 722 of file [lasso\\_config.c](#).

#### 4.183.2.4 `get_lasso_pcg_tol()`

```
abip_float get_lasso_pcg_tol (
    abip_int k,
    abip_float error_ratio,
    abip_float norm_p )
```

Get the tolerance of the conjugate gradient method for the lasso problem.

Definition at line 592 of file [lasso\\_config.c](#).

#### 4.183.2.5 `get_unscaled_s()`

```
void get_unscaled_s (
    lasso * self,
    abip_float * s,
    abip_float * us_s )
```

Get the unscaled solution s of the lasso qcp problem.

Definition at line 289 of file [lasso\\_config.c](#).

#### 4.183.2.6 `get_unscaled_x()`

```
void get_unscaled_x (
    lasso * self,
    abip_float * x,
    abip_float * us_x )
```

Get the unscaled solution x of the lasso qcp problem.

Definition at line 265 of file [lasso\\_config.c](#).

#### 4.183.2.7 `get_unscaled_y()`

```
void get_unscaled_y (
    lasso * self,
    abip_float * y,
    abip_float * us_y )
```

Get the unscaled solution y of the lasso qcp problem.

Definition at line 279 of file [lasso\\_config.c](#).

#### 4.183.2.8 `init_lasso()`

```
abip_int init_lasso (
    lasso ** self,
    ABIPData * d,
    ABIPSettings * stgs )
```

Initialize the lasso problem structure.

Definition at line 8 of file [lasso\\_config.c](#).

#### 4.183.2.9 `init_lasso_linsys_work()`

```
abip_int init_lasso_linsys_work (
    lasso * self )
```

Initialize the linear system solver work space for the lasso problem.

Definition at line 624 of file [lasso\\_config.c](#).

#### 4.183.2.10 `init_lasso_precon()`

```
void init_lasso_precon (
    lasso * self )
```

Initialize the preconditioner of conjugate gradient method for the lasso problem.

Definition at line 571 of file [lasso\\_config.c](#).

#### 4.183.2.11 `lasso_A_times()`

```
void lasso_A_times (
    lasso * self,
    const abip_float * x,
    abip_float * y )
```

Customized matrix-vector multiplication for the lasso problem with A untransposed.

Definition at line 99 of file [lasso\\_config.c](#).

#### 4.183.2.12 lasso\_AT\_times()

```
void lasso_AT_times (
    lasso * self,
    const abip_float * x,
    abip_float * y )
```

Customized matrix-vector multiplication for the lasso problem with A transposed.

Definition at line 116 of file [lasso\\_config.c](#).

#### 4.183.2.13 lasso\_inner\_conv\_check()

```
abip_float lasso_inner_conv_check (
    lasso * self,
    ABIPWork * w )
```

Check whether the inner loop of the lasso problem has converged.

Definition at line 323 of file [lasso\\_config.c](#).

#### 4.183.2.14 scaling\_lasso\_data()

```
void scaling_lasso_data (
    lasso * self,
    ABIPCone * k )
```

Customized scaling procedure for the lasso problem.

Definition at line 131 of file [lasso\\_config.c](#).

#### 4.183.2.15 solve\_lasso\_linsys()

```
abip_int solve_lasso_linsys (
    lasso * self,
    abip_float * b,
    abip_float * pcg_warm_start,
    abip_int iter,
    abip_float error_ratio )
```

Customized linear system solver for the lasso problem.

Definition at line 648 of file [lasso\\_config.c](#).

#### 4.183.2.16 un\_scaling\_lasso\_sol()

```
void un_scaling_lasso_sol (
    lasso * self,
    ABIPSolution * sol )
```

Get the unscaled solution of the original lasso problem.

Definition at line 303 of file [lasso\\_config.c](#).

### 4.184 lasso\_config.c

[Go to the documentation of this file.](#)

```
00001 #include "lasso_config.h"
00002 #define MIN_SCALE (1e-3)
00003 #define MAX_SCALE (1e3)
00004
00008 abip_int init_lasso(lasso **self, ABIPData *d, ABIPSettings *stgs) {
00009     lasso *this_lasso = (lasso *)abip_malloc(sizeof(lasso));
00010     *self = this_lasso;
00011
00012     this_lasso->m = d->m;
00013     this_lasso->n = d->n;
00014     this_lasso->p = d->m + 1;
00015     this_lasso->q = 2 + 2 * d->n + d->m;
00016     abip_int m = this_lasso->p;
00017     abip_int n = this_lasso->q;
00018
00019     this_lasso->lambda = d->lambda;
00020     this_lasso->q = ABIP_NULL;
00021     this_lasso->sparsity = (((abip_float)d->A->p[d->n] / (d->m * d->n)) < 0.1);
00022
00023     // non-identity DR scaling
00024     this_lasso->rho_dr =
00025         (abip_float *)abip_malloc((m + n + 1) * sizeof(abip_float));
00026     for (int i = 0; i < m + n + 1; i++) {
00027         if (i < m) {
00028             this_lasso->rho_dr[i] = stgs->rho_y;
00029         } else if (i < m + n) {
00030             this_lasso->rho_dr[i] = stgs->rho_x;
00031         } else {
00032             this_lasso->rho_dr[i] = stgs->rho_tau;
00033         }
00034     }
00035
00036     if (this_lasso->sparsity) {
00037         this_lasso->sc = 2;
00038         this_lasso->sc_c = 1 / d->lambda;
00039         this_lasso->sc_cone2 = d->lambda / d->m * 80;
00040         this_lasso->sc_cone1 = 0.8 / this_lasso->sc_c / this_lasso->sc_cone2;
00041         this_lasso->sc_b = this_lasso->sc_c * 300 * d->lambda / d->m;
00042     } else {
00043         if (d->m < d->n)
00044             this_lasso->sc = 4;
00045         else
00046             this_lasso->sc = 1;
00047         this_lasso->sc_c = 1 / d->lambda;
00048         this_lasso->sc_b = this_lasso->sc_c;
00049         this_lasso->sc_cone2 = 0.8;
00050         this_lasso->sc_cone1 = 1 / this_lasso->sc_c;
00051     }
00052
00053     this_lasso->L = (ABIPLinSysWork *)abip_malloc(sizeof(ABIPLinSysWork));
00054     this_lasso->pro_type = LASSO;
00055     this_lasso->stgs = stgs;
00056     this_lasso->data = d;
00057
00058     abip_float *data_b =
00059         (abip_float *)abip_malloc(this_lasso->p * sizeof(abip_float));
00060     data_b[0] = 1;
00061     memcpy(&data_b[1], d->b, d->m * sizeof(abip_float));
00062     this_lasso->data->b = data_b;
00063
00064     abip_float *data_c =
00065         (abip_float *)abip_malloc(this_lasso->q * sizeof(abip_float));
00066     data_c[0] = 0;
```

```

00067     data_c[1] = 2;
00068     memset(&data_c[2], 0, d->m * sizeof(abip_float));
00069     for (int i = 0; i < 2 * d->n; i++) {
00070         data_c[i + 2 + d->m] = d->lambdas;
00071     }
00072     this_lasso->data->c = data_c;
00073     /* for lasso problem, no need for inputting c*/
00074     d->c = data_c;
00075
00076     this_lasso->A = (ABIPMatrix *)abip_malloc(sizeof(ABIPMatrix));
00077     this_lasso->b = (abip_float *)abip_malloc(this_lasso->p * sizeof(abip_float));
00078     this_lasso->c = (abip_float *)abip_malloc(this_lasso->q * sizeof(abip_float));
00079     this_lasso->D = (abip_float *)abip_malloc(this_lasso->m * sizeof(abip_float));
00080     this_lasso->E = (abip_float *)abip_malloc(this_lasso->n * sizeof(abip_float));
00081
00082     this_lasso->scaling_data = &scaling_lasso_data;
00083     this_lasso->un_scaling_sol = &un_scaling_lasso_sol;
00084     this_lasso->calc_residuals = &calc_lasso_residuals;
00085     this_lasso->init_spe_linsys_work = &init_lasso_linsys_work;
00086     this_lasso->solve_spe_linsys = &solve_lasso_linsys;
00087     this_lasso->free_spe_linsys_work = &free_lasso_linsys_work;
00088     this_lasso->spe_A_times = &lasso_A_times;
00089     this_lasso->spe_AT_times = &lasso_AT_times;
00090     this_lasso->inner_conv_check = &lasso_inner_conv_check;
00091
00092     return 0;
00093 }
00094
00099 void lasso_A_times(lasso *self, const abip_float *x, abip_float *y) {
00100     y[0] += x[0];
00101
00102     for (int i = 1; i < self->p; i++) {
00103         y[i] += self->D[i - 1] * SQRTF(self->sc_cone2) * x[i + 1];
00104     }
00105
00106     ABIP(accum_by_A)(self->A, &x[self->m + 2], &y[1]);
00107     ABIP(scale_array)(&y[1], -1, self->m);
00108     ABIP(accum_by_A)(self->A, &x[self->m + self->n + 2], &y[1]);
00109     ABIP(scale_array)(&y[1], -1, self->m);
00110 }
00111
00116 void lasso_AT_times(lasso *self, const abip_float *x, abip_float *y) {
00117     y[0] += x[0];
00118     for (int i = 2; i < self->m + 2; i++) {
00119         y[i] += x[i - 1] * self->D[i - 2] * SQRTF(self->sc_cone2);
00120     }
00121
00122     ABIP(accum_by_Atrans)(self->A, &x[1], &y[self->m + 2]);
00123     ABIP(scale_array)(&y[self->m + 2 + self->n], -1, self->n);
00124     ABIP(accum_by_Atrans)(self->A, &x[1], &y[self->m + 2 + self->n]);
00125     ABIP(scale_array)(&y[self->m + 2 + self->n], -1, self->n);
00126 }
00127
00131 void scaling_lasso_data(lasso *self, ABIPConc *k) {
00132     if (!ABIP(copy_A_matrix)(&(self->A), self->data->A)) {
00133         abip_printf("ERROR: copy A matrix failed\n");
00134         RETURN;
00135     }
00136
00137     abip_int m = self->m;
00138     abip_int n = self->n;
00139     ABIPMatrix *A = self->A;
00140
00141     abip_float min_row_scale = MIN_SCALE * SQRTF((abip_float)n);
00142     abip_float max_row_scale = MAX_SCALE * SQRTF((abip_float)n);
00143     abip_float min_col_scale = MIN_SCALE * SQRTF((abip_float)m);
00144     abip_float max_col_scale = MAX_SCALE * SQRTF((abip_float)m);
00145
00146     abip_float *E = self->E;
00147     memset(E, 0, n * sizeof(abip_float));
00148
00149     abip_float *D = self->D;
00150     memset(D, 0, m * sizeof(abip_float));
00151
00152     abip_float avg = 0;
00153     abip_float avg1 = 0;
00154
00155     if (self->stgs->scale_E) {
00156         if (self->sparsity) {
00157             for (int i = 0; i < n; i++) {
00158                 for (int j = A->p[i]; j < A->p[i + 1]; j++) {
00159                     E[i] += A->x[j] * A->x[j];
00160                 }
00161             }
00162             avg += SQRTF(E[i]);
00163         }
00164     }

```

```

00165     avg /= n;
00166
00167     for (int i = 0; i < n; i++) {
00168         E[i] = avg / SQRTF(E[i] + 1e-4) / self->sc;
00169         if (E[i] > 1000 * SQRTF(m)) {
00170             E[i] = 1000 * SQRTF(m);
00171         }
00172         if (E[i] < 0.001 * SQRTF(m)) {
00173             E[i] = 1;
00174         }
00175         if (E[i] > 50) {
00176             E[i] = 50;
00177         }
00178
00179         avg1 += E[i];
00180     }
00181
00182     avg1 /= n;
00183
00184     for (int i = 0; i < n; i++) {
00185         E[i] = avg1 / E[i] / self->sc;
00186     }
00187 } else {
00188     for (int i = 0; i < n; i++) {
00189         for (int j = A->p[i]; j < A->p[i + 1]; j++) {
00190             E[i] += A->x[j] * A->x[j];
00191         }
00192
00193         E[i] = SQRTF(E[i]);
00194
00195         if (E[i] > 1000 * SQRTF(m)) {
00196             E[i] = 1000 * SQRTF(m);
00197         }
00198         if (E[i] < 0.001 * SQRTF(m)) {
00199             E[i] = 1;
00200         }
00201         if (E[i] > 7) {
00202             E[i] = 7;
00203         }
00204
00205         E[i] = 1 / (E[i] * self->sc);
00206     }
00207 }
00208
00209     for (int i = 0; i < n; i++) {
00210         for (int j = A->p[i]; j < A->p[i + 1]; j++) {
00211             A->x[j] *= E[i];
00212         }
00213     }
00214 }
00215
00216     for (int i = 0; i < A->p[n]; i++) {
00217         D[A->i[i]] += A->x[i] * A->x[i];
00218     }
00219
00220     avg = 0;
00221     for (int i = 0; i < m; i++) {
00222         avg += SQRTF(2 * D[i] + self->sc_cone2);
00223     }
00224
00225     avg /= m;
00226
00227     for (int i = 0; i < m; i++) {
00228         D[i] = avg / SQRTF(2 * D[i] + self->sc_cone2);
00229     }
00230
00231     for (int i = 0; i < A->p[n]; i++) {
00232         A->x[i] *= D[A->i[i]];
00233     }
00234
00235     memcpy(self->b, self->data->b, self->p * sizeof(abip_float));
00236
00237     self->b[0] = self->sc_cone1;
00238
00239     for (int i = 1; i < m + 1; i++) {
00240         self->b[i] *= D[i - 1];
00241     }
00242     ABIP(scale_array)(self->b, self->sc_b, self->p);
00243
00244     self->c[0] = 0;
00245     self->c[1] = self->sc_cone1 * self->sc_cone2;
00246     memset(&self->c[2], 0, self->m * sizeof(abip_float));
00247
00248     for (int i = 0; i < self->n; i++) {
00249         self->c[i + self->m + 2] = E[i] * self->lambda;
00250         self->c[i + self->m + 2 + self->n] = E[i] * self->lambda;
00251     }

```



```

00252     ABIP(scale_array)(self->c, self->sc_c, self->q);
00253
00254     self->D_hat = (abip_float *)abip_malloc(m * sizeof(abip_float));
00255
00256     for (int i = 0; i < m; i++) {
00257         self->D_hat[i] =
00258             self->D[i] * self->D[i] * self->sc_cone2 + self->stgs->rho_y;
00259     }
00260 }
00261
00265 void get_unscaled_x(lasso *self, abip_float *x, abip_float *us_x) {
00266     memcpy(us_x, x, self->q * sizeof(abip_float));
00267     for (int i = 0; i < self->m; i++) {
00268         us_x[i + 2] /= (self->sc_b / SQRTF(self->sc_cone2));
00269     }
00270     for (int i = 0; i < self->n; i++) {
00271         us_x[self->m + 2 + i] /= (self->E[i] * self->sc_b);
00272         us_x[self->m + self->n + 2 + i] /= (self->E[i] * self->sc_b);
00273     }
00274 }
00275
00279 void get_unscaled_y(lasso *self, abip_float *y, abip_float *us_y) {
00280     memcpy(us_y, y, self->p * sizeof(abip_float));
00281     for (int i = 0; i < self->m; i++) {
00282         us_y[i + 1] *= (self->D[i] / self->sc_c);
00283     }
00284 }
00285
00289 void get_unscaled_s(lasso *self, abip_float *s, abip_float *us_s) {
00290     memcpy(us_s, s, self->q * sizeof(abip_float));
00291     for (int i = 0; i < self->m; i++) {
00292         us_s[i + 2] /= (self->sc_c / SQRTF(self->sc_cone2));
00293     }
00294     for (int i = 0; i < self->n; i++) {
00295         us_s[self->m + 2 + i] /= (self->E[i] * self->sc_c);
00296         us_s[self->m + self->n + 2 + i] /= (self->E[i] * self->sc_c);
00297     }
00298 }
00299
00303 void un_scaling_lasso_sol(lasso *self, ABIPSolution *sol) {
00304     abip_int m = self->m;
00305     abip_int n = self->n;
00306
00307     abip_float *beta = (abip_float *)abip_malloc(n * sizeof(abip_float));
00308     memcpy(beta, &sol->x[m + 2], n * sizeof(abip_float));
00309     ABIP(add_scaled_array)(beta, &sol->x[m + n + 2], n, -1);
00310     ABIP(c_dot)(beta, self->E, n);
00311     ABIP(scale_array)(beta, 1 / self->sc_b, n);
00312
00313     abip_free(sol->x);
00314     abip_free(sol->y);
00315     abip_free(sol->s);
00316
00317     sol->x = beta;
00318 }
00319
00323 abip_float lasso_inner_conv_check(lasso *self, ABIPWork *w) {
00324     abip_int m = self->p;
00325     abip_int n = self->q;
00326
00327     abip_float *Qu = (abip_float *)abip_malloc((m + n + 1) * sizeof(abip_float));
00328     abip_float *Mu = (abip_float *)abip_malloc((m + n) * sizeof(abip_float));
00329
00330     memset(Mu, 0, (m + n) * sizeof(abip_float));
00331
00332     self->spe_A_times(self, &w->u[m], Mu);
00333
00334     self->spe_AT_times(self, w->u, &Mu[m]);
00335     ABIP(scale_array)(&Mu[m], -1, n);
00336
00337     if (self->Q != ABIP_NULL) {
00338         ABIP(accum_by_A)(self->Q, &w->u[m], &Mu[m]);
00339     }
00340
00341     memcpy(Qu, Mu, (m + n) * sizeof(abip_float));
00342
00343     ABIP(add_scaled_array)(Qu, self->b, m, -w->u[m + n]);
00344     ABIP(add_scaled_array)(&Qu[m], self->c, n, w->u[m + n]);
00345
00346     Qu[m + n] = -ABIP(dot)(w->u, Mu, m + n) / w->u[m + n] +
00347         ABIP(dot)(w->u, self->b, m) - ABIP(dot)(&w->u[m], self->c, n);
00348
00349     abip_float *tem = (abip_float *)abip_malloc((m + n + 1) * sizeof(abip_float));
00350     memcpy(tem, Qu, (m + n + 1) * sizeof(abip_float));
00351     ABIP(add_scaled_array)(tem, w->v_origin, m + n + 1, -1);
00352
00353     abip_float error_inner =

```

```

00354         ABIP(norm)(tem, m + n + 1) /
00355         (1 + ABIP(norm)(w->u, m + n + 1) + ABIP(norm)(w->v_origin, m + n + 1));
00356
00357     abip_free(Qu);
00358     abip_free(Mu);
00359     abip_free(tem);
00360
00361     return error_inner;
00362 }
00363
00364 void calc_lasso_residuals(lasso *self, ABIPWork *w, ABIPResiduals *r,
00365                          abip_int ipm_iter, abip_int admm_iter) {
00366
00367     abip_int p = w->m;
00368     abip_int q = w->n;
00369     abip_int m = self->m;
00370     abip_int n = self->n;
00371     abip_float this_pr;
00372     abip_float this_dr;
00373     abip_float this_gap;
00374
00375     r->tau = w->u[p + q];
00376
00377     abip_float *x = (abip_float *)abip_malloc(m * sizeof(abip_float));
00378     memcpy(x, &(w->u[m + 3]), m * sizeof(abip_float));
00379     ABIP(scale_array)(x, SQRTF(self->sc_cone2) / (r->tau * self->sc_b), m);
00380
00381     abip_float *beta_plus = (abip_float *)abip_malloc(n * sizeof(abip_float));
00382     memcpy(beta_plus, &(w->u[2 * m + 3]), n * sizeof(abip_float));
00383     for (int i = 0; i < n; i++) {
00384         beta_plus[i] *= self->E[i] / (r->tau * self->sc_b);
00385     }
00386
00387     abip_float *beta_minus = (abip_float *)abip_malloc(n * sizeof(abip_float));
00388     memcpy(beta_minus, &(w->u[2 * m + 3 + n]), n * sizeof(abip_float));
00389     for (int i = 0; i < n; i++) {
00390         beta_minus[i] *= self->E[i] / (r->tau * self->sc_b);
00391     }
00392
00393     abip_float *z = (abip_float *)abip_malloc(m * sizeof(abip_float));
00394     memcpy(z, &(w->u[1]), m * sizeof(abip_float));
00395     for (int i = 0; i < m; i++) {
00396         z[i] *= self->D[i] / (r->tau * self->sc_c);
00397     }
00398
00399     abip_float *s1 = (abip_float *)abip_malloc(n * sizeof(abip_float));
00400     memcpy(s1, &(w->v[2 * m + 3]), n * sizeof(abip_float));
00401     for (int i = 0; i < n; i++) {
00402         s1[i] /= self->E[i] * r->tau * self->sc_c;
00403     }
00404
00405     abip_float *s2 = (abip_float *)abip_malloc(n * sizeof(abip_float));
00406     memcpy(s2, &(w->v[2 * m + n + 3]), n * sizeof(abip_float));
00407     for (int i = 0; i < n; i++) {
00408         s2[i] /= self->E[i] * r->tau * self->sc_c;
00409     }
00410
00411     abip_float *pr = (abip_float *)abip_malloc(m * sizeof(abip_float));
00412     memcpy(pr, x, m * sizeof(abip_float));
00413     ABIP(accum_by_A)(self->data->A, beta_plus, pr);
00414     ABIP(scale_array)(pr, -1, m);
00415     ABIP(accum_by_A)(self->data->A, beta_minus, pr);
00416     ABIP(scale_array)(pr, -1, m);
00417     ABIP(add_scaled_array)(pr, &self->data->b[1], m, -1);
00418     this_pr = ABIP(norm)(pr, m) / MAX(ABIP(norm)(self->data->b[1], m), 1);
00419
00420     abip_float *dr1 = (abip_float *)abip_malloc(n * sizeof(abip_float));
00421     memcpy(dr1, s1, n * sizeof(abip_float));
00422     ABIP(accum_by_Atrans)(self->data->A, z, dr1);
00423     for (int i = 0; i < n; i++) {
00424         dr1[i] -= self->lambda;
00425     }
00426
00427     abip_float *dr2 = (abip_float *)abip_malloc(n * sizeof(abip_float));
00428     memcpy(dr2, s2, n * sizeof(abip_float));
00429     ABIP(scale_array)(dr2, -1, n);
00430     ABIP(accum_by_Atrans)(self->data->A, z, dr2);
00431     ABIP(scale_array)(dr2, -1, n);
00432     for (int i = 0; i < n; i++) {
00433         dr2[i] -= self->lambda;
00434     }
00435
00436     abip_float *dr = (abip_float *)abip_malloc(2 * n * sizeof(abip_float));
00437     memcpy(dr, dr1, n * sizeof(abip_float));
00438     memcpy(&dr[n], dr2, n * sizeof(abip_float));
00439     this_dr = ABIP(norm)(dr, 2 * n) / (SQRTF(2 * n) * self->lambda);
00440
00441     abip_float *lambda_ones = (abip_float *)abip_malloc(n * sizeof(abip_float));
00442     for (int i = 0; i < n; i++) {

```

```

00444     lambda_ones[i] = self->lambda;
00445 }
00446 this_gap =
00447     ABS(0.5 * ABIP(dot)(x, x, m) + ABIP(dot)(lambda_ones, beta_plus, n) +
00448         ABIP(dot)(lambda_ones, beta_minus, n) + 0.5 * ABIP(dot)(z, z, m) -
00449         ABIP(dot)(self->data->b[1], z, m)) /
00450     (1 + ABS(0.5 * ABIP(dot)(x, x, m) + ABIP(dot)(lambda_ones, beta_plus, n) +
00451         ABIP(dot)(lambda_ones, beta_minus, n)));
00452
00453 r->last_ipm_iter = ipm_iter;
00454 r->last_admm_iter = admm_iter;
00455
00456 r->dobj = (-0.5 * ABIP(dot)(z, z, m) + ABIP(dot)(self->data->b[1], z, m));
00457 r->pobj = (0.5 * ABIP(dot)(x, x, m) + ABIP(dot)(lambda_ones, beta_plus, n) +
00458     ABIP(dot)(lambda_ones, beta_minus, n));
00459
00460 r->res_dif = MAX(MAX(ABS(this_pr - r->res_pri), ABS(this_dr - r->res_dual)),
00461     ABS(this_gap - r->rel_gap));
00462 r->res_pri = this_pr;
00463 r->res_dual = this_dr;
00464 r->rel_gap = this_gap;
00465 r->error_ratio =
00466     MAX(r->res_pri / self->stgs->eps_p,
00467     MAX(r->res_dual / self->stgs->eps_d, r->rel_gap / self->stgs->eps_g));
00468
00469 if (ABIP(dot)(self->c, &w->u[p], q) < 0) {
00470     abip_float *Ax = (abip_float *)abip_malloc(p * sizeof(abip_float));
00471     memset(Ax, 0, p * sizeof(abip_float));
00472     self->spe_A_times(self, &w->u[p], Ax);
00473     r->res_unbdd = ABIP(norm)(Ax, p) / (-ABIP(dot)(self->c, &w->u[p], q));
00474     abip_free(Ax);
00475 } else {
00476     r->res_unbdd = INFINITY;
00477 }
00478
00479 if (ABIP(dot)(self->b, w->u, p) > 0) {
00480     abip_float *ATy_s = (abip_float *)abip_malloc(q * sizeof(abip_float));
00481     memset(ATy_s, 0, q * sizeof(abip_float));
00482     self->spe_AT_times(self, w->u, ATy_s);
00483     ABIP(add_scaled_array)(ATy_s, &w->v_origin[p], q, 1);
00484
00485     r->res_infeas = ABIP(norm)(ATy_s, q) / ABIP(dot)(self->b, w->u, p);
00486     abip_free(ATy_s);
00487 } else {
00488     r->res_infeas = INFINITY;
00489 }
00490
00491 abip_free(x);
00492 abip_free(beta_plus);
00493 abip_free(beta_minus);
00494 abip_free(z);
00495 abip_free(s1);
00496 abip_free(s2);
00497 abip_free(pr);
00498 abip_free(dr);
00499 abip_free(dr1);
00500 abip_free(dr2);
00501 abip_free(lambda_ones);
00502 }
00503
00507 cs *form_lasso_kkt(lasso *self) {
00508     abip_int n = self->n;
00509     abip_int m = self->m;
00510     cs *LTL;
00511
00512     cs *Y1 = cs_spalloc(m, n, self->A->p[n], 1, 0);
00513     memcpy(Y1->i, self->A->i, self->A->p[n] * sizeof(abip_int));
00514     memcpy(Y1->p, self->A->p, (n + 1) * sizeof(abip_int));
00515     memcpy(Y1->x, self->A->x, self->A->p[n] * sizeof(abip_float));
00516
00517     if (m > n) {
00518         cs *Y2 = cs_spalloc(m, n, self->A->p[n], 1, 0);
00519         memcpy(Y2->i, self->A->i, self->A->p[n] * sizeof(abip_int));
00520         memcpy(Y2->p, self->A->p, (n + 1) * sizeof(abip_int));
00521         memcpy(Y2->x, self->A->x, self->A->p[n] * sizeof(abip_float));
00522
00523         cs *T1 = cs_spalloc(n, n, n, 1, 1); /* create triplet identity matrix */
00524
00525         for (int i = 0; i < n; i++) {
00526             cs_entry(T1, i, i, 0.5);
00527         }
00528
00529         cs *half_eye = cs_compress(T1);
00530
00531         cs_spfree(T1);
00532
00533         for (int i = 0; i < Y1->p[n]; i++) {

```

```

00534     Y1->x[i] /= self->D_hat[Y1->i[i]];
00535 }
00536
00537 LTL = cs_add(half_eye, cs_multiply(cs_transpose(Y2, 1), Y1), 1, 1);
00538
00539 cs_spfree(Y1);
00540 cs_spfree(Y2);
00541 cs_spfree(half_eye);
00542 } else {
00543     cs *YYT = cs_multiply(Y1, cs_transpose(Y1, 1));
00544     cs_spfree(Y1);
00545
00546     cs *diag = cs_salloc(m, m, m, 1, 1);
00547
00548     for (int i = 0; i < m; i++) {
00549         cs_entry(diag, i, i, self->D_hat[i]);
00550     }
00551     cs *diag_D_hat = cs_compress(diag);
00552     cs_spfree(diag);
00553
00554     LTL = cs_add(diag_D_hat, YYT, 1, 2);
00555     cs_spfree(YYT);
00556 }
00557
00558 for (int i = 0; i < LTL->n; i++) {
00559     for (int j = LTL->p[i]; j < LTL->p[i + 1]; j++) {
00560         if (LTL->i[j] > i) LTL->x[j] = 0;
00561     }
00562 }
00563 cs_dropzeros(LTL);
00564 return LTL;
00565 }
00566
00571 void init_lasso_precon(lasso *self) {
00572     self->L->M = (abip_float *)abip_malloc(self->p * sizeof(abip_float));
00573     memset(self->L->M, 0, self->p * sizeof(abip_float));
00574
00575     abip_float *M = self->L->M;
00576
00577     M[0] = 1 / (self->stgs->rho_y + 1);
00578
00579     for (int i = 0; i < self->A->p[self->A->n]; i++) {
00580         M[self->A->i[i] + 1] += 2 * POWF(self->A->x[i], 2);
00581     }
00582
00583     for (int i = 1; i < self->p; i++) {
00584         M[i] = 1 / (self->stgs->rho_y + M[i] +
00585             self->sc_cone2 * POWF(self->D[i - 1], 2));
00586     }
00587 }
00588
00592 abip_float get_lasso_pcg_tol(abip_int k, abip_float error_ratio,
00593     abip_float norm_p) {
00594     if (k == -1) {
00595         return 1e-9 * norm_p;
00596     } else {
00597         if (error_ratio > 100000) {
00598             return MAX(1e-9, 1.2e-2 * norm_p / POWF((k + 1), 2));
00599         } else if (error_ratio > 30000) {
00600             return MAX(1e-9, 8e-3 * norm_p / POWF((k + 1), 2));
00601         } else if (error_ratio > 10000) {
00602             return MAX(1e-9, 6e-3 * norm_p / POWF((k + 1), 2));
00603         } else if (error_ratio > 3000) {
00604             return MAX(1e-9, 5e-3 * norm_p / POWF((k + 1), 2));
00605         } else if (error_ratio > 1000) {
00606             return MAX(1e-9, 3e-3 * norm_p / POWF((k + 1), 2));
00607         } else if (error_ratio > 300) {
00608             return MAX(1e-9, 2e-3 * norm_p / POWF((k + 1), 2));
00609         } else if (error_ratio > 100) {
00610             return MAX(1e-9, 1.5e-3 * norm_p / POWF((k + 1), 2));
00611         } else if (error_ratio > 30) {
00612             return MAX(1e-9, 8e-4 * norm_p / POWF((k + 1), 2));
00613         } else if (error_ratio > 10) {
00614             return MAX(1e-9, 6e-4 * norm_p / POWF((k + 1), 2));
00615         } else {
00616             return MAX(1e-9, 5e-4 * norm_p / POWF((k + 1), 2));
00617         }
00618     }
00619 }
00620
00624 abip_int init_lasso_linsys_work(lasso *self) {
00625     if (self->stgs->linsys_solver == 0) {
00626         self->L->K = cs_transpose(form_lasso_kkt(self), 1);
00627     } else if (self->stgs->linsys_solver == 1) {
00628         self->L->K = form_lasso_kkt(self);
00629     } else if (self->stgs->linsys_solver == 2) {
00630         self->L->K = form_lasso_kkt(self);

```

```

00631 } else if (self->stgs->linsys_solver == 3) {
00632     init_lasso_precon(self);
00633     self->L->K = ABIP_NULL;
00634 } else if (self->stgs->linsys_solver == 4) {
00635     self->L->K = cs_transpose(form_lasso_kkt(self), 1);
00636 } else if (self->stgs->linsys_solver == 5) {
00637     self->L->K = form_lasso_kkt(self);
00638 } else {
00639     printf("\nlinsys solver type error\n");
00640     return -1;
00641 }
00642 return ABIP(init_linsys_work)(self);
00643 }
00644
00648 abip_int solve_lasso_linsys(lasso *self, abip_float *b,
00649                             abip_float *pcg_warm_start, abip_int iter,
00650                             abip_float error_ratio) {
00651     ABIP(timer) linsys_timer;
00652     ABIP(tic) (&linsys_timer);
00653
00654     ABIP(scale_array) (&b[self->p], -1, self->q);
00655
00656     if (self->stgs->linsys_solver == 3) { // pcg
00657
00658         abip_int p = self->p;
00659         abip_float norm_p = ABIP(norm)(b, p);
00660
00661         self->spe_A_times(self, &b[p], b);
00662         abip_float pcg_tol = get_lasso_pcg_tol(iter, error_ratio, norm_p);
00663         abip_int cg_its = ABIP(solve_linsys)(self, b, p, pcg_warm_start, pcg_tol);
00664
00665         if (iter >= 0) {
00666             self->L->total_cg_its += cg_its;
00667         }
00668     }
00669
00670     else { // direct methods
00671         abip_float *b2 = (abip_float *)abip_malloc(self->p * sizeof(abip_float));
00672         memcpy(b2, b, self->p * sizeof(abip_float));
00673         self->spe_A_times(self, &b[self->p], b2);
00674         b[0] = b2[0] / (1 + self->stgs->rho_y);
00675         abip_int n = self->n;
00676         abip_int m = self->m;
00677
00678         if (m > n) {
00679             for (int i = 0; i < m; i++) {
00680                 b2[i + 1] /= self->D_hat[i];
00681                 b[i + 1] = b2[i + 1];
00682             }
00683
00684             abip_float *tmp = (abip_float *)abip_malloc(n * sizeof(abip_float));
00685             memset(tmp, 0, n * sizeof(abip_float));
00686             ABIP(accum_by_Atrans)(self->A, &b[1], tmp);
00687
00688             ABIP(solve_linsys)(self, tmp, n, ABIP_NULL, 0);
00689
00690             abip_float *tmp2 = (abip_float *)abip_malloc(m * sizeof(abip_float));
00691             memset(tmp2, 0, m * sizeof(abip_float));
00692
00693             ABIP(accum_by_A)(self->A, tmp, tmp2);
00694             for (int i = 0; i < m; i++) {
00695                 tmp2[i] /= self->D_hat[i];
00696             }
00697
00698             ABIP(add_scaled_array) (&b[1], tmp2, m, -1);
00699
00700             abip_free(tmp);
00701             abip_free(tmp2);
00702         } else {
00703             memcpy(&b[1], &b2[1], m * sizeof(abip_float));
00704
00705             ABIP(solve_linsys)(self, &b[1], m, ABIP_NULL, 0);
00706         }
00707
00708         abip_free(b2);
00709     }
00710
00711     ABIP(scale_array) (&b[self->p], -1, self->q);
00712     self->spe_AT_times(self, b, &b[self->p]);
00713
00714     self->L->total_solve_time += ABIP(tocq) (&linsys_timer);
00715
00716     return 0;
00717 }
00718
00722 void free_lasso_linsys_work(lasso *self) { ABIP(free_linsys)(self); }

```

## 4.185 source/linalg.c File Reference

```
#include "linalg.h"
#include <math.h>
```

### Functions

- void [ABIP\(\)](#) [c\\_dot](#) ([abip\\_float](#) \*x, const [abip\\_float](#) \*y, const [abip\\_int](#) len)  
*Elementwise multiplication of two vectors.*
- [abip\\_float](#) [ABIP\(\)](#) [vec\\_mean](#) ([abip\\_float](#) \*x, [abip\\_int](#) len)  
*Calculate the mean of a vector.*
- void [ABIP\(\)](#) [set\\_as\\_scaled\\_array](#) ([abip\\_float](#) \*x, const [abip\\_float](#) \*a, const [abip\\_float](#) b, [abip\\_int](#) len)  
*Elementwise multiplication of a vector by a scalar.*
- void [ABIP\(\)](#) [set\\_as\\_sqrt](#) ([abip\\_float](#) \*x, const [abip\\_float](#) \*v, [abip\\_int](#) len)  
*Elementwise square root of a vector.*
- void [ABIP\(\)](#) [set\\_as\\_sq](#) ([abip\\_float](#) \*x, const [abip\\_float](#) \*v, [abip\\_int](#) len)  
*Elementwise square of a vector.*
- void [ABIP\(\)](#) [scale\\_array](#) ([abip\\_float](#) \*a, const [abip\\_float](#) b, [abip\\_int](#) len)  
*Elementwise multiplication of a vector by a scalar with replacement.*
- [abip\\_float](#) [ABIP\(\)](#) [dot](#) (const [abip\\_float](#) \*x, const [abip\\_float](#) \*y, [abip\\_int](#) len)  
*Dot product of two vectors.*
- [abip\\_float](#) [ABIP\(\)](#) [norm\\_sq](#) (const [abip\\_float](#) \*v, [abip\\_int](#) len)  
*L2 norm of a vector.*
- [abip\\_float](#) [ABIP\(\)](#) [norm](#) (const [abip\\_float](#) \*v, [abip\\_int](#) len)  
*Square of L2 norm of a vector.*
- [abip\\_float](#) [ABIP\(\)](#) [norm\\_1](#) (const [abip\\_float](#) \*x, const [abip\\_int](#) len)  
*L1 norm of a vector.*
- [abip\\_float](#) [ABIP\(\)](#) [cone\\_norm\\_1](#) (const [abip\\_float](#) \*x, const [abip\\_int](#) len)  
*The absolute value of the largest component of x.*
- [abip\\_float](#) [ABIP\(\)](#) [norm\\_inf](#) (const [abip\\_float](#) \*a, [abip\\_int](#) len)  
*Calculate the maximum absolute value of a vector.*
- void [ABIP\(\)](#) [add\\_array](#) ([abip\\_float](#) \*a, const [abip\\_float](#) b, [abip\\_int](#) len)  
*Elementwise addition of two vectors with coefficients.*
- void [ABIP\(\)](#) [add\\_scaled\\_array](#) ([abip\\_float](#) \*a, const [abip\\_float](#) \*b, [abip\\_int](#) len, const [abip\\_float](#) sc)  
*Elementwise addition of two vectors with coefficients 1.*
- [abip\\_float](#) [ABIP\(\)](#) [norm\\_diff](#) (const [abip\\_float](#) \*a, const [abip\\_float](#) \*b, [abip\\_int](#) len)  
*L2 norm of the difference of two vectors.*
- [abip\\_float](#) [ABIP\(\)](#) [norm\\_inf\\_diff](#) (const [abip\\_float](#) \*a, const [abip\\_float](#) \*b, [abip\\_int](#) len)  
*Maximum of the difference of two vectors.*
- [abip\\_int](#) [arr\\_ind](#) (const [abip\\_int](#) i\_col, const [abip\\_int](#) i\_row, const [abip\\_int](#) nrows, const [abip\\_int](#) ncols, const [abip\\_int](#) format)
- [abip\\_float](#) \*[ABIP\(\)](#) [csc\\_to\\_dense](#) (const [cs](#) \*in\_csc, const [abip\\_int](#) out\_format)  
*Convert a CSC matrix to a dense matrix.*

### 4.185.1 Function Documentation

#### 4.185.1.1 `add_array()`

```
void ABIP() add_array (
    abip_float * a,
    const abip_float b,
    abip_int len )
```

Elementwise addition of two vectors with coefficients.

Definition at line 181 of file [linalg.c](#).

#### 4.185.1.2 `add_scaled_array()`

```
void ABIP() add_scaled_array (
    abip_float * a,
    const abip_float * b,
    abip_int len,
    const abip_float sc )
```

Elementwise addition of two vectors with coefficients 1.

Definition at line 192 of file [linalg.c](#).

#### 4.185.1.3 `arr_ind()`

```
abip_int arr_ind (
    const abip_int i_col,
    const abip_int i_row,
    const abip_int nrows,
    const abip_int ncols,
    const abip_int format )
```

Definition at line 237 of file [linalg.c](#).

#### 4.185.1.4 `c_dot()`

```
void ABIP() c_dot (
    abip_float * x,
    const abip_float * y,
    const abip_int len )
```

Elementwise multiplication of two vectors.

Definition at line 9 of file [linalg.c](#).

#### 4.185.1.5 cone\_norm\_1()

```
abip_float ABIP() cone_norm_1 (
    const abip_float * x,
    const abip_int len )
```

The absolute value of the largest component of x.

Definition at line 144 of file [linalg.c](#).

#### 4.185.1.6 csc\_to\_dense()

```
abip_float *ABIP() csc_to_dense (
    const cs * in_csc,
    const abip_int out_format )
```

Convert a CSC matrix to a dense matrix.

Definition at line 247 of file [linalg.c](#).

#### 4.185.1.7 dot()

```
abip_float ABIP() dot (
    const abip_float * x,
    const abip_float * y,
    abip_int len )
```

Dot product of two vectors.

Definition at line 85 of file [linalg.c](#).

#### 4.185.1.8 norm()

```
abip_float ABIP() norm (
    const abip_float * v,
    abip_int len )
```

Square of L2 norm of a vector.

Definition at line 119 of file [linalg.c](#).



#### 4.185.1.9 norm\_1()

```
abip_float ABIP() norm_1 (
    const abip_float * x,
    const abip_int len )
```

L1 norm of a vector.

Definition at line 130 of file [linalg.c](#).

#### 4.185.1.10 norm\_diff()

```
abip_float ABIP() norm_diff (
    const abip_float * a,
    const abip_float * b,
    abip_int len )
```

L2 norm of the difference of two vectors.

Definition at line 207 of file [linalg.c](#).

#### 4.185.1.11 norm\_inf()

```
abip_float ABIP() norm_inf (
    const abip_float * a,
    abip_int len )
```

Calculate the maximum absolute value of a vector.

Definition at line 161 of file [linalg.c](#).

#### 4.185.1.12 norm\_inf\_diff()

```
abip_float ABIP() norm_inf_diff (
    const abip_float * a,
    const abip_float * b,
    abip_int len )
```

Maximum of the difference of two vectors.

Definition at line 223 of file [linalg.c](#).

#### 4.185.1.13 `norm_sq()`

```
abip_float ABIP() norm_sq (
    const abip_float * v,
    abip_int len )
```

L2 norm of a vector.

Definition at line 102 of file [linalg.c](#).

#### 4.185.1.14 `scale_array()`

```
void ABIP() scale_array (
    abip_float * a,
    const abip_float b,
    abip_int len )
```

Elementwise multiplication of a vector by a scalar with replacement.

Definition at line 71 of file [linalg.c](#).

#### 4.185.1.15 `set_as_scaled_array()`

```
void ABIP() set_as_scaled_array (
    abip_float * x,
    const abip_float * a,
    const abip_float b,
    abip_int len )
```

Elementwise multiplication of a vector by a scalar.

Definition at line 37 of file [linalg.c](#).

#### 4.185.1.16 `set_as_sq()`

```
void ABIP() set_as_sq (
    abip_float * x,
    const abip_float * v,
    abip_int len )
```

Elementwise square of a vector.

Definition at line 60 of file [linalg.c](#).

**4.185.1.17 set\_as\_sqrt()**

```
void ABIP() set_as_sqrt (
    abip_float * x,
    const abip_float * v,
    abip_int len )
```

Elementwise square root of a vector.

Definition at line 49 of file [linalg.c](#).

**4.185.1.18 vec\_mean()**

```
abip_float ABIP() vec_mean (
    abip_float * x,
    abip_int len )
```

Calculate the mean of a vector.

Definition at line 19 of file [linalg.c](#).

**4.186 linalg.c**

[Go to the documentation of this file.](#)

```
00001 #include "linalg.h"
00002
00003 #include <math.h>
00004
00008 /* x = x .* y */
00009 void ABIP(c_dot)(abip_float *x, const abip_float *y, const abip_int len) {
00010     for (int i = 0; i < len; i++) {
00011         x[i] *= y[i];
00012     }
00013 }
00014
00018 /* y = mean(x) */
00019 abip_float ABIP(vec_mean)(abip_float *x, abip_int len) {
00020     if (len <= 0 || x == ABIP_NULL) {
00021         printf("invalid ABIP(vec_mean) parameter");
00022         return -1;
00023     }
00024     abip_float y = 0;
00025
00026     for (int i = 0; i < len; i++) {
00027         y += x[i];
00028     }
00029
00030     return y / len;
00031 }
00032
00036 /* x = b*a */
00037 void ABIP(set_as_scaled_array)(abip_float *x, const abip_float *a,
00038                                const abip_float b, abip_int len) {
00039     abip_int i;
00040     for (i = 0; i < len; ++i) {
00041         x[i] = b * a[i];
00042     }
00043 }
00044
00048 /* x = sqrt(v) */
00049 void ABIP(set_as_sqrt)(abip_float *x, const abip_float *v, abip_int len) {
00050     abip_int i;
00051     for (i = 0; i < len; ++i) {
00052         x[i] = SQRTF(v[i]);
00053     }
00054 }
```

```

00055
00059 /* x = v.^2 */
00060 void ABIP(set_as_sq)(abip_float *x, const abip_float *v, abip_int len) {
00061     abip_int i;
00062     for (i = 0; i < len; ++i) {
00063         x[i] = v[i] * v[i];
00064     }
00065 }
00066
00070 /* a *= b */
00071 void ABIP(scale_array)(abip_float *a, const abip_float b, abip_int len) {
00072     if (a == ABIP_NULL) {
00073         return;
00074     }
00075     abip_int i;
00076     for (i = 0; i < len; ++i) {
00077         a[i] *= b;
00078     }
00079 }
00080
00084 /* x'*y */
00085 abip_float ABIP(dot)(const abip_float *x, const abip_float *y, abip_int len) {
00086     if (x == ABIP_NULL || y == ABIP_NULL) {
00087         return 0;
00088     }
00089
00090     abip_int i;
00091     abip_float ip = 0.0;
00092     for (i = 0; i < len; ++i) {
00093         ip += x[i] * y[i];
00094     }
00095     return ip;
00096 }
00097
00101 /* ||v||_2^2 */
00102 abip_float ABIP(norm_sq)(const abip_float *v, abip_int len) {
00103     if (v == ABIP_NULL) {
00104         return 0;
00105     }
00106
00107     abip_int i;
00108     abip_float nmsq = 0.0;
00109     for (i = 0; i < len; ++i) {
00110         nmsq += v[i] * v[i];
00111     }
00112     return nmsq;
00113 }
00114
00118 /* ||v||_2 */
00119 abip_float ABIP(norm)(const abip_float *v, abip_int len) {
00120     if (v == ABIP_NULL) {
00121         return 0;
00122     }
00123     return SQRTF(ABIP(norm_sq)(v, len));
00124 }
00125
00129 /* ||x||_1 */
00130 abip_float ABIP(norm_1)(const abip_float *x, const abip_int len) {
00131     if (x == ABIP_NULL) {
00132         return 0;
00133     }
00134     abip_float result = 0;
00135     for (int i = 0; i < len; i++) {
00136         result += ABS(x[i]);
00137     }
00138     return result;
00139 }
00140
00144 abip_float ABIP(cone_norm_1)(const abip_float *x, const abip_int len) {
00145     abip_int i;
00146     abip_float tmp;
00147     abip_float max = 0.0;
00148     for (i = 0; i < len; ++i) {
00149         tmp = x[i];
00150         if (tmp > max) {
00151             max = tmp;
00152         }
00153     }
00154     return ABS(max);
00155 }
00156
00160 /* max(|v|) */
00161 abip_float ABIP(norm_inf)(const abip_float *a, abip_int len) {
00162     if (a == ABIP_NULL || len == 0) {
00163         return 0;
00164     }
00165     abip_int i;

```

```

00166     abip_float tmp;
00167     abip_float max = 0.0;
00168     for (i = 0; i < len; ++i) {
00169         tmp = ABS(a[i]);
00170         if (tmp > max) {
00171             max = tmp;
00172         }
00173     }
00174     return max;
00175 }
00176
00180 /* a += b */
00181 void ABIP(add_array)(abip_float *a, const abip_float b, abip_int len) {
00182     abip_int i;
00183     for (i = 0; i < len; ++i) {
00184         a[i] += b;
00185     }
00186 }
00187
00191 /* saxpy a += sc*b */
00192 void ABIP(add_scaled_array)(abip_float *a, const abip_float *b, abip_int len,
00193                             const abip_float sc) {
00194     if (b == ABIP_NULL) {
00195         return;
00196     }
00197     abip_int i;
00198     for (i = 0; i < len; ++i) {
00199         a[i] += sc * b[i];
00200     }
00201 }
00202
00206 /* ||a-b||_2^2 */
00207 abip_float ABIP(norm_diff)(const abip_float *a, const abip_float *b,
00208                             abip_int len) {
00209     abip_int i;
00210     abip_float tmp;
00211     abip_float nm_diff = 0.0;
00212     for (i = 0; i < len; ++i) {
00213         tmp = (a[i] - b[i]);
00214         nm_diff += tmp * tmp;
00215     }
00216     return SQRTF(nm_diff);
00217 }
00218
00222 /* max(|a-b|) */
00223 abip_float ABIP(norm_inf_diff)(const abip_float *a, const abip_float *b,
00224                                 abip_int len) {
00225     abip_int i;
00226     abip_float tmp;
00227     abip_float max = 0.0;
00228     for (i = 0; i < len; ++i) {
00229         tmp = ABS(a[i] - b[i]);
00230         if (tmp > max) {
00231             max = tmp;
00232         }
00233     }
00234     return max;
00235 }
00236
00237 abip_int arr_ind(const abip_int i_col, const abip_int i_row,
00238                  const abip_int nrows, const abip_int ncols,
00239                  const abip_int format) {
00240     return (format == RowMajor) ? (i_col + i_row * ncols)
00241                                : (i_row + i_col * nrows);
00242 }
00243
00247 abip_float *ABIP(csc_to_dense)(const cs *in_csc, const abip_int out_format) {
00248     abip_int i_row, i_col, nnz_in_col, i_val = 0, i_nnz;
00249     const abip_int nrows = in_csc->m;
00250     const abip_int ncols = in_csc->n;
00251     abip_float *out_matrix =
00252         (abip_float *)abip_malloc(nrows * ncols * sizeof(abip_float));
00253     memset(out_matrix, 0, nrows * ncols * sizeof(abip_float));
00254     abip_int *col_nnz = in_csc->p;
00255     abip_int *rows = in_csc->i;
00256     abip_float *values = in_csc->x;
00257
00258     for (i_col = 0; i_col < ncols; i_col++) {
00259         nnz_in_col = col_nnz[i_col + 1] - col_nnz[i_col];
00260         if (nnz_in_col > 0) {
00261             for (i_nnz = 0; i_nnz < nnz_in_col; i_nnz++) {
00262                 i_row = rows[i_val];
00263                 out_matrix[arr_ind(i_col, i_row, nrows, ncols, out_format)] =
00264                     values[i_val];
00265                 i_val++;
00266             }
00267         }
00268     }

```

```

00268     }
00269
00270     return out_matrix;
00271 }

```

## 4.187 source/linsys.c File Reference

```

#include "linsys.h"
#include "amd.h"

```

### Macros

- `#define _CRT_SECURE_NO_WARNINGS`
- `#define MIN_SCALE (1e-3)`
- `#define MAX_SCALE (1e3)`

### Functions

- `abip_int ABIP() copy_A_matrix (ABIPMatrix **dstp, const ABIPMatrix *src)`  
*Copy a matrix.*
- `char *ABIP() get_lin_sys_method (spe_problem *spe)`  
*Get the method used to solve the linear system.*
- `char *ABIP() get_lin_sys_summary (spe_problem *self, ABIPInfo *info)`  
*Get the summary information of the linear system.*
- `abip_int ABIP() validate_lin_sys (const ABIPMatrix *A)`  
*Check the validity of the linear system.*
- `void ABIP() free_A_matrix (ABIPMatrix *A)`  
*Free the memory of a matrix.*
- `void ABIP() accum_by_Atrans (const ABIPMatrix *A, const abip_float *x, abip_float *y)`  
*Add the transposed matrix-vector product to a vector.*
- `void ABIP() accum_by_A (const ABIPMatrix *A, const abip_float *x, abip_float *y)`  
*Add the matrix-vector product to a vector.*
- `cs *permute_kkt (spe_problem *spe)`
- `_MKL_DSS_HANDLE_t init_mkl_work (cs *K)`
- `abip_int mkl_solve_linsys (_MKL_DSS_HANDLE_t handle, abip_float *b, abip_int n)`
- `abip_int init_pardiso (spe_problem *self)`
- `abip_int pardiso_solve (spe_problem *self, abip_float *b, abip_int n)`
- `abip_int pardiso_free (spe_problem *self)`
- `abip_int LDL_factor (cs *A, cs **L, abip_float *Dinv)`
- `abip_int abip_cholsol (spe_problem *self, abip_float *b, abip_int n)`
- `abip_int pcg (spe_problem *self, abip_float *b, abip_float *x, abip_float rho_x, abip_int max_iter, abip_float tol)`  
*Preconditioned conjugate gradient method for general linear system.*
- `abip_int qcp_pcg (spe_problem *self, abip_float *b, abip_float *x, abip_int max_iter, abip_float tol)`  
*Preconditioned conjugate gradient method for qcp.*
- `abip_int svmqp_pcg (spe_problem *self, abip_float *b, abip_float *x, abip_int max_iter, abip_float tol)`  
*Customized preconditioned conjugate gradient solver for QP formulation of SVM.*
- `abip_int init_dense_chol (spe_problem *spe)`  
*MKL-LAPACK dense cholesky linear system solver.*

- [abip\\_int dense\\_chol\\_sol](#) ([spe\\_problem](#) \*spe, [abip\\_float](#) \*b, [abip\\_int](#) n)
- [abip\\_int dense\\_chol\\_free](#) ([spe\\_problem](#) \*spe)
- [abip\\_int ABIP\(\)](#) [init\\_linsys\\_work](#) ([spe\\_problem](#) \*spe)  
*Initialize linear system solver work space.*
- [abip\\_int ABIP\(\)](#) [solve\\_linsys](#) ([spe\\_problem](#) \*spe, [abip\\_float](#) \*b, [abip\\_int](#) n, [abip\\_float](#) \*pcg\_warm\_start, [abip\\_float](#) pcg\_tol)  
*solve linear system according to the specific linsys solver*
- [abip\\_int ABIP\(\)](#) [free\\_linsys](#) ([spe\\_problem](#) \*spe)  
*free memory for linear system solver*

## 4.187.1 Macro Definition Documentation

### 4.187.1.1 `_CRT_SECURE_NO_WARNINGS`

```
#define _CRT_SECURE_NO_WARNINGS
```

Definition at line 1 of file [linsys.c](#).

### 4.187.1.2 `MAX_SCALE`

```
#define MAX_SCALE (1e3)
```

Definition at line 7 of file [linsys.c](#).

### 4.187.1.3 `MIN_SCALE`

```
#define MIN_SCALE (1e-3)
```

Definition at line 6 of file [linsys.c](#).

## 4.187.2 Function Documentation

### 4.187.2.1 `abip_cholsol()`

```
abip\_int abip\_cholsol (  
    spe\_problem * self,  
    abip\_float * b,  
    abip\_int n )
```

Definition at line 613 of file [linsys.c](#).

#### 4.187.2.2 accum\_by\_A()

```
void ABIP() accum_by_A (
    const ABIPMatrix * A,
    const abip_float * x,
    abip_float * y )
```

Add the matrix-vector product to a vector.

Definition at line 229 of file [linsys.c](#).

#### 4.187.2.3 accum\_by\_Atrans()

```
void ABIP() accum_by_Atrans (
    const ABIPMatrix * A,
    const abip_float * x,
    abip_float * y )
```

Add the transposed matrix-vector product to a vector.

Definition at line 191 of file [linsys.c](#).

#### 4.187.2.4 copy\_A\_matrix()

```
abip_int ABIP() copy_A_matrix (
    ABIPMatrix ** dstp,
    const ABIPMatrix * src )
```

Copy a matrix.

Definition at line 12 of file [linsys.c](#).

#### 4.187.2.5 dense\_chol\_free()

```
abip_int dense_chol_free (
    spe_problem * spe )
```

Definition at line 1012 of file [linsys.c](#).



#### 4.187.2.6 dense\_chol\_sol()

```
abip_int dense_chol_sol (
    spe_problem * spe,
    abip_float * b,
    abip_int n )
```

Definition at line 1005 of file [linsys.c](#).

#### 4.187.2.7 free\_A\_matrix()

```
void ABIP() free_A_matrix (
    ABIPMatrix * A )
```

Free the memory of a matrix.

Definition at line 152 of file [linsys.c](#).

#### 4.187.2.8 free\_linsys()

```
abip_int ABIP() free_linsys (
    spe_problem * spe )
```

free memory for linear system solver

Definition at line 1181 of file [linsys.c](#).

#### 4.187.2.9 get\_lin\_sys\_method()

```
char *ABIP() get_lin_sys_method (
    spe_problem * spe )
```

Get the method used to solve the linear system.

Definition at line 39 of file [linsys.c](#).

#### 4.187.2.10 get\_lin\_sys\_summary()

```
char *ABIP() get_lin_sys_summary (
    spe_problem * self,
    ABIPInfo * info )
```

Get the summary information of the linear system.

Definition at line 71 of file [linsys.c](#).

#### 4.187.2.11 `init_dense_chol()`

```
abip_int init_dense_chol (
    spe_problem * spe )
```

MKL-LAPACK dense cholesky linear system solver.

Definition at line 995 of file [linsys.c](#).

#### 4.187.2.12 `init_linsys_work()`

```
abip_int ABIP() init_linsys_work (
    spe_problem * spe )
```

Initialize linear system solver work space.

Definition at line 1027 of file [linsys.c](#).

#### 4.187.2.13 `init_mkl_work()`

```
_MKL_DSS_HANDLE_t init_mkl_work (
    cs * K )
```

Definition at line 318 of file [linsys.c](#).

#### 4.187.2.14 `init_pardiso()`

```
abip_int init_pardiso (
    spe_problem * self )
```

Definition at line 361 of file [linsys.c](#).

#### 4.187.2.15 `LDL_factor()`

```
abip_int LDL_factor (
    cs * A,
    cs ** L,
    abip_float * Dinv )
```

Definition at line 542 of file [linsys.c](#).

**4.187.2.16 mkl\_solve\_linsys()**

```
abip_int mkl_solve_linsys (
    _MKL_DSS_HANDLE_t handle,
    abip_float * b,
    abip_int n )
```

Definition at line 345 of file [linsys.c](#).

**4.187.2.17 pardiso\_free()**

```
abip_int pardiso_free (
    spe_problem * self )
```

Definition at line 527 of file [linsys.c](#).

**4.187.2.18 pardiso\_solve()**

```
abip_int pardiso_solve (
    spe_problem * self,
    abip_float * b,
    abip_int n )
```

Definition at line 503 of file [linsys.c](#).

**4.187.2.19 pcg()**

```
abip_int pcg (
    spe_problem * self,
    abip_float * b,
    abip_float * x,
    abip_float rho_x,
    abip_int max_iter,
    abip_float tol )
```

Preconditioned conjugate gradient method for general linear system.

Definition at line 630 of file [linsys.c](#).

**4.187.2.20 permute\_kkt()**

```
cs * permute_kkt (
    spe_problem * spe )
```

Definition at line 275 of file [linsys.c](#).

#### 4.187.2.21 qcp\_pcg()

```
abip_int qcp_pcg (
    spe_problem * self,
    abip_float * b,
    abip_float * x,
    abip_int max_iter,
    abip_float tol )
```

Preconditioned conjugate gradient method for qcp.

Definition at line 755 of file [linsys.c](#).

#### 4.187.2.22 solve\_linsys()

```
abip_int ABIP() solve_linsys (
    spe_problem * spe,
    abip_float * b,
    abip_int n,
    abip_float * pcg_warm_start,
    abip_float pcg_tol )
```

solve linear system according to the specific linsys solver

Definition at line 1141 of file [linsys.c](#).

#### 4.187.2.23 svmqp\_pcg()

```
abip_int svmqp_pcg (
    spe_problem * self,
    abip_float * b,
    abip_float * x,
    abip_int max_iter,
    abip_float tol )
```

Customized preconditioned conjugate gradient solver for QP formulation of SVM.

Definition at line 894 of file [linsys.c](#).

#### 4.187.2.24 validate\_lin\_sys()

```
abip_int ABIP() validate_lin_sys (
    const ABIPMatrix * A )
```

Check the validity of the linear system.

Definition at line 102 of file [linsys.c](#).

## 4.188 linsys.c

[Go to the documentation of this file.](#)

```

00001 #define _CRT_SECURE_NO_WARNINGS
00002 #include "linsys.h"
00003
00004 #include "amd.h"
00005
00006 #define MIN_SCALE (1e-3)
00007 #define MAX_SCALE (1e3)
00008
00012 abip_int ABIP(copy_A_matrix)(ABIPMatrix **dstp, const ABIPMatrix *src) {
00013     abip_int Annz = src->p[src->n];
00014     ABIPMatrix *A = (ABIPMatrix *)abip_calloc(1, sizeof(ABIPMatrix));
00015     if (!A) {
00016         return 0;
00017     }
00018     A->n = src->n;
00019     A->m = src->m;
00020     A->x = (abip_float *)abip_malloc(sizeof(abip_float) * Annz);
00021     A->i = (abip_int *)abip_malloc(sizeof(abip_int) * Annz);
00022     A->p = (abip_int *)abip_malloc(sizeof(abip_int) * (src->n + 1));
00023
00024     if (!A->x || !A->i || !A->p) {
00025         return 0;
00026     }
00027
00028     memcpy(A->x, src->x, sizeof(abip_float) * Annz);
00029     memcpy(A->i, src->i, sizeof(abip_int) * Annz);
00030     memcpy(A->p, src->p, sizeof(abip_int) * (src->n + 1));
00031
00032     *dstp = A;
00033     return 1;
00034 }
00035
00039 char *ABIP(get_lin_sys_method)(spe_problem *spe) {
00040     char *tmp = (char *)abip_malloc(sizeof(char) * 128);
00041
00042     if (spe->data->A == ABIP_NULL) {
00043         sprintf(tmp, "This problem has no linear constraints");
00044         return tmp;
00045     }
00046
00047     abip_int n = spe->data->A->p[spe->data->A->n];
00048
00049     if (spe->stgs->linsys_solver == 0) {
00050         sprintf(tmp, "sparse-direct using MKL-DSS, nnz in A = %li", (long)n);
00051     } else if (spe->stgs->linsys_solver == 1) {
00052         sprintf(tmp, "sparse-direct using QDLDL, nnz in A = %li", (long)n);
00053     } else if (spe->stgs->linsys_solver == 2) {
00054         sprintf(tmp, "sparse-direct using sparse cholesky, nnz in A = %li",
00055             (long)n);
00056     } else if (spe->stgs->linsys_solver == 3) {
00057         sprintf(tmp, "sparse-indirect using pcg, nnz in A = %li", (long)n);
00058     } else if (spe->stgs->linsys_solver == 4) {
00059         sprintf(tmp, "sparse-direct using MKL-PARDISO, nnz in A = %li", (long)n);
00060     } else if (spe->stgs->linsys_solver == 5) {
00061         sprintf(tmp, "dense-direct using dense cholesky, nnz in A = %li", (long)n);
00062     } else {
00063         sprintf(tmp, "\nlinsys solver type error\n");
00064     }
00065     return tmp;
00066 }
00067
00071 char *ABIP(get_lin_sys_summary)(spe_problem *self, ABIPInfo *info) {
00072     char *str = (char *)abip_malloc(sizeof(char) * 128);
00073
00074     abip_int n = self->L->nnz_LDL;
00075     info->avg_linsys_time =
00076         self->L->total_solve_time / (info->admm_iter + 1) / 1e3;
00077
00078     if (self->stgs->linsys_solver == 3) {
00079         sprintf(str,
00080             "\tLin-sys: avg # CG iterations: %2.2f, avg solve time per admm "
00081             "iter: %1.2es\n",
00082             (abip_float)self->L->total_cg_iters / (info->admm_iter + 1),
00083             info->avg_linsys_time);
00084     } else {
00085         sprintf(str,
00086             "\tLin-sys: nnz in L factor: %li, avg solve time per admm iter: "
00087             "%1.2es\n",
00088             (long)n, info->avg_linsys_time);
00089     }
00090
00091     info->avg_cg_iters =

```

```

00092         (abip_float)self->L->total_cg_iters / (info->admm_iter + 1);
00093     self->L->total_solve_time = 0;
00094     self->L->total_cg_iters = 0;
00095
00096     return str;
00097 }
00098
00102 abip_int ABIP(validate_lin_sys)(const ABIPMatrix *A) {
00103     abip_int i;
00104     abip_int r_max;
00105     abip_int Annz;
00106
00107     if (A == ABIP_NULL) {
00108         return 0;
00109     }
00110
00111     if (!A->x || !A->i || !A->p) {
00112         abip_printf("ERROR: incomplete data!\n");
00113         return -1;
00114     }
00115
00116     for (i = 0; i < A->n; ++i) {
00117         if (A->p[i] == A->p[i + 1]) {
00118             abip_printf("WARN: the %li-th column empty!\n", (long)i);
00119         } else if (A->p[i] > A->p[i + 1]) {
00120             abip_printf("ERROR: the column pointers decreases!\n");
00121             return -1;
00122         }
00123     }
00124
00125     Annz = A->p[A->n];
00126     if (((abip_float)Annz / A->m > A->n) || (Annz <= 0)) {
00127         abip_printf(
00128             "ERROR: the number of nonzeros in A = %li, outside of valid range!\n",
00129             (long)Annz);
00130         return -1;
00131     }
00132
00133     r_max = 0;
00134     for (i = 0; i < Annz; ++i) {
00135         if (A->i[i] > r_max) {
00136             r_max = A->i[i];
00137         }
00138     }
00139     if (r_max > A->m - 1) {
00140         abip_printf(
00141             "ERROR: the number of rows in A is inconsistent with input "
00142             "dimension!\n");
00143         return -1;
00144     }
00145
00146     return 0;
00147 }
00148
00152 void ABIP(free_A_matrix)(ABIPMatrix *A) {
00153     if (A->x) {
00154         abip_free(A->x);
00155     }
00156     if (A->i) {
00157         abip_free(A->i);
00158     }
00159     if (A->p) {
00160         abip_free(A->p);
00161     }
00162
00163     abip_free(A);
00164 }
00165
00166 #if EXTRA_VERBOSE > 0
00167
00168 static void print_A_matrix(const ABIPMatrix *A) {
00169     abip_int i;
00170     abip_int j;
00171
00172     if (A->p[A->n] < 2500) {
00173         abip_printf("\n");
00174         for (i = 0; i < A->n; ++i) {
00175             abip_printf("Col %li: ", (long)i);
00176             for (j = A->p[i]; j < A->p[i + 1]; j++) {
00177                 abip_printf("A[%li,%li] = %4f, ", (long)A->i[j], (long)i, A->x[j]);
00178             }
00179             abip_printf("norm col = %4f\n",
00180                 ABIP(norm)(&(A->x[A->p[i]]), A->p[i + 1] - A->p[i]));
00181         }
00182         abip_printf("norm A = %4f\n", ABIP(norm)(A->x, A->p[A->n]));
00183     }
00184 }

```

```

00185 #endif
00186
00190 // y += A'*x
00191 void ABIP(accum_by_Atrans)(const ABIPMatrix *A, const abip_float *x,
00192                          abip_float *y) {
00193     abip_int p;
00194     abip_int j;
00195
00196     abip_int c1;
00197     abip_int c2;
00198     abip_float yj;
00199
00200 #if EXTRA_VERBOSE > 0
00201     ABIP(timer) mult_by_Atrans_timer;
00202     ABIP(tic) (&mult_by_Atrans_timer);
00203 #endif
00204
00205 #ifdef _OPENMP
00206 #pragma omp parallel for private(p, c1, c2, yj)
00207 #endif
00208
00209     for (j = 0; j < A->n; j++) {
00210         yj = y[j];
00211         c1 = A->p[j];
00212         c2 = A->p[j + 1];
00213         for (p = c1; p < c2; p++) {
00214             yj += A->x[p] * x[A->i[p]];
00215         }
00216         y[j] = yj;
00217     }
00218
00219 #if EXTRA_VERBOSE > 0
00220     abip_printf("mult By A trans time: %1.2e seconds. \n",
00221                ABIP(tocq)(&mult_by_Atrans_timer) / 1e3);
00222 #endif
00223 }
00224
00228 // y += A*x
00229 void ABIP(accum_by_A)(const ABIPMatrix *A, const abip_float *x, abip_float *y) {
00230     abip_int p;
00231     abip_int j;
00232
00233     abip_int c1;
00234     abip_int c2;
00235     abip_float xj;
00236
00237 #if EXTRA_VERBOSE > 0
00238     ABIP(timer) mult_by_A_timer;
00239     ABIP(tic) (&mult_by_A_timer);
00240 #endif
00241
00242 #ifdef _OPENMP
00243 #pragma omp parallel for private(p, c1, c2, xj)
00244     for (j = 0; j < n; j++) {
00245         xj = x[j];
00246         c1 = A->p[j];
00247         c2 = A->p[j + 1];
00248         for (p = c1; p < c2; p++) {
00249             #pragma omp atomic
00250             y[A->i[p]] += A->x[p] * xj;
00251         }
00252     }
00253 #endif
00254
00255     for (j = 0; j < A->n; j++) {
00256         xj = x[j];
00257         c1 = A->p[j];
00258         c2 = A->p[j + 1];
00259         for (p = c1; p < c2; p++) {
00260             y[A->i[p]] += A->x[p] * xj;
00261         }
00262     }
00263
00264 #if EXTRA_VERBOSE > 0
00265     abip_printf("mult By A time: %1.2e seconds \n",
00266                ABIP(tocq)(&mult_by_A_timer) / 1e3);
00267 #endif
00268 }
00269
00270 static abip_int _ldl_init(cs *A, abip_int *P, abip_float **info) {
00271     *info = (abip_float *)abip_calloc(AMD_INFO, sizeof(abip_float));
00272     return amd_order(A->n, A->p, A->i, P, (abip_float *)ABIP_NULL, *info);
00273 }
00274
00275 cs *permute_kkt(spe_problem *spe) {
00276     abip_float *info;
00277     abip_int *Pinv, amd_status, *idx_mapping, i;

```

```

00278     cs *kkt = spe->L->K;
00279     cs *kkt_perm;
00280     if (!kkt) {
00281         return ABIP_NULL;
00282     }
00283     amd_status = _ldl_init(kkt, spe->L->P, &info);
00284     if (amd_status < 0) {
00285         abip_printf("AMD permutatation error.\n");
00286         return ABIP_NULL;
00287     }
00288     #if VERBOSITY > 0
00289     abip_printf("Matrix factorization info:\n");
00290     amd_info(info);
00291     #endif
00292     Pinv = cs_pinv(spe->L->P, spe->L->K->n);
00293     kkt_perm = cs_symperm(kkt, Pinv, 1);
00294     abip_free(Pinv);
00295     abip_free(info);
00296     return kkt_perm;
00297 }
00298
00299 static void _ldl_perm(abip_int n, abip_float *x, abip_float *b, abip_int *P) {
00300     abip_int j;
00301     for (j = 0; j < n; j++) x[j] = b[P[j]];
00302 }
00303
00304 static void _ldl_permt(abip_int n, abip_float *x, abip_float *b, abip_int *P) {
00305     abip_int j;
00306     for (j = 0; j < n; j++) x[P[j]] = b[j];
00307 }
00308
00309 static void _ldl_solve(abip_float *b, cs *L, abip_float *Dinv, abip_int *P,
00310                      abip_float *bp) {
00311     /* solves PLDL'P' x = b for x */
00312     abip_int n = L->n;
00313     _ldl_perm(n, bp, b, P);
00314     QDLDL_solve(n, L->p, L->i, L->x, Dinv, bp);
00315     _ldl_permt(n, b, bp, P);
00316 }
00317
00318 _MKL_DSS_HANDLE_t init_mkl_work(cs *K) {
00319     _INTEGER_t error;
00320     MKL_INT create_opt = MKL_DSS_ZERO_BASED_INDEXING;
00321     MKL_INT order_opt = MKL_DSS_DEFAULTS;
00322     MKL_INT sym = MKL_DSS_SYMMETRIC;
00323     MKL_INT type = MKL_DSS_INDEFINITE;
00324
00325     _MKL_DSS_HANDLE_t handle;
00326
00327     error = dss_create(handle, create_opt);
00328     if (error != MKL_DSS_SUCCESS) goto printError;
00329
00330     error = dss_define_structure(handle, sym, K->p, K->m, K->n, K->i, K->p[K->n]);
00331     if (error != MKL_DSS_SUCCESS) goto printError;
00332
00333     error = dss_reorder(handle, order_opt, 0);
00334     if (error != MKL_DSS_SUCCESS) goto printError;
00335
00336     error = dss_factor_real(handle, type, K->x);
00337     if (error != MKL_DSS_SUCCESS) goto printError;
00338
00339     return handle;
00340 printError:
00341     printf("MKL-DSS returned error code %i\n", error);
00342     return -1;
00343 }
00344
00345 abip_int mkl_solve_linsys(_MKL_DSS_HANDLE_t handle, abip_float *b, abip_int n) {
00346     _INTEGER_t error;
00347     abip_int nrhs = 1;
00348     abip_float *solValues = (abip_float *)abip_malloc(n * sizeof(abip_float));
00349     MKL_INT opt = MKL_DSS_DEFAULTS;
00350     error = dss_solve_real(handle, opt, b, nrhs, solValues);
00351     if (error != MKL_DSS_SUCCESS) {
00352         printf("solve err");
00353         exit(1);
00354     }
00355
00356     memcpy(b, solValues, n * sizeof(abip_float));
00357     abip_free(solValues);
00358     return 0;
00359 }
00360
00361 abip_int init_pardiso(spe_problem *self) {
00362     MKL_INT PARDISO_PARAMS_LDL[64] = {
00363
00364         1,

```



```

00365      /* Non-default value */ 3,
00366      /* P Nested dissection */ 0, /* Reserved          */
00367      0,
00368      /* No CG                  */ 0,
00369      /* No user permutation */ 0, /* Overwriting      */
00370      0,
00371      /* Refinement report */ 0,
00372      /* Auto ItRef step   */ 0, /* Reserved          */
00373      8,
00374      /* Perturb            */ 0,
00375      /* Disable scaling    */ 0, /* No transpose      */
00376      0,
00377      /* Disable matching   */ 0,
00378      /* Report on pivots   */ 0, /* Output            */
00379      0,
00380      /* Output              */ 0,
00381      /* Output              */ -1, /* No report          */
00382      0,
00383      /* No report          */ 0,
00384      /* Output              */ 2, /* Pivoting            */
00385      0,
00386      /* nPosEigVals        */ 0,
00387      /* nNegEigVals        */ 0, /* Classic factorize */
00388      0,
00389      0,
00390      0, /* Matrix checker */
00391      0,
00392      0,
00393      0,
00394      0,
00395      0,
00396      0,
00397      0,
00398      1,
00399      /* 0-based solve      */ 0,
00400      0,
00401      0,
00402      0,
00403      0,
00404      0,
00405      0,
00406      0,
00407      0,
00408      0,
00409      0,
00410      0,
00411      0,
00412      0,
00413      0,
00414      0,
00415      0,
00416      0,
00417      0,
00418      0,
00419      0,
00420      /* No diagonal        */ 0,
00421      0,
00422      0,
00423      0,
00424      0,
00425      0,
00426      0,
00427      0};
00428
00429      cs *K = self->L->K;
00430      MKL_INT n = K->n;
00431      MKL_INT *ia = K->i;
00432      MKL_INT *ja = K->j;
00433      abip_float *a = K->x;
00434
00435      self->L->mtype = -2; /* Real symmetric matrix */
00436      /* RHS and solution vectors. */
00437      MKL_INT nrhs = 1; /* Number of right hand sides. */
00438      /* Internal solver memory pointer pt, */
00439      /* 32-bit: int pt[64]; 64-bit: long int pt[64] */
00440      /* or void *pt[64] should be OK on both architectures */
00441      // void *pt[64];
00442      /* Pardiso control parameters. */
00443      // MKL_INT iparm[64];
00444      MKL_INT phase;
00445      /* Auxiliary variables. */
00446      MKL_INT i;
00447
00448      /* -----*/
00449      /* .. Setup Pardiso control parameters. */
00450      /* -----*/
00451      for (i = 0; i < 64; i++) {

```

```

00452     self->L->iparm[i] = 0;
00453 }
00454
00455 for (i = 0; i < 64; i++) {
00456     self->L->iparm[i] = PARDISO_PARAMS_LDL[i];
00457 }
00458
00459 self->L->maxfct = 1; /* Maximum number of numerical factorizations. */
00460 self->L->mnum = 1; /* Which factorization to use. */
00461 self->L->msglvl = 1; /* Print statistical information in file */
00462 self->L->error = 0; /* Initialize error flag */
00463 /* -----*/
00464 /* .. Initialize the internal solver memory pointer. This is only */
00465 /* necessary for the FIRST call of the PARDISO solver. */
00466 /* -----*/
00467 for (i = 0; i < 64; i++) {
00468     self->L->pt[i] = 0;
00469 }
00470 /* -----*/
00471 /* .. Reordering and Symbolic Factorization. This step also allocates */
00472 /* all memory that is necessary for the factorization. */
00473 /* -----*/
00474 phase = 11;
00475 PARDISO(self->L->pt, &(self->L->maxfct), &(self->L->mnum), &(self->L->mtype),
00476         &phase, &n, a, ja, ia, &(self->L->idum), &nrhs, self->L->iparm,
00477         &(self->L->msglvl), &(self->L->ddum), &(self->L->ddum),
00478         &(self->L->error));
00479 if (self->L->error != 0) {
00480     printf("\nERROR during symbolic factorization: %i", self->L->error);
00481     exit(1);
00482 }
00483 printf("\nReordering completed ... ");
00484 printf("\nNumber of nonzeros in factors = %i", self->L->iparm[17]);
00485 printf("\nNumber of factorization MFLOPS = %i", self->L->iparm[18]);
00486 /* -----*/
00487 /* .. Numerical factorization. */
00488 /* -----*/
00489 phase = 22;
00490 PARDISO(self->L->pt, &(self->L->maxfct), &(self->L->mnum), &(self->L->mtype),
00491         &phase, &n, a, ja, ia, &(self->L->idum), &nrhs, self->L->iparm,
00492         &(self->L->msglvl), &(self->L->ddum), &(self->L->ddum),
00493         &(self->L->error));
00494 if (self->L->error != 0) {
00495     printf("\nERROR during numerical factorization: %i", self->L->error);
00496     exit(2);
00497 }
00498 printf("\nFactorization completed ... ");
00499
00500 return 0;
00501 }
00502
00503 abip_int pardiso_solve(spe_problem *self, abip_float *b, abip_int n) {
00504     /* -----*/
00505     /* .. Back substitution and iterative refinement. */
00506     /* -----*/
00507     cs *K = self->L->K;
00508     MKL_INT phase = 33;
00509     self->L->iparm[7] = 2; /* Max numbers of iterative refinement steps. */
00510     /* Set right hand side to one. */
00511     MKL_INT nrhs = 1; /* Number of right hand sides. */
00512     abip_float *x = (abip_float *)abip_malloc(n * sizeof(abip_float));
00513
00514     PARDISO(self->L->pt, &(self->L->maxfct), &(self->L->mnum), &(self->L->mtype),
00515             &phase, &n, K->x, K->p, K->i, &(self->L->idum), &nrhs, self->L->iparm,
00516             &(self->L->msglvl), b, x, &(self->L->error));
00517     if (self->L->error != 0) {
00518         printf("\nERROR during solution: %i", self->L->error);
00519         exit(3);
00520     }
00521     memcpy(b, x, n * sizeof(abip_float));
00522     abip_free(x);
00523
00524     return 0;
00525 }
00526
00527 abip_int pardiso_free(spe_problem *self) {
00528     cs *K = self->L->K;
00529     MKL_INT phase = -1;
00530     MKL_INT nrhs = 1;
00531     MKL_INT n = K->n;
00532     PARDISO(self->L->pt, &(self->L->maxfct), &(self->L->mnum), &(self->L->mtype),
00533             &phase, &n, &(self->L->ddum), K->p, K->i, &(self->L->idum), &nrhs,
00534             self->L->iparm, &(self->L->msglvl), &(self->L->ddum), &(self->L->ddum),
00535             &(self->L->error));
00536
00537     cs_spfree(K);
00538

```

```

00539     return 0;
00540 }
00541
00542 abip_int LDL_factor(cs *A, cs **L, abip_float *Dinv) {
00543     // data for elim tree calculation
00544     QDLDL_int *etree;
00545     QDLDL_int *Lnz;
00546     QDLDL_int sumLnz;
00547
00548     // working data for factorisation
00549     QDLDL_int *iwork;
00550     QDLDL_bool *bwork;
00551     QDLDL_float *fwork;
00552
00553     /*-----
00554      * pre-factorisation memory allocations
00555      *-----*/
00556
00557     // These can happen *before* the etree is calculated
00558     // since the sizes are not sparsity pattern specific
00559
00560     // For the elimination tree
00561     etree = (QDLDL_int *)malloc(sizeof(QDLDL_int) * A->n);
00562     Lnz = (QDLDL_int *)malloc(sizeof(QDLDL_int) * A->n);
00563
00564     // For the L factors. Li and Lx are sparsity dependent
00565     // so must be done after the etree is constructed
00566
00567     QDLDL_float *D = (QDLDL_float *)malloc(sizeof(QDLDL_float) * A->n);
00568     // (*Dinv) = (QDLDL_float*)malloc(sizeof(QDLDL_float) * A->n);
00569
00570     // Working memory. Note that both the etree and factor
00571     // calls requires a working vector of QDLDL_int, with
00572     // the factor function requiring 3*An elements and the
00573     // etree only An elements. Just allocate the larger
00574     // amount here and use it in both places
00575     iwork = (QDLDL_int *)malloc(sizeof(QDLDL_int) * (3 * A->n));
00576     bwork = (QDLDL_bool *)malloc(sizeof(QDLDL_bool) * A->n);
00577     fwork = (QDLDL_float *)malloc(sizeof(QDLDL_float) * A->n);
00578
00579     /*-----
00580      * elimination tree calculation
00581      *-----*/
00582     sumLnz = QDLDL_etree(A->n, A->p, A->i, iwork, Lnz, etree);
00583
00584     if (sumLnz < 0) {
00585         return sumLnz; // error
00586     }
00587     /*-----
00588      * LDL factorisation
00589      *-----*/
00590
00591     // First allocate memory for Li and Lx
00592     (*L) = cs_spallocc(A->n, A->n, sumLnz, 1, 0);
00593     // L->p = (QDLDL_int*)malloc(sizeof(QDLDL_int)*(A->n+1));
00594     // L->i = (QDLDL_int*)malloc(sizeof(QDLDL_int)*sumLnz);
00595     // L->x = (QDLDL_float*)malloc(sizeof(QDLDL_float)*sumLnz);
00596     // Dinv = (QDLDL_float*)malloc(sizeof(QDLDL_float)*A->n);
00597
00598     // now factor
00599     abip_int status =
00600         QDLDL_factor(A->n, A->p, A->i, A->x, (*L)->p, (*L)->i, (*L)->x, D, Dinv,
00601             Lnz, etree, bwork, iwork, fwork);
00602
00603     free(D);
00604     free(etree);
00605     free(Lnz);
00606     free(iwork);
00607     free(bwork);
00608     free(fwork);
00609
00610     return status;
00611 }
00612
00613 abip_int abip_cholsol(spe_problem *self, abip_float *b, abip_int n) {
00614     abip_float *x;
00615     x = cs_malloc(n, sizeof(abip_float)); /* get workspace */
00616     abip_int ok = (self->L->S && self->L->N && x);
00617     if (ok) {
00618         cs_ipvec(self->L->S->pinv, b, x, n); /* x = P*b */
00619         cs_lsolve(self->L->N->L, x); /* x = L\ x */
00620         cs_ltsolve(self->L->N->L, x); /* x = L'\ x */
00621         cs_pvec(self->L->S->pinv, x, b, n); /* b = P' * x */
00622     }
00623     cs_free(x);
00624     return (ok);
00625 }

```

```

00626
00630 abip_int pcg(spe_problem *self, abip_float *b, abip_float *x, abip_float rho_x,
00631             abip_int max_iter, abip_float tol) {
00632     /*
00633         x is used for warm start
00634         result overwrite b
00635     */
00636
00637     abip_int m = self->p;
00638     abip_int n = self->q;
00639
00640     abip_float *ATx = (abip_float *)abip_malloc(n * sizeof(abip_float));
00641     memset(ATx, 0, n * sizeof(abip_float));
00642     self->spe_AT_times(self, x, ATx);
00643     ABIP(scale_array)(ATx, -1, n);
00644
00645     abip_float *r = (abip_float *)abip_malloc(m * sizeof(abip_float));
00646     memcpy(r, b, m * sizeof(abip_float));
00647     self->spe_A_times(self, ATx, r);
00648     ABIP(add_scaled_array)(r, x, m, -rho_x);
00649
00650     abip_float *z = (abip_float *)abip_malloc(m * sizeof(abip_float));
00651     for (int k = 0; k < m; k++) {
00652         z[k] = self->L->M[k] * r[k];
00653     }
00654
00655     abip_float *p = (abip_float *)abip_malloc(m * sizeof(abip_float));
00656     memcpy(p, z, m * sizeof(abip_float));
00657
00658     abip_float ip = ABIP(dot)(r, z, m);
00659
00660     abip_int i;
00661     abip_float *Ap = (abip_float *)abip_malloc(m * sizeof(abip_float));
00662     abip_float alpha, ipold, beta;
00663
00664     memcpy(b, x, m * sizeof(abip_float));
00665
00666     for (i = 0; i < max_iter; i++) {
00667         memset(ATx, 0, n * sizeof(abip_float));
00668         self->spe_AT_times(self, p, ATx);
00669         memcpy(Ap, p, m * sizeof(abip_float));
00670         ABIP(scale_array)(Ap, rho_x, m);
00671         self->spe_A_times(self, ATx, Ap);
00672
00673         alpha = ip / (ABIP(dot)(Ap, p, m));
00674
00675         // ABIP(add_scaled_array)(x, p, m, alpha);
00676         ABIP(add_scaled_array)(b, p, m, alpha);
00677
00678         ABIP(add_scaled_array)(r, Ap, m, -alpha);
00679
00680         if (ABIP(norm)(r, m) < tol) {
00681             #if EXTRA_VERBOSE > 0
00682                 abip_printf(
00683                     "CG took %d iterations to converge, residual %4f <= tolerance %4f\n",
00684                     i, ABIP(norm)(r, m), tol);
00685             #endif
00686
00687             abip_free(ATx);
00688             abip_free(r);
00689             abip_free(z);
00690             abip_free(p);
00691             abip_free(Ap);
00692
00693             return i + 1;
00694         }
00695
00696         for (int j = 0; j < m; j++) {
00697             z[j] = self->L->M[j] * r[j];
00698         }
00699
00700         ipold = ip;
00701         ip = ABIP(dot)(z, r, m);
00702         beta = ip / ipold;
00703
00704         ABIP(scale_array)(p, beta, m);
00705         ABIP(add_scaled_array)(p, z, m, 1);
00706     }
00707
00708     printf(
00709         "CG did not converge within %d iterations, residual %4f > tolerance "
00710         "%4f\n",
00711         max_iter, ABIP(norm)(r, m), tol);
00712
00713     abip_free(ATx);
00714     abip_free(r);
00715     abip_free(z);

```

```

00716     abip_free(p);
00717     abip_free(Ap);
00718
00719     return i + 1;
00720 }
00721
00722 static void mat_vec(spe_problem *self, const abip_float *x, abip_float *y) {
00723     abip_int m = self->m;
00724     abip_int n = self->n;
00725     abip_int i;
00726
00727     memcpy(y, x, n * sizeof(abip_float));
00728
00729     for (i = 0; i < n; i++) {
00730         y[i] *= self->rho_dr[i + m];
00731     }
00732
00733     if (self->Q != ABIP_NULL) {
00734         ABIP(accum_by_A)(self->Q, x, y);
00735     }
00736
00737     abip_float *tem = (abip_float *)abip_malloc(sizeof(abip_float) * m);
00738     memset(tem, 0, m * sizeof(abip_float));
00739     ABIP(accum_by_A)(self->A, x, tem);
00740     for (i = 0; i < m; i++) {
00741         tem[i] /= self->rho_dr[i];
00742     }
00743
00744     ABIP(accum_by_Atrans)(self->A, tem, y);
00745
00746     abip_free(tem);
00747 }
00748
00749 abip_int qcp_pcg(spe_problem *self, abip_float *b, abip_float *x,
00750                 abip_int max_iter, abip_float tol) {
00751     /*
00752      * x is used for warm start
00753      * result overwrite b
00754      */
00755
00756     abip_int m = self->m;
00757     abip_int n = self->n;
00758     abip_int i, j;
00759
00760     abip_float ztr, ztr_prev, alpha;
00761     abip_float *p =
00762         (abip_float *)abip_calloc(n, sizeof(abip_float)); /* cg direction */
00763     abip_float *Gp = (abip_float *)abip_calloc(
00764         n, sizeof(abip_float)); /* updated CG direction */
00765     abip_float *r =
00766         (abip_float *)abip_calloc(n, sizeof(abip_float)); /* cg residual */
00767     abip_float *z = (abip_float *)abip_calloc(n, sizeof(abip_float));
00768     /* for preconditioning */
00769
00770     if (x == ABIP_NULL) {
00771         /* no warm_start, take x = 0 */
00772         /* r = b */
00773         memcpy(r, b, n * sizeof(abip_float));
00774         /* b = 0 */
00775         memset(b, 0, n * sizeof(abip_float));
00776     } else {
00777         /* r = Mat * s */
00778         mat_vec(self, x, r);
00779         /* r = Mat * s - b */
00780         ABIP(add_scaled_array)(r, b, n, -1.);
00781         /* r = b - Mat * s */
00782         ABIP(scale_array)(r, -1., n);
00783         /* b = s */
00784         memcpy(b, x, n * sizeof(abip_float));
00785     }
00786
00787     /* check to see if we need to run CG at all */
00788     if (ABIP(norm_inf)(r, n) < MAX(tol, 1e-12)) {
00789         abip_free(p);
00790         abip_free(Gp);
00791         abip_free(r);
00792         abip_free(z);
00793         return 0;
00794     }
00795
00796     /* z = M r (M is inverse preconditioner) */
00797     memcpy(z, r, n * sizeof(abip_float));
00798     ABIP(c_dot)(z, self->L->M, n);
00799
00800     /* ztr = z' r */
00801     ztr = ABIP(dot)(z, r, n);
00802     /* p = z */
00803     memcpy(p, z, n * sizeof(abip_float));

```

```

00809
00810     for (i = 0; i < max_iter; ++i) {
00811         /* Gp = Mat * p */
00812         mat_vec(self, p, Gp);
00813         /* alpha = z'r / p'G p */
00814         alpha = ztr / ABIP(dot)(p, Gp, n);
00815         /* b += alpha * p */
00816         ABIP(add_scaled_array)(b, p, n, alpha);
00817         /* r -= alpha * G p */
00818         ABIP(add_scaled_array)(r, Gp, n, -alpha);
00819
00820         if (ABIP(norm_inf)(r, n) < tol) {
00821             break;
00822         }
00823         /* z = M r (M is inverse preconditioner) */
00824         memcpy(z, r, n * sizeof(abip_float));
00825         ABIP(c_dot)(z, self->L->M, n);
00826
00827         ztr_prev = ztr;
00828         /* ztr = z'r */
00829         ztr = ABIP(dot)(z, r, n);
00830         /* p = beta * p, where beta = ztr / ztr_prev */
00831         ABIP(scale_array)(p, ztr / ztr_prev, n);
00832         /* p = z + beta * p */
00833         ABIP(add_scaled_array)(p, z, n, 1.);
00834     }
00835
00836     printf("tol: %.4e, resid: %.4e, iters: %li\n", tol, ABIP(norm_inf)(r, n),
00837           (long)i + 1);
00838     if (i == max_iter - 1) {
00839         printf(
00840             "CG did not converge within %d iterations, residual %4f > tolerance "
00841             "%4f\n",
00842             max_iter, ABIP(norm)(r, m), tol);
00843     }
00844
00845     abip_free(p);
00846     abip_free(Gp);
00847     abip_free(r);
00848     abip_free(z);
00849
00850     return i + 1;
00851 }
00852
00853 static void svm_mat_vec(spe_problem *self, const abip_float *x, abip_float *y) {
00854     abip_int m = self->p;
00855     abip_int n = self->q;
00856     abip_int i;
00857
00858     memcpy(y, x, m * sizeof(abip_float));
00859
00860     for (i = 0; i < m; i++) {
00861         y[i] *= self->rho_dr[i];
00862     }
00863
00864     abip_float *tem = (abip_float *)abip_malloc(sizeof(abip_float) * n);
00865     memset(tem, 0, n * sizeof(abip_float));
00866     self->spe_AT_times(self, x, tem);
00867
00868     abip_float *H = (abip_float *)abip_malloc(sizeof(abip_float) * n);
00869     for (i = 0; i < self->q; i++) {
00870         if (i < self->n)
00871             H[i] = self->stgs->rho_x + self->Q->x[i];
00872         else
00873             H[i] = self->stgs->rho_x;
00874     }
00875
00876     for (i = 0; i < n; i++) {
00877         // tem[i] /= self->H[i];
00878         tem[i] /= H[i];
00879     }
00880
00881     self->spe_A_times(self, tem, y);
00882
00883     abip_free(tem);
00884     abip_free(H);
00885 }
00886
00887 abip_int svmpc_pcg(spe_problem *self, abip_float *b, abip_float *x,
00888                   abip_int max_iter, abip_float tol) {
00889     /*
00890      * x is used for warm start
00891      * result overwrite b
00892      */
00893
00894     // abip_int m = self->p;
00895     abip_int n = self->q;

```

```

00903     abip_int i, j;
00904
00905     abip_float ztr, ztr_prev, alpha;
00906     abip_float *p =
00907         (abip_float *)abip_calloc(n, sizeof(abip_float)); /* cg direction */
00908     abip_float *Gp = (abip_float *)abip_calloc(
00909         n, sizeof(abip_float)); /* updated CG direction */
00910     abip_float *r =
00911         (abip_float *)abip_calloc(n, sizeof(abip_float)); /* cg residual */
00912     abip_float *z = (abip_float *)abip_calloc(n, sizeof(abip_float));
00913     /* for preconditioning */
00914
00915     if (x == ABIP_NULL) {
00916         /* no warm_start, take x = 0 */
00917         /* r = b */
00918         memcpy(r, b, n * sizeof(abip_float));
00919         /* b = 0 */
00920         memset(b, 0, n * sizeof(abip_float));
00921     } else {
00922         /* r = Mat * s */
00923         svm_mat_vec(self, x, r);
00924         /* r = Mat * s - b */
00925         ABIP(add_scaled_array)(r, b, n, -1.);
00926         /* r = b - Mat * s */
00927         ABIP(scale_array)(r, -1., n);
00928         /* b = s */
00929         memcpy(b, x, n * sizeof(abip_float));
00930     }
00931     /* check to see if we need to run CG at all */
00932     if (ABIP(norm_inf)(r, n) < MAX(tol, 1e-12)) {
00933         abip_free(p);
00934         abip_free(Gp);
00935         abip_free(r);
00936         abip_free(z);
00937         return 0;
00938     }
00939
00940     /* z = M r (M is inverse preconditioner) */
00941     memcpy(z, r, n * sizeof(abip_float));
00942     ABIP(c_dot)(z, self->L->M, n);
00943
00944     /* ztr = z' r */
00945     ztr = ABIP(dot)(z, r, n);
00946     /* p = z */
00947     memcpy(p, z, n * sizeof(abip_float));
00948
00949     for (i = 0; i < max_iter; ++i) {
00950         /* Gp = Mat * p */
00951         svm_mat_vec(self, p, Gp);
00952         /* alpha = z' r / p' G p */
00953         alpha = ztr / ABIP(dot)(p, Gp, n);
00954         /* b += alpha * p */
00955         ABIP(add_scaled_array)(b, p, n, alpha);
00956         /* r -= alpha * G p */
00957         ABIP(add_scaled_array)(r, Gp, n, -alpha);
00958
00959         if (ABIP(norm_inf)(r, n) < tol) {
00960             break;
00961         }
00962         /* z = M r (M is inverse preconditioner) */
00963         memcpy(z, r, n * sizeof(abip_float));
00964         ABIP(c_dot)(z, self->L->M, n);
00965
00966         ztr_prev = ztr;
00967         /* ztr = z' r */
00968         ztr = ABIP(dot)(z, r, n);
00969         /* p = beta * p, where beta = ztr / ztr_prev */
00970         ABIP(scale_array)(p, ztr / ztr_prev, n);
00971         /* p = z + beta * p */
00972         ABIP(add_scaled_array)(p, z, n, 1.);
00973     }
00974
00975     printf("tol: %.4e, resid: %.4e, iters: %li\n", tol, ABIP(norm_inf)(r, n),
00976         (long)i + 1);
00977     if (i == max_iter - 1) {
00978         printf(
00979             "CG did not converge within %d iterations, residual %4f > tolerance "
00980             "%4f\n",
00981             max_iter, ABIP(norm)(r, n), tol);
00982     }
00983
00984     abip_free(p);
00985     abip_free(Gp);
00986     abip_free(r);
00987     abip_free(z);
00988
00989     return i + 1;

```

```

00990 }
00991
00995 abip_int init_dense_chol(spe_problem *spe) {
00996     // K is CSC format and only stores upper triangle
00997     spe->L->U = ABIP(csc_to_dense)(spe->L->K, ColMajor);
00998     abip_int info;
00999     abip_int n = spe->L->K->n;
01000
01001     info = LAPACKE_dpotrf(LAPACK_COL_MAJOR, 'U', n, spe->L->U, n);
01002     return info; // 0 if successful
01003 }
01004
01005 abip_int dense_chol_sol(spe_problem *spe, abip_float *b, abip_int n) {
01006     abip_int info;
01007
01008     info = LAPACKE_dpotrs(LAPACK_COL_MAJOR, 'U', n, 1, spe->L->U, n, b, n);
01009     return info; // 0 if successful
01010 }
01011
01012 abip_int dense_chol_free(spe_problem *spe) {
01013     if (spe->L->U) {
01014         abip_free(spe->L->U);
01015     }
01016     if (spe->L) {
01017         abip_free(spe->L);
01018     }
01019
01020     return 0;
01021 }
01022 /*-----*/
01023
01027 abip_int ABIP(init_linsys_work)(spe_problem *spe) {
01028     if (spe->stgs->linsys_solver == 3) { // pcg
01029         spe->L->S = ABIP_NULL;
01030         spe->L->N = ABIP_NULL;
01031         spe->L->handle = ABIP_NULL;
01032         spe->L->Dinv = ABIP_NULL;
01033         spe->L->L = ABIP_NULL;
01034         spe->L->P = ABIP_NULL;
01035         spe->L->U = ABIP_NULL;
01036
01037         spe->L->total_solve_time = 0.0;
01038         spe->L->total_cg_iters = 0;
01039
01040         return 0;
01041     }
01042
01043     cs *K = spe->L->K;
01044     spe->L->nnz_LDL = K->nzmax; // K is NULL ptr if using pcg
01045
01046     printf("\nStarting decomposition, with\nn = %d, m = %d , nnz = %d\n", K->m,
01047           K->n, K->nzmax);
01048
01049     if (spe->stgs->linsys_solver == 0) { // mkl_dss
01050         spe->L->handle = init_mkl_work(K);
01051         if (spe->L->handle == -1) {
01052             printf("\nerror in LDL factorization using MKL-DSS\n");
01053             return -1;
01054         }
01055         cs_spfree(K);
01056         spe->L->Dinv = ABIP_NULL;
01057         spe->L->L = ABIP_NULL;
01058         spe->L->P = ABIP_NULL;
01059         spe->L->M = ABIP_NULL;
01060         spe->L->S = ABIP_NULL;
01061         spe->L->N = ABIP_NULL;
01062         spe->L->U = ABIP_NULL;
01063
01064         spe->L->total_solve_time = 0.0;
01065         return 0;
01066     }
01067
01068     else if (spe->stgs->linsys_solver == 1) { // qdldl
01069
01070         spe->L->Dinv = (abip_float *)abip_malloc(K->n * sizeof(abip_float));
01071         spe->L->bp = (abip_float *)abip_malloc(K->n * sizeof(abip_float));
01072         spe->L->P = (abip_int *)abip_malloc(K->n * sizeof(abip_int));
01073         cs *kkt_perm = permute_kkt(spe);
01074         cs_spfree(K);
01075         if (LDL_factor(kkt_perm, &(spe->L->L), spe->L->Dinv) < 0) {
01076             spe->L->L = ABIP_NULL;
01077             printf("\nerror in LDL factorization using QDLDL\n");
01078             return -1;
01079         }
01080         cs_spfree(kkt_perm);
01081
01082         spe->L->handle = ABIP_NULL;

```



```

01083     spe->L->M = ABIP_NULL;
01084     spe->L->S = ABIP_NULL;
01085     spe->L->N = ABIP_NULL;
01086     spe->L->U = ABIP_NULL;
01087
01088     spe->L->total_solve_time = 0.0;
01089     return 0;
01090
01091 } else if (spe->stgs->linsys_solver == 2) { // cholesky
01092     spe->L->S = cs_schol(l, K);
01093     spe->L->N = cs_chol(K, spe->L->S);
01094     cs_spfree(K);
01095     spe->L->handle = ABIP_NULL;
01096     spe->L->Dinv = ABIP_NULL;
01097     spe->L->L = ABIP_NULL;
01098     spe->L->P = ABIP_NULL;
01099     spe->L->M = ABIP_NULL;
01100     spe->L->U = ABIP_NULL;
01101
01102     spe->L->total_solve_time = 0.0;
01103     return 0;
01104 } else if (spe->stgs->linsys_solver == 4) { // mkl_pardiso
01105     init_pardiso(spe);
01106     spe->L->Dinv = ABIP_NULL;
01107     spe->L->handle = ABIP_NULL;
01108     spe->L->L = ABIP_NULL;
01109     spe->L->P = ABIP_NULL;
01110     spe->L->M = ABIP_NULL;
01111     spe->L->S = ABIP_NULL;
01112     spe->L->N = ABIP_NULL;
01113     spe->L->U = ABIP_NULL;
01114
01115     spe->L->total_solve_time = 0.0;
01116     return 0;
01117 } else if (spe->stgs->linsys_solver == 5) { // lapack dense chol
01118
01119     init_dense_chol(spe);
01120     spe->L->handle = ABIP_NULL;
01121     spe->L->Dinv = ABIP_NULL;
01122     spe->L->L = ABIP_NULL;
01123     spe->L->P = ABIP_NULL;
01124     spe->L->M = ABIP_NULL;
01125     spe->L->S = ABIP_NULL;
01126     spe->L->N = ABIP_NULL;
01127
01128     spe->L->total_solve_time = 0.0;
01129     return 0;
01130 }
01131
01132 else {
01133     printf("\nlinsys solver type error\n");
01134     return -1;
01135 }
01136 }
01137
01141 abip_int ABIP(solve_linsys)(spe_problem *spe, abip_float *b, abip_int n,
01142                        abip_float *pcg_warm_start, abip_float pcg_tol) {
01143     if (spe->stgs->linsys_solver == 0) { // mkl_dss
01144         mkl_solve_linsys(spe->L->handle, b, n);
01145         return 0;
01146     } else if (spe->stgs->linsys_solver == 1) { // qdldl
01147         // QDLDL_solve(spe->L->L->n, spe->L->L->p, spe->L->L->i, spe->L->L->x, spe->L->Dinv, b);
01148         // return 0;
01149
01150         // for new ld1
01151         _ldl_solve(b, spe->L->L, spe->L->Dinv, spe->L->P, spe->L->bp);
01152         return 0;
01153     } else if (spe->stgs->linsys_solver == 2) { // cholesky
01154         abip_cholsol(spe, b, n);
01155         return 0;
01156     } else if (spe->stgs->linsys_solver == 3) { // PCG
01157
01158         if (spe->stgs->prob_type == 3) {
01159             // return qcp_pcg(spe, b, pcg_warm_start, n, pcg_tol);
01160             return qcp_pcg(spe, b, pcg_warm_start, n, pcg_tol);
01161         } else if (spe->stgs->prob_type == 4) {
01162             return svmqp_pcg(spe, b, pcg_warm_start, n, pcg_tol);
01163         } else {
01164             return pcg(spe, b, pcg_warm_start, spe->stgs->rho_y, n, pcg_tol);
01165         }
01166     } else if (spe->stgs->linsys_solver == 4) { // mkl_pardiso
01167         pardiso_solve(spe, b, n);
01168         return 0;
01169     } else if (spe->stgs->linsys_solver == 5) { // lapack dense chol
01170         dense_chol_sol(spe, b, n);
01171         return 0;
01172     } else {

```

```

01173     printf("\nlinsys solver type error\n");
01174     return -1;
01175 }
01176 }
01177
01181 abip_int ABIP(free_linsys)(spe_problem *spe) {
01182     if (spe->L) {
01183         if (spe->stgs->linsys_solver == 0) { // mkl_dss
01184             MKL_INT opt = MKL_DSS_DEFAULTS;
01185             dss_delete(spe->L->handle, opt);
01186             return 0;
01187         } else if (spe->stgs->linsys_solver == 1) { // qdldl
01188             if (spe->L->Dinv) abip_free(spe->L->Dinv);
01189             if (spe->L->L) cs_spfree(spe->L->L);
01190             if (spe->L->P) abip_free(spe->L->P);
01191             if (spe->L->bp) abip_free(spe->L->bp);
01192             return 0;
01193         } else if (spe->stgs->linsys_solver == 2) { // cholesky
01194             if (spe->L->S) cs_sfree(spe->L->S);
01195             if (spe->L->N) cs_nfree(spe->L->N);
01196             return 0;
01197         } else if (spe->stgs->linsys_solver == 3) { // pcg
01198             if (spe->L->M) abip_free(spe->L->M);
01199             return 0;
01200         } else if (spe->stgs->linsys_solver == 4) { // mkl_pardiso
01201             pardiso_free(spe);
01202             return 0;
01203         } else if (spe->stgs->linsys_solver == 5) { // lapack dense chol
01204             dense_chol_free(spe);
01205             return 0;
01206         }
01207     }
01208     else {
01209         printf("\nlinsys solver type error\n");
01210         return -1;
01211     }
01212 }
01213 }

```

## 4.189 source/qcp\_config.c File Reference

```
#include "qcp_config.h"
```

### Macros

- #define [MIN\\_SCALE](#) (1e-3)
- #define [MAX\\_SCALE](#) (1e3)

### Functions

- [abip\\_int init\\_qcp](#) ([qcp \\*\\*self](#), [ABIPData \\*d](#), [ABIPSettings \\*stgs](#))  
*Initialize the qcp problem structure.*
- [void qcp\\_A\\_times](#) ([qcp \\*self](#), [const abip\\_float \\*x](#), [abip\\_float \\*y](#))  
*Matrix-vector multiplication for the general qcp problem with A untransposed.*
- [void qcp\\_AT\\_times](#) ([qcp \\*self](#), [const abip\\_float \\*x](#), [abip\\_float \\*y](#))  
*Matrix-vector multiplication for the general qcp problem with A transposed.*
- [void scaling\\_qcp\\_data](#) ([qcp \\*self](#), [ABIPConc \\*k](#))  
*Scale the data for the qcp problem.*
- [void un\\_scaling\\_qcp\\_sol](#) ([qcp \\*self](#), [ABIPSolution \\*sol](#))  
*Get the unscaled solution of the general qcp problem.*
- [abip\\_float qcp\\_inner\\_conv\\_check](#) ([qcp \\*self](#), [ABIPWork \\*w](#))  
*Check whether the inner loop of the genral qcp problem has converged.*

- void `calc_qcp_residuals` (`qcp *self`, `ABIPWork *w`, `ABIPResiduals *r`, `abip_int ipm_iter`, `abip_int admm_iter`)  
*Calculate the residuals of the general qcp problem.*
- `cs * form_qcp_kkt` (`qcp *self`)  
*Formulate the qcp KKT matrix of the general qcp problem.*
- void `init_qcp_precon` (`qcp *self`)  
*Initialize the preconditioner of conjugate gradient method for the general qcp problem.*
- `abip_float get_qcp_pcg_tol` (`abip_int k`, `abip_float error_ratio`, `abip_float norm_p`)  
*Get the tolerance of the conjugate gradient method for the general qcp problem.*
- `abip_int init_qcp_linsys_work` (`qcp *self`)  
*Initialize the linear system solver work space for the general qcp problem.*
- `abip_int solve_qcp_linsys` (`qcp *self`, `abip_float *b`, `abip_float *pcg_warm_start`, `abip_int iter`, `abip_float error_ratio`)  
*Linear system solver for the general qcp problem.*
- void `free_qcp_linsys_work` (`qcp *self`)  
*Free the linear system solver work space for the general qcp problem.*

## 4.189.1 Macro Definition Documentation

### 4.189.1.1 MAX\_SCALE

```
#define MAX_SCALE (1e3)
```

Definition at line 3 of file `qcp_config.c`.

### 4.189.1.2 MIN\_SCALE

```
#define MIN_SCALE (1e-3)
```

Definition at line 2 of file `qcp_config.c`.

## 4.189.2 Function Documentation

### 4.189.2.1 calc\_qcp\_residuals()

```
void calc_qcp_residuals (
    qcp * self,
    ABIPWork * w,
    ABIPResiduals * r,
    abip_int ipm_iter,
    abip_int admm_iter )
```

Calculate the residuals of the general qcp problem.

Definition at line 562 of file `qcp_config.c`.

#### 4.189.2.2 form\_qcp\_kkt()

```
cs * form_qcp_kkt (
    qcp * self )
```

Formulate the qcp KKT matrix of the general qcp problem.

Definition at line 699 of file [qcp\\_config.c](#).

#### 4.189.2.3 free\_qcp\_linsys\_work()

```
void free_qcp_linsys_work (
    qcp * self )
```

Free the linear system solver work space for the general qcp problem.

Definition at line 886 of file [qcp\\_config.c](#).

#### 4.189.2.4 get\_qcp\_pcg\_tol()

```
abip_float get_qcp_pcg_tol (
    abip_int k,
    abip_float error_ratio,
    abip_float norm_p )
```

Get the tolerance of the conjugate gradient method for the general qcp problem.

Definition at line 786 of file [qcp\\_config.c](#).

#### 4.189.2.5 init\_qcp()

```
abip_int init_qcp (
    qcp ** self,
    ABIPData * d,
    ABIPSettings * stgs )
```

Initialize the qcp problem structure.

Definition at line 8 of file [qcp\\_config.c](#).

#### 4.189.2.6 init\_qcp\_linsys\_work()

```
abip_int init_qcp_linsys_work (
    qcp * self )
```

Initialize the linear system solver work space for the general qcp problem.

Definition at line 799 of file [qcp\\_config.c](#).

#### 4.189.2.7 init\_qcp\_precon()

```
void init_qcp_precon (
    qcp * self )
```

Initialize the preconditioner of conjugate gradient method for the general qcp problem.

Definition at line 754 of file [qcp\\_config.c](#).

#### 4.189.2.8 qcp\_A\_times()

```
void qcp_A_times (
    qcp * self,
    const abip_float * x,
    abip_float * y )
```

Matrix-vector multiplication for the general qcp problem with A untransposed.

Definition at line 72 of file [qcp\\_config.c](#).

#### 4.189.2.9 qcp\_AT\_times()

```
void qcp_AT_times (
    qcp * self,
    const abip_float * x,
    abip_float * y )
```

Matrix-vector multiplication for the general qcp problem with A transposed.

Definition at line 82 of file [qcp\\_config.c](#).

#### 4.189.2.10 qcp\_inner\_conv\_check()

```
abip_float qcp_inner_conv_check (
    qcp * self,
    ABIPWork * w )
```

Check whether the inner loop of the genral qcp problem has converged.

Definition at line 518 of file [qcp\\_config.c](#).

#### 4.189.2.11 scaling\_qcp\_data()

```
void scaling_qcp_data (
    qcp * self,
    ABIPCone * k )
```

Scale the data for the qcp problem.

Definition at line 91 of file [qcp\\_config.c](#).

#### 4.189.2.12 solve\_qcp\_linsys()

```
abip_int solve_qcp_linsys (
    qcp * self,
    abip_float * b,
    abip_float * pcg_warm_start,
    abip_int iter,
    abip_float error_ratio )
```

Linear system solver for the general qcp problem.

Definition at line 826 of file [qcp\\_config.c](#).

#### 4.189.2.13 un\_scaling\_qcp\_sol()

```
void un_scaling_qcp_sol (
    qcp * self,
    ABIPSolution * sol )
```

Get the unscaled solution of the general qcp problem.

Definition at line 496 of file [qcp\\_config.c](#).

## 4.190 qcp\_config.c

[Go to the documentation of this file.](#)

```

00001 #include "qcp_config.h"
00002 #define MIN_SCALE (1e-3)
00003 #define MAX_SCALE (1e3)
00004
00008 abip_int init_qcp(qcp **self, ABIPData *d, ABIPSettings *stgs) {
00009     qcp *this_qcp = (qcp *)abip_malloc(sizeof(qcp));
00010     *self = this_qcp;
00011
00012     this_qcp->m = d->m;
00013     this_qcp->n = d->n;
00014     this_qcp->p = d->p;
00015     this_qcp->q = d->q;
00016     abip_int m = this_qcp->p;
00017     abip_int n = this_qcp->q;
00018
00019     if (d->A == ABIP_NULL) {
00020         this_qcp->sparsity = 0;
00021     } else {
00022         this_qcp->sparsity = ((d->A->p[d->n] / (d->m * d->n)) < 0.05);
00023     }
00024
00025     // non-identity DR scaling
00026     this_qcp->rho_dr =
00027         (abip_float *)abip_malloc((m + n + 1) * sizeof(abip_float));
00028     for (int i = 0; i < m + n + 1; i++) {
00029         if (i < m) {
00030             this_qcp->rho_dr[i] = stgs->rho_y;
00031         } else if (i < m + n) {
00032             this_qcp->rho_dr[i] = stgs->rho_x;
00033         } else {
00034             this_qcp->rho_dr[i] = stgs->rho_tau;
00035         }
00036     }
00037
00038     this_qcp->L = (ABIPLinSysWork *)abip_malloc(sizeof(ABIPLinSysWork));
00039     this_qcp->pro_type = QCP;
00040     this_qcp->stgs = stgs;
00041     this_qcp->data = d;
00042     if (d->A != ABIP_NULL) {
00043         this_qcp->A = (ABIPMatrix *)abip_malloc(sizeof(ABIPMatrix));
00044     }
00045     if (d->Q != ABIP_NULL) {
00046         this_qcp->Q = (ABIPMatrix *)abip_malloc(sizeof(ABIPMatrix));
00047     }
00048     if (d->b != ABIP_NULL) {
00049         this_qcp->b = (abip_float *)abip_malloc(this_qcp->p * sizeof(abip_float));
00050     }
00051     this_qcp->c = (abip_float *)abip_malloc(this_qcp->q * sizeof(abip_float));
00052     this_qcp->D = (abip_float *)abip_malloc(this_qcp->m * sizeof(abip_float));
00053     this_qcp->E = (abip_float *)abip_malloc(this_qcp->n * sizeof(abip_float));
00054
00055     this_qcp->scaling_data = &scaling_qcp_data;
00056     this_qcp->un_scaling_sol = &un_scaling_qcp_sol;
00057     this_qcp->calc_residuals = &calc_qcp_residuals;
00058     this_qcp->init_spe_linsys_work = &init_qcp_linsys_work;
00059     this_qcp->solve_spe_linsys = &solve_qcp_linsys;
00060     this_qcp->free_spe_linsys_work = &free_qcp_linsys_work;
00061     this_qcp->spe_A_times = &qcp_A_times;
00062     this_qcp->spe_AT_times = &qcp_AT_times;
00063     this_qcp->inner_conv_check = &qcp_inner_conv_check;
00064
00065     return 0;
00066 }
00067
00072 void qcp_A_times(qcp *self, const abip_float *x, abip_float *y) {
00073     if (self->A != ABIP_NULL) {
00074         ABIP(accum_by_A)(self->A, x, y);
00075     }
00076 }
00077
00082 void qcp_AT_times(qcp *self, const abip_float *x, abip_float *y) {
00083     if (self->A != ABIP_NULL) {
00084         ABIP(accum_by_Atrans)(self->A, x, y);
00085     }
00086 }
00087
00091 void scaling_qcp_data(qcp *self, ABIPConc *k) {
00092     if (self->data->b == ABIP_NULL) {
00093         self->b = ABIP_NULL;
00094     } else {
00095         memcpy(self->b, self->data->b, self->m * sizeof(abip_float));
00096     }
}

```

```

00097
00098 memcpy(self->c, self->data->c, self->n * sizeof(abip_float));
00099
00100 if (self->data->A == ABIP_NULL) {
00101     self->A = ABIP_NULL;
00102 } else if (!ABIP(copy_A_matrix)(&(self->A), self->data->A)) {
00103     abip_printf("ERROR: copy A matrix failed\n");
00104     RETURN;
00105 }
00106
00107 if (self->data->Q == ABIP_NULL) {
00108     self->Q = ABIP_NULL;
00109 } else if (!ABIP(copy_A_matrix)(&(self->Q), self->data->Q)) {
00110     abip_printf("ERROR: copy Q matrix failed\n");
00111     RETURN;
00112 }
00113
00114 abip_int m = self->m;
00115 abip_int n = self->n;
00116 ABIPMatrix *A = self->A;
00117 ABIPMatrix *Q = self->Q;
00118
00119 abip_float min_row_scale = MIN_SCALE * SQRTF((abip_float)n);
00120 abip_float max_row_scale = MAX_SCALE * SQRTF((abip_float)n);
00121 abip_float min_col_scale = MIN_SCALE * SQRTF((abip_float)m);
00122 abip_float max_col_scale = MAX_SCALE * SQRTF((abip_float)m);
00123
00124 abip_float *E_hat = self->E;
00125 abip_float *D_hat = self->D;
00126
00127 for (int i = 0; i < n; i++) {
00128     E_hat[i] = 1;
00129 }
00130 for (int i = 0; i < m; i++) {
00131     D_hat[i] = 1;
00132 }
00133
00134 abip_float *E = (abip_float *)abip_malloc(n * sizeof(abip_float));
00135 memset(E, 0, n * sizeof(abip_float));
00136
00137 abip_float *E1 = (abip_float *)abip_malloc(n * sizeof(abip_float));
00138 memset(E1, 0, n * sizeof(abip_float));
00139
00140 abip_float *E2 = (abip_float *)abip_malloc(n * sizeof(abip_float));
00141 memset(E2, 0, n * sizeof(abip_float));
00142
00143 abip_float *D = (abip_float *)abip_malloc(m * sizeof(abip_float));
00144 memset(D, 0, m * sizeof(abip_float));
00145
00146 abip_int origin_scaling = self->stgs->origin_scaling;
00147 abip_int ruiz_scaling = self->stgs->ruiz_scaling;
00148 abip_int pc_scaling = self->stgs->pc_scaling;
00149 abip_int count;
00150 abip_float mean_E;
00151
00152 if (A == ABIP_NULL && Q == ABIP_NULL) {
00153     origin_scaling = 0;
00154     ruiz_scaling = 0;
00155     pc_scaling = 0;
00156 }
00157
00158 if (ruiz_scaling) {
00159     abip_int n_ruiz = 10;
00160
00161     for (int ruiz_iter = 0; ruiz_iter < n_ruiz; ruiz_iter++) {
00162         count = 0;
00163         memset(E, 0, n * sizeof(abip_float));
00164         memset(E1, 0, n * sizeof(abip_float));
00165         memset(E2, 0, n * sizeof(abip_float));
00166         memset(D, 0, m * sizeof(abip_float));
00167
00168         if (A != ABIP_NULL) {
00169             for (int j = 0; j < n; j++) {
00170                 if (A->p[j] == A->p[j + 1]) {
00171                     E1[j] = 0;
00172                 } else {
00173                     E1[j] =
00174                         SQRTF(ABIP(norm_inf)(&A->x[A->p[j]], A->p[j + 1] - A->p[j]));
00175                 }
00176             }
00177         }
00178
00179         if (Q != ABIP_NULL) {
00180             for (int j = 0; j < n; j++) {
00181                 if (Q->p[j] == Q->p[j + 1]) {
00182                     E2[j] = 0;
00183                 } else {

```



```

00184         E2[j] =
00185             SQRTRF(ABIP(norm_inf)(&Q->x[Q->p[j]], Q->p[j + 1] - Q->p[j]));
00186     }
00187 }
00188 }
00189
00190 for (int i = 0; i < n; i++) {
00191     E[i] = E1[i] < E2[i] ? E2[i] : E1[i];
00192 }
00193
00194 if (k->q) {
00195     for (int i = 0; i < k->qsize; i++) {
00196         mean_E = ABIP(vec_mean)(&E[count], k->q[i]);
00197         for (int j = 0; j < k->q[i]; j++) {
00198             E[j + count] = mean_E;
00199         }
00200         count += k->q[i];
00201     }
00202 }
00203
00204 if (k->rq) {
00205     for (int i = 0; i < k->rqsize; i++) {
00206         mean_E = ABIP(vec_mean)(&E[count], k->rq[i]);
00207         for (int j = 0; j < k->rq[i]; j++) {
00208             E[j + count] = mean_E;
00209         }
00210         count += k->rq[i];
00211     }
00212 }
00213
00214 if (A != ABIP_NULL) {
00215     for (int i = 0; i < A->p[n]; i++) {
00216         if (D[A->i[i]] < ABS(A->x[i])) {
00217             D[A->i[i]] = ABS(A->x[i]);
00218         }
00219     }
00220     for (int i = 0; i < m; i++) {
00221         D[i] = SQRTRF(D[i]);
00222         if (D[i] < min_row_scale)
00223             D[i] = 1;
00224         else if (D[i] > max_row_scale)
00225             D[i] = max_row_scale;
00226     }
00227
00228     for (int i = 0; i < n; i++) {
00229         if (E[i] < min_col_scale)
00230             E[i] = 1;
00231         else if (E[i] > max_col_scale)
00232             E[i] = max_col_scale;
00233         for (int j = A->p[i]; j < A->p[i + 1]; j++) {
00234             A->x[j] /= E[i];
00235         }
00236     }
00237 }
00238
00239 if (Q != ABIP_NULL) {
00240     for (int i = 0; i < n; i++) {
00241         for (int j = Q->p[i]; j < Q->p[i + 1]; j++) {
00242             Q->x[j] /= E[i];
00243         }
00244     }
00245     for (int i = 0; i < Q->p[n]; i++) {
00246         Q->x[i] /= E[Q->i[i]];
00247     }
00248 }
00249
00250 if (A != ABIP_NULL) {
00251     for (int i = 0; i < A->p[n]; i++) {
00252         A->x[i] /= D[A->i[i]];
00253     }
00254 }
00255
00256 for (int i = 0; i < n; i++) {
00257     E_hat[i] *= E[i];
00258 }
00259
00260 for (int i = 0; i < m; i++) {
00261     D_hat[i] *= D[i];
00262 }
00263 }
00264 }
00265
00266 if (origin_scaling) {
00267     memset(E, 0, n * sizeof(abip_float));
00268     memset(E1, 0, n * sizeof(abip_float));
00269     memset(E2, 0, n * sizeof(abip_float));
00270     memset(D, 0, m * sizeof(abip_float));

```

```

00271
00272     count = 0;
00273
00274     if (A != ABIP_NULL) {
00275         for (int i = 0; i < n; i++) {
00276             for (int j = A->p[i]; j < A->p[i + 1]; j++) {
00277                 E1[i] += A->x[j] * A->x[j];
00278             }
00279             E1[i] = SQRTF(E1[i]);
00280         }
00281     }
00282
00283     if (Q != ABIP_NULL) {
00284         for (int i = 0; i < n; i++) {
00285             for (int j = Q->p[i]; j < Q->p[i + 1]; j++) {
00286                 E2[i] += Q->x[j] * Q->x[j];
00287             }
00288             E2[i] = SQRTF(E2[i]);
00289         }
00290     }
00291
00292     for (int i = 0; i < n; i++) {
00293         E[i] = SQRTF(E1[i] < E2[i] ? E2[i] : E1[i]);
00294     }
00295
00296     if (k->q) {
00297         for (int i = 0; i < k->qsize; i++) {
00298             mean_E = ABIP(vec_mean)(&E[count], k->q[i]);
00299             for (int j = 0; j < k->q[i]; j++) {
00300                 E[j + count] = mean_E;
00301             }
00302             count += k->q[i];
00303         }
00304     }
00305
00306     if (k->rq) {
00307         for (int i = 0; i < k->rqsize; i++) {
00308             mean_E = ABIP(vec_mean)(&E[count], k->rq[i]);
00309             for (int j = 0; j < k->rq[i]; j++) {
00310                 E[j + count] = mean_E;
00311             }
00312             count += k->rq[i];
00313         }
00314     }
00315
00316     if (A != ABIP_NULL) {
00317         for (int i = 0; i < A->p[n]; i++) {
00318             D[A->i[i]] += A->x[i] * A->x[i];
00319         }
00320         for (int i = 0; i < m; i++) {
00321             D[i] = SQRTF(SQRTF(D[i]));
00322             if (D[i] < min_row_scale)
00323                 D[i] = 1;
00324             else if (D[i] > max_row_scale)
00325                 D[i] = max_row_scale;
00326         }
00327
00328         for (int i = 0; i < n; i++) {
00329             if (E[i] < min_col_scale)
00330                 E[i] = 1;
00331             else if (E[i] > max_col_scale)
00332                 E[i] = max_col_scale;
00333             for (int j = A->p[i]; j < A->p[i + 1]; j++) {
00334                 A->x[j] /= E[i];
00335             }
00336         }
00337     }
00338
00339     if (Q != ABIP_NULL) {
00340         for (int i = 0; i < n; i++) {
00341             for (int j = Q->p[i]; j < Q->p[i + 1]; j++) {
00342                 Q->x[j] /= E[i];
00343             }
00344         }
00345         for (int i = 0; i < Q->p[n]; i++) {
00346             Q->x[i] /= E[Q->i[i]];
00347         }
00348     }
00349
00350     if (A != ABIP_NULL) {
00351         for (int i = 0; i < A->p[n]; i++) {
00352             A->x[i] /= D[A->i[i]];
00353         }
00354     }
00355
00356     for (int i = 0; i < n; i++) {
00357         E_hat[i] *= E[i];

```

```

00358     }
00359
00360     for (int i = 0; i < m; i++) {
00361         D_hat[i] *= D[i];
00362     }
00363 }
00364
00365 if (pc_scaling) {
00366     memset(E, 0, n * sizeof(abip_float));
00367     memset(E1, 0, n * sizeof(abip_float));
00368     memset(E2, 0, n * sizeof(abip_float));
00369     memset(D, 0, m * sizeof(abip_float));
00370     count = 0;
00371     abip_float alpha_pc = 1;
00372     if (A != ABIP_NULL) {
00373         for (int i = 0; i < n; i++) {
00374             for (int j = A->p[i]; j < A->p[i + 1]; j++) {
00375                 E1[i] += POWF(ABS(A->x[j]), alpha_pc);
00376             }
00377             E1[i] = SQRTF(POWF(E1[i], 1 / alpha_pc));
00378         }
00379     }
00380     if (Q != ABIP_NULL) {
00381         for (int i = 0; i < n; i++) {
00382             for (int j = Q->p[i]; j < Q->p[i + 1]; j++) {
00383                 E2[i] += POWF(ABS(Q->x[j]), alpha_pc);
00384             }
00385             E2[i] = SQRTF(POWF(E2[i], 1 / alpha_pc));
00386         }
00387     }
00388
00389     for (int i = 0; i < n; i++) {
00390         E[i] = E1[i] < E2[i] ? E2[i] : E1[i];
00391     }
00392
00393     if (k->q) {
00394         for (int i = 0; i < k->qsize; i++) {
00395             mean_E = ABIP(vec_mean)(&E[count], k->q[i]);
00396             for (int j = 0; j < k->q[i]; j++) {
00397                 E[j + count] = mean_E;
00398             }
00399             count += k->q[i];
00400         }
00401     }
00402
00403     if (k->rq) {
00404         for (int i = 0; i < k->rqsize; i++) {
00405             mean_E = ABIP(vec_mean)(&E[count], k->rq[i]);
00406             for (int j = 0; j < k->rq[i]; j++) {
00407                 E[j + count] = mean_E;
00408             }
00409             count += k->rq[i];
00410         }
00411     }
00412
00413     if (A != ABIP_NULL) {
00414         for (int i = 0; i < A->p[n]; i++) {
00415             D[A->i[i]] += POWF(ABS(A->x[i]), 2 - alpha_pc);
00416         }
00417         for (int i = 0; i < m; i++) {
00418             D[i] = SQRTF(POWF(D[i], 1 / (2 - alpha_pc)));
00419             if (D[i] < min_row_scale)
00420                 D[i] = 1;
00421             else if (D[i] > max_row_scale)
00422                 D[i] = max_row_scale;
00423         }
00424
00425         for (int i = 0; i < n; i++) {
00426             if (E[i] < min_col_scale)
00427                 E[i] = 1;
00428             else if (E[i] > max_col_scale)
00429                 E[i] = max_col_scale;
00430             for (int j = A->p[i]; j < A->p[i + 1]; j++) {
00431                 A->x[j] /= E[i];
00432             }
00433         }
00434     }
00435
00436     if (Q != ABIP_NULL) {
00437         for (int i = 0; i < n; i++) {
00438             for (int j = Q->p[i]; j < Q->p[i + 1]; j++) {
00439                 Q->x[j] /= E[i];
00440             }
00441         }
00442         for (int i = 0; i < Q->p[n]; i++) {
00443             Q->x[i] /= E[Q->i[i]];
00444         }

```

```

00445     }
00446
00447     if (A != ABIP_NULL) {
00448         for (int i = 0; i < A->p[n]; i++) {
00449             A->x[i] /= D[A->i[i]];
00450         }
00451     }
00452
00453     for (int i = 0; i < n; i++) {
00454         E_hat[i] *= E[i];
00455     }
00456
00457     for (int i = 0; i < m; i++) {
00458         D_hat[i] *= D[i];
00459     }
00460 }
00461
00462 abip_float sc =
00463     SQRTRF(SQRTRF(ABIP(norm_sq)(self->c, n) + ABIP(norm_sq)(self->b, m)));
00464
00465 if (self->b != ABIP_NULL) {
00466     for (int i = 0; i < m; i++) {
00467         self->b[i] /= D_hat[i];
00468     }
00469 }
00470
00471 for (int i = 0; i < n; i++) {
00472     self->c[i] /= E_hat[i];
00473 }
00474
00475 if (sc < MIN_SCALE)
00476     sc = 1;
00477 else if (sc > MAX_SCALE)
00478     sc = MAX_SCALE;
00479 self->sc_b = 1 / sc;
00480 self->sc_c = 1 / sc;
00481
00482 if (self->b != ABIP_NULL) {
00483     ABIP(scale_array)(self->b, self->sc_b * self->stgs->scale, m);
00484 }
00485 ABIP(scale_array)(self->c, self->sc_c * self->stgs->scale, n);
00486
00487 abip_free(E);
00488 abip_free(E1);
00489 abip_free(E2);
00490 abip_free(D);
00491 }
00492
00496 void un_scaling_qcp_sol(qcp *self, ABIPSolution *sol) {
00497     abip_int i;
00498
00499     abip_float *D = self->D;
00500     abip_float *E = self->E;
00501
00502     for (i = 0; i < self->n; ++i) {
00503         sol->x[i] /= (E[i] * self->sc_b);
00504     }
00505
00506     for (i = 0; i < self->m; ++i) {
00507         sol->y[i] /= (D[i] * self->sc_c);
00508     }
00509
00510     for (i = 0; i < self->n; ++i) {
00511         sol->s[i] *= E[i] / (self->sc_c * self->stgs->scale);
00512     }
00513 }
00514
00518 abip_float qcp_inner_conv_check(qcp *self, ABIPWork *w) {
00519     abip_int m = self->p;
00520     abip_int n = self->q;
00521
00522     abip_float *Qu = (abip_float *)abip_malloc((m + n + 1) * sizeof(abip_float));
00523     abip_float *Mu = (abip_float *)abip_malloc((m + n) * sizeof(abip_float));
00524
00525     memset(Mu, 0, (m + n) * sizeof(abip_float));
00526
00527     self->spe_A_times(self, &w->u[m], Mu);
00528
00529     self->spe_AT_times(self, w->u, &Mu[m]);
00530     ABIP(scale_array)(&Mu[m], -1, n);
00531
00532     if (self->Q != ABIP_NULL) {
00533         ABIP(accum_by_A)(self->Q, &w->u[m], &Mu[m]);
00534     }
00535
00536     memcpy(Qu, Mu, (m + n) * sizeof(abip_float));
00537

```

```

00538     ABIP(add_scaled_array)(Qu, self->b, m, w->u[m + n]);
00539     ABIP(add_scaled_array)(&Qu[m], self->c, n, w->u[m + n]);
00540
00541     Qu[m + n] = -ABIP(dot)(w->u, Mu, m + n) / w->u[m + n] +
00542         ABIP(dot)(w->u, self->b, m) - ABIP(dot)(&w->u[m], self->c, n);
00543
00544     abip_float *tem = (abip_float *)abip_malloc((m + n + 1) * sizeof(abip_float));
00545     memcpy(tem, Qu, (m + n + 1) * sizeof(abip_float));
00546     ABIP(add_scaled_array)(tem, w->v_origin, m + n + 1, -1);
00547
00548     abip_float error_inner =
00549         ABIP(norm)(tem, m + n + 1) /
00550         (1 + ABIP(norm)(Qu, m + n + 1) + ABIP(norm)(w->v_origin, m + n + 1));
00551
00552     abip_free(Qu);
00553     abip_free(Mu);
00554     abip_free(tem);
00555
00556     return error_inner;
00557 }
00558
00562 void calc_qcp_residuals(qcp *self, ABIPWork *w, ABIPResiduals *r,
00563     abip_int ipm_iter, abip_int admm_iter) {
00564     DEBUG_FUNC
00565
00566     abip_int n = w->n;
00567     abip_int m = w->m;
00568
00569     abip_float *y = (abip_float *)abip_malloc(m * sizeof(abip_float));
00570     abip_float *x = (abip_float *)abip_malloc(n * sizeof(abip_float));
00571     abip_float *s = (abip_float *)abip_malloc(n * sizeof(abip_float));
00572
00573     abip_float this_pr;
00574     abip_float this_dr;
00575     abip_float this_gap;
00576
00577     if (admm_iter && r->last_admm_iter == admm_iter) {
00578         RETURN;
00579     }
00580
00581     r->last_ipm_iter = ipm_iter;
00582     r->last_admm_iter = admm_iter;
00583
00584     r->tau = ABS(w->u[n + m]);
00585     r->kap =
00586         ABS(w->v_origin[n + m]) /
00587         (self->stgs->normalize * (self->stgs->scale * self->sc_c * self->sc_b)
00588          : 1);
00589
00590     memcpy(y, w->u, m * sizeof(abip_float));
00591     memcpy(x, &w->u[m], n * sizeof(abip_float));
00592     memcpy(s, &w->v_origin[m], n * sizeof(abip_float));
00593
00594     ABIP(scale_array)(y, 1 / r->tau, m);
00595     ABIP(scale_array)(x, 1 / r->tau, n);
00596     ABIP(scale_array)(s, 1 / r->tau, n);
00597
00598     abip_float *Ax = (abip_float *)abip_malloc(m * sizeof(abip_float));
00599     abip_float *Ax_b = (abip_float *)abip_malloc(m * sizeof(abip_float));
00600
00601     memset(Ax, 0, m * sizeof(abip_float));
00602     self->spe_A_times(self, x, Ax);
00603
00604     memcpy(Ax_b, Ax, m * sizeof(abip_float));
00605     ABIP(add_scaled_array)(Ax_b, self->b, m, -1);
00606
00607     // for scs cg tol
00608     r->Ax_b_norm = ABIP(norm_inf)(Ax_b, m);
00609
00610     ABIP(c_dot)(Ax, self->D, m);
00611     ABIP(c_dot)(Ax_b, self->D, m);
00612
00613     this_pr = ABIP(norm_inf)(Ax_b, m) /
00614         (self->sc_b + MAX(ABIP(norm_inf)(Ax, m), self->sc_b * w->nm_inf_b));
00615
00616     abip_float *Qx = (abip_float *)abip_malloc(n * sizeof(abip_float));
00617     abip_float *ATy = (abip_float *)abip_malloc(n * sizeof(abip_float));
00618     abip_float *Qx_ATy_c_s = (abip_float *)abip_malloc(n * sizeof(abip_float));
00619
00620     memset(Qx, 0, n * sizeof(abip_float));
00621     abip_float xQx_2 = 0;
00622
00623     if (self->Q != ABIP_NULL) {
00624         ABIP(accum_by_A)(self->Q, x, Qx);
00625         xQx_2 = ABIP(dot)(x, Qx, n) / (2 * self->sc_b * self->sc_c);
00626     }
00627

```

```

00628  memset(ATy, 0, n * sizeof(abip_float));
00629  self->spe_AT_times(self, y, ATy);
00630
00631  memcpy(Qx_ATy_c_s, Qx, n * sizeof(abip_float));
00632  ABIP(add_scaled_array)(Qx_ATy_c_s, ATy, n, -1);
00633  ABIP(add_scaled_array)(Qx_ATy_c_s, self->c, n, 1);
00634  ABIP(add_scaled_array)(Qx_ATy_c_s, s, n, -1);
00635
00636  r->Qx_ATy_c_s_norm = ABIP(norm_inf)(Qx_ATy_c_s, n);
00637
00638  ABIP(c_dot)(Qx, self->E, n);
00639  ABIP(c_dot)(ATy, self->E, n);
00640  ABIP(c_dot)(Qx_ATy_c_s, self->E, n);
00641  ABIP(c_dot)(s, self->E, n);
00642
00643  this_dr = ABIP(norm_inf)(Qx_ATy_c_s, n) /
00644            (self->sc_c + MAX(self->sc_c * w->nm_inf_c, ABIP(norm_inf)(Qx, n)));
00645
00646  abip_float cTx = ABIP(dot)(self->c, x, n) / (self->sc_b * self->sc_c);
00647  abip_float bTy = ABIP(dot)(self->b, y, m) / (self->sc_b * self->sc_c);
00648
00649  this_gap = ABS(2 * xQx_2 + cTx - bTy) /
00650            (1 + MAX(2 * xQx_2, MAX(ABS(cTx), ABS(bTy))));
00651
00652  r->pobj = xQx_2 + cTx;
00653  r->dobj = -xQx_2 + bTy;
00654
00655  r->res_dif = MAX(MAX(ABS(this_pr - r->res_pri), ABS(this_dr - r->res_dual)),
00656                ABS(this_gap - r->rel_gap));
00657  r->res_pri = this_pr;
00658  r->res_dual = this_dr;
00659  r->rel_gap = this_gap;
00660  r->error_ratio =
00661      MAX(r->res_pri / self->stgs->eps_p,
00662          MAX(r->res_dual / self->stgs->eps_d, r->rel_gap / self->stgs->eps_g));
00663
00664  if (ABIP(dot)(self->c, &w->u[m], n) < 0) {
00665      ABIP(scale_array)(Qx, r->tau, n);
00666      ABIP(scale_array)(Ax, r->tau, m);
00667      r->res_unbdd = MAX(ABIP(norm)(Qx, n), ABIP(norm)(Ax, m)) /
00668                    (-ABIP(dot)(self->c, &w->u[m], n));
00669  } else {
00670      r->res_unbdd = INFINITY;
00671  }
00672
00673  if (ABIP(dot)(self->b, w->u, m) > 0) {
00674      ABIP(scale_array)(ATy, r->tau, n);
00675      ABIP(scale_array)(s, r->tau, n);
00676      ABIP(add_scaled_array)(ATy, s, n, 1);
00677
00678      r->res_infeas = ABIP(norm)(ATy, n) / ABIP(dot)(self->b, w->u, m);
00679  } else {
00680      r->res_infeas = INFINITY;
00681  }
00682
00683  abip_free(x);
00684  abip_free(y);
00685  abip_free(s);
00686  abip_free(Ax);
00687  abip_free(Ax_b);
00688  abip_free(Qx);
00689  abip_free(ATy);
00690  abip_free(Qx_ATy_c_s);
00691 }
00692
00696 /*K = -rho_dr(1:m)*I -A
00697          Q + rho_dr(m+1:m+n)*I
00698 */
00699 cs *form_qcp_kkt(qcp *self) {
00700     abip_int m = self->m;
00701     abip_int n = self->n;
00702     ABIPMatrix *A = self->A;
00703     ABIPMatrix *Q = self->Q;
00704
00705     abip_int nnzA = A == ABIP_NULL ? 0 : A->p[A->n];
00706     ;
00707     abip_int nnzQ = Q == ABIP_NULL ? 0 : Q->p[Q->n];
00708     abip_int nnzK = m + nnzA + nnzQ + n;
00709     abip_int i;
00710     abip_int j;
00711     abip_float tem;
00712
00713     cs *K = cs_spalloc(m + n, m + n, nnzK, 1, 1);
00714
00715     for (i = 0; i < m; i++) {
00716         cs_entry(K, i, i, -self->rho_dr[i]);
00717     }

```

```

00718
00719     if (A != ABIP_NULL) {
00720         for (i = 0; i < n; i++) {
00721             for (j = A->p[i]; j < A->p[i + 1]; j++) {
00722                 cs_entry(K, A->i[j], m + i, -A->x[j]);
00723             }
00724         }
00725     }
00726
00727     for (i = 0; i < n; i++) {
00728         if (Q == ABIP_NULL || Q->p[i] == Q->p[i + 1]) {
00729             cs_entry(K, m + i, m + i, self->rho_dr[m + i]);
00730         } else {
00731             for (j = Q->p[i]; j < Q->p[i + 1]; j++) {
00732                 if (Q->i[j] > i)
00733                     tem = 0;
00734                 else if (Q->i[j] == i)
00735                     tem = Q->x[j] + self->rho_dr[m + i];
00736                 else
00737                     tem = Q->x[j];
00738                 cs_entry(K, m + Q->i[j], m + i, tem);
00739             }
00740         }
00741     }
00742
00743     cs *K_csc = cs_compress(K);
00744     cs_spfree(K);
00745     cs_dropzeros(K_csc);
00746
00747     return K_csc;
00748 }
00749
00754 void init_qcp_precon(qcp *self) {
00755     self->L->M = (abip_float *)abip_malloc(self->q * sizeof(abip_float));
00756     memset(self->L->M, 0, self->q * sizeof(abip_float));
00757
00758     for (int i = 0; i < self->q; i++) {
00759         for (int j = self->A->p[i]; j < self->A->p[i + 1]; j++) {
00760             self->L->M[i] +=
00761                 self->A->x[j] * self->A->x[j] / self->rho_dr[self->A->i[j]];
00762         }
00763     }
00764
00765     if (self->Q != ABIP_NULL) {
00766         for (int i = 0; i < self->q; i++) {
00767             for (int j = self->Q->p[i]; j < self->Q->p[i + 1]; j++) {
00768                 if (i == self->Q->i[j]) {
00769                     self->L->M[i] += self->Q->x[j];
00770                     break;
00771                 }
00772             }
00773         }
00774     }
00775
00776     ABIP(add_scaled_array)(self->L->M, &self->rho_dr[self->p], self->q, 1);
00777     for (int i = 0; i < self->q; i++) {
00778         self->L->M[i] = 1.0 / self->L->M[i];
00779     }
00780 }
00781
00786 abip_float get_qcp_pcg_tol(abip_int k, abip_float error_ratio,
00787                             abip_float norm_p) {
00788     if (k == -1) {
00789         return 1e-9 * norm_p;
00790     } else {
00791         return MAX(1e-9, 1e-5 * norm_p / POWF((k + 1), 2));
00792     }
00793 }
00794
00799 abip_int init_qcp_linsys_work(qcp *self) {
00800     if (self->stgs->linsys_solver == 0) { // mkl-dss need lower triangle
00801         self->L->K = cs_transpose(form_qcp_kkt(self), 1);
00802     } else if (self->stgs->linsys_solver == 1) { // qdldl need upper triangle
00803         self->L->K = form_qcp_kkt(self);
00804     } else if (self->stgs->linsys_solver == 2) { // cholesky need upper triangle
00805         self->L->K = form_qcp_kkt(self);
00806     } else if (self->stgs->linsys_solver == 3) { // pcg doesn't need kkt matrix
00807         init_qcp_precon(self);
00808         self->L->K = ABIP_NULL;
00809     } else if (self->stgs->linsys_solver ==
00810                 4) { // mkl-pardiso need lower triangle
00811         self->L->K = cs_transpose(form_qcp_kkt(self), 1);
00812     } else if (self->stgs->linsys_solver ==
00813                 5) { // dense cholesky need upper triangle
00814         self->L->K = form_qcp_kkt(self);
00815     } else {
00816         printf("\nlinsys solver type error\n");

```

```

00817     return -1;
00818 }
00819
00820 return ABIP(init_linsys_work)(self);
00821 }
00822
00826 abip_int solve_qcp_linsys(qcp *self, abip_float *b, abip_float *pcg_warm_start,
00827                          abip_int iter, abip_float error_ratio) {
00828     ABIP(timer) linsys_timer;
00829     ABIP(tic)(&linsys_timer);
00830
00831     if (self->stgs->linsys_solver == 3) { // pcg
00832
00833         abip_int n = self->n;
00834         abip_int m = self->m;
00835         abip_int i;
00836
00837         abip_float norm_p = ABIP(norm)(&b[m], n);
00838
00839         abip_float *tem = (abip_float *)abip_malloc(sizeof(abip_float) * m);
00840         memcpy(tem, b, m * sizeof(abip_float));
00841         for (i = 0; i < m; i++) {
00842             tem[i] /= self->rho_dr[i];
00843         }
00844         self->spe_AT_times(self, tem, &b[m]);
00845
00846         abip_free(tem);
00847
00848         abip_float pcg_tol = get_qcp_pcg_tol(iter, error_ratio, norm_p);
00849
00850         abip_int cg_its;
00851         if (iter == -1) {
00852             cg_its = ABIP(solve_linsys)(self, &b[m], n, pcg_warm_start, error_ratio);
00853         } else {
00854             cg_its =
00855                 ABIP(solve_linsys)(self, &b[m], n, &pcg_warm_start[m], error_ratio);
00856         }
00857
00858         if (iter >= 0) {
00859             self->L->total_cg_iters += cg_its;
00860         }
00861
00862         ABIP(scale_array)(b, -1, m);
00863         self->spe_A_times(self, &b[m], b);
00864         for (i = 0; i < m; i++) {
00865             b[i] /= -self->rho_dr[i];
00866         }
00867
00868     } else { // direct methods
00869
00870         abip_int n = self->n + self->m;
00871
00872         // difference here
00873         ABIP(scale_array)(b, -1, self->m);
00874
00875         ABIP(solve_linsys)(self, b, n, ABIP_NULL, 0);
00876     }
00877
00878     self->L->total_solve_time += ABIP(tocq)(&linsys_timer);
00879
00880     return 0;
00881 }
00882
00886 void free_qcp_linsys_work(qcp *self) { ABIP(free_linsys)(self); }

```

## 4.191 source/svm\_config.c File Reference

```
#include "svm_config.h"
```

### Macros

- #define `MIN_SCALE` (1e-3)
- #define `MAX_SCALE` (1e3)



## Functions

- [abip\\_int init\\_svm](#) ([svm](#) \*\*self, [ABIPData](#) \*d, [ABIPSettings](#) \*stgs)  
*Initialize the svm socp formulation structure.*
- void [svm\\_A\\_times](#) ([svm](#) \*self, const [abip\\_float](#) \*x, [abip\\_float](#) \*y)  
*Customized matrix-vector multiplication for the svm socp formulation with A untransposed.*
- void [svm\\_AT\\_times](#) ([svm](#) \*self, const [abip\\_float](#) \*x, [abip\\_float](#) \*y)  
*Customized matrix-vector multiplication for the svm socp formulation with A transposed.*
- [abip\\_float svm\\_inner\\_conv\\_check](#) ([svm](#) \*self, [ABIPWork](#) \*w)  
*Check whether the inner loop of the svm socp formulation has converged.*
- void [scaling\\_svm\\_data](#) ([svm](#) \*self, [ABIPConc](#) \*k)  
*Customized scaling procedure for the svm socp formulation.*
- void [un\\_scaling\\_svm\\_sol](#) ([svm](#) \*self, [ABIPSolution](#) \*sol)  
*Get the unscaled solution of the original svm problem.*
- void [calc\\_svm\\_residuals](#) ([svm](#) \*self, [ABIPWork](#) \*w, [ABIPResiduals](#) \*r, [abip\\_int](#) ipm\_iter, [abip\\_int](#) admm\_iter)  
*Calculate the residuals of the svm socp formulation.*
- [cs \\* form\\_svm\\_kkt](#) ([svm](#) \*self)  
*Formulate the qcp KKT matrix of the svm socp formulation.*
- void [init\\_svm\\_precon](#) ([svm](#) \*self)  
*Initialize the preconditioner of conjugate gradient method for the svm socp formulation.*
- [abip\\_float get\\_svm\\_pcg\\_tol](#) ([abip\\_int](#) k, [abip\\_float](#) error\_ratio, [abip\\_float](#) norm\_p)  
*Get the tolerance of the conjugate gradient method for the svm socp formulation.*
- [abip\\_int init\\_svm\\_linsys\\_work](#) ([svm](#) \*self)  
*Initialize the linear system solver work space for the svm socp formulation.*
- [abip\\_int solve\\_svm\\_linsys](#) ([svm](#) \*self, [abip\\_float](#) \*b, [abip\\_float](#) \*pcg\_warm\_start, [abip\\_int](#) iter, [abip\\_float](#) error\_ratio)  
*Customized linear system solver for the svm socp formulation.*
- void [free\\_svm\\_linsys\\_work](#) ([svm](#) \*self)  
*Free the linear system solver work space for the svm socp formulation.*

### 4.191.1 Macro Definition Documentation

#### 4.191.1.1 MAX\_SCALE

```
#define MAX_SCALE (1e3)
```

Definition at line 3 of file [svm\\_config.c](#).

#### 4.191.1.2 MIN\_SCALE

```
#define MIN_SCALE (1e-3)
```

Definition at line 2 of file [svm\\_config.c](#).

## 4.191.2 Function Documentation

### 4.191.2.1 `calc_svm_residuals()`

```
void calc_svm_residuals (
    svm * self,
    ABIPWork * w,
    ABIPResiduals * r,
    abip_int ipm_iter,
    abip_int admm_iter )
```

Calculate the residuals of the svm socp formulation.

Definition at line 445 of file [svm\\_config.c](#).

### 4.191.2.2 `form_svm_kkt()`

```
cs * form_svm_kkt (
    svm * self )
```

Formulate the qcp KKT matrix of the svm socp formulation.

Definition at line 577 of file [svm\\_config.c](#).

### 4.191.2.3 `free_svm_linsys_work()`

```
void free_svm_linsys_work (
    svm * self )
```

Free the linear system solver work space for the svm socp formulation.

Definition at line 812 of file [svm\\_config.c](#).

### 4.191.2.4 `get_svm_pcg_tol()`

```
abip_float get_svm_pcg_tol (
    abip_int k,
    abip_float error_ratio,
    abip_float norm_p )
```

Get the tolerance of the conjugate gradient method for the svm socp formulation.

Definition at line 669 of file [svm\\_config.c](#).

#### 4.191.2.5 init\_svm()

```
abip_int init_svm (
    svm ** self,
    ABIPData * d,
    ABIPSettings * stgs )
```

Initialize the svm socp formulation structure.

Definition at line 8 of file [svm\\_config.c](#).

#### 4.191.2.6 init\_svm\_linsys\_work()

```
abip_int init_svm_linsys_work (
    svm * self )
```

Initialize the linear system solver work space for the svm socp formulation.

Definition at line 702 of file [svm\\_config.c](#).

#### 4.191.2.7 init\_svm\_precon()

```
void init_svm_precon (
    svm * self )
```

Initialize the preconditioner of conjugate gradient method for the svm socp formulation.

Definition at line 642 of file [svm\\_config.c](#).

#### 4.191.2.8 scaling\_svm\_data()

```
void scaling_svm_data (
    svm * self,
    ABIPCone * k )
```

Customized scaling procedure for the svm socp formulation.

Definition at line 283 of file [svm\\_config.c](#).

#### 4.191.2.9 solve\_svm\_linsys()

```
abip_int solve_svm_linsys (
    svm * self,
    abip_float * b,
    abip_float * pcg_warm_start,
    abip_int iter,
    abip_float error_ratio )
```

Customized linear system solver for the svm socp formulation.

Definition at line 728 of file [svm\\_config.c](#).

#### 4.191.2.10 svm\_A\_times()

```
void svm_A_times (
    svm * self,
    const abip_float * x,
    abip_float * y )
```

Customized matrix-vector multiplication for the svm socp formulation with A untransposed.

Definition at line 175 of file [svm\\_config.c](#).

#### 4.191.2.11 svm\_AT\_times()

```
void svm_AT_times (
    svm * self,
    const abip_float * x,
    abip_float * y )
```

Customized matrix-vector multiplication for the svm socp formulation with A transposed.

Definition at line 202 of file [svm\\_config.c](#).

#### 4.191.2.12 svm\_inner\_conv\_check()

```
abip_float svm_inner_conv_check (
    svm * self,
    ABIPWork * w )
```

Check whether the inner loop of the svm socp formulation has converged.

Definition at line 234 of file [svm\\_config.c](#).

## 4.191.2.13 un\_scaling\_svm\_sol()

```
void un_scaling_svm_sol (
    svm * self,
    ABIPSolution * sol )
```

Get the unscaled solution of the original svm problem.

Definition at line 413 of file [svm\\_config.c](#).

## 4.192 svm\_config.c

[Go to the documentation of this file.](#)

```
00001 #include "svm_config.h"
00002 #define MIN_SCALE (1e-3)
00003 #define MAX_SCALE (1e3)
00004
00008 abip_int init_svm(svm **self, ABIPData *d, ABIPSettings *stgs) {
00009     svm *this_svm = (svm *)abip_malloc(sizeof(svm));
00010     *self = this_svm;
00011
00012     this_svm->m = d->m;
00013     this_svm->n = d->n;
00014     this_svm->p = d->m + d->n + 1;
00015     this_svm->q = 4 + 3 * d->n + 2 * d->m;
00016     this_svm->lambda = d->lambda;
00017     abip_int m = this_svm->p;
00018     abip_int n = this_svm->q;
00019     this_svm->Q = ABIP_NULL;
00020     this_svm->sparsity = (((abip_float)d->A->p[d->n] / (d->m * d->n)) < 0.05);
00021
00022     // non-identity DR scaling
00023     this_svm->rho_dr =
00024         (abip_float *)abip_malloc((m + n + 1) * sizeof(abip_float));
00025     for (int i = 0; i < m + n + 1; i++) {
00026         if (i < m) {
00027             this_svm->rho_dr[i] = stgs->rho_y;
00028         } else if (i < m + n) {
00029             this_svm->rho_dr[i] = stgs->rho_x;
00030         } else {
00031             this_svm->rho_dr[i] = stgs->rho_tau;
00032         }
00033     }
00034
00035     this_svm->L = (ABIPLinSysWork *)abip_malloc(sizeof(ABIPLinSysWork));
00036     this_svm->pro_type = SVM;
00037     this_svm->stgs = stgs;
00038
00039     this_svm->data = (ABIPData *)abip_malloc(sizeof(ABIPData));
00040
00041     abip_float *data_b =
00042         (abip_float *)abip_malloc(this_svm->p * sizeof(abip_float));
00043     memset(data_b, 0, this_svm->p * sizeof(abip_float));
00044     for (int i = 0; i < d->m + 1; i++) {
00045         data_b[i] = 1;
00046     }
00047     this_svm->data->b = data_b;
00048
00049     abip_float *data_c =
00050         (abip_float *)abip_malloc(this_svm->q * sizeof(abip_float));
00051     memset(data_c, 0, this_svm->q * sizeof(abip_float));
00052
00053     data_c[1] = 1;
00054
00055     for (int i = 0; i < d->m; i++) {
00056         data_c[i + 4 + 3 * d->n] = 1;
00057     }
00058
00059     this_svm->data->c = data_c;
00060
00061     /* for svm problem, no need for inputing c*/
00062     d->c = (abip_float *)abip_malloc(this_svm->q * sizeof(abip_float));
00063     memcpy(d->c, data_c, this_svm->q * sizeof(abip_float));
00064
00065     if ((d->m < 10 * d->n) && (10 * d->m > d->n)) {
00066         this_svm->sc = 1;
```

```

00067     this_svm->sc_c = MAX(0.45, POWF(7.5, (-log(2 * d->lambdab) / log(10))) * 2);
00068     this_svm->sc_b = 1;
00069     this_svm->sc_conel = MAX(3, log(2 * d->lambdab) / log(10) * 4 + 4);
00070     this_svm->sc_cone2 = this_svm->sc_conel;
00071 } else if (10 * d->m < d->n) {
00072     this_svm->sc = 1;
00073     this_svm->sc_b = 1;
00074     this_svm->sc_cone2 = MAX(3, log(2 * d->lambdab) / log(10) * 2 + 2);
00075     if (d->lambdab >= 1) {
00076         this_svm->sc_c =
00077             MAX(0.2, POWF(0.2, (log(2 * d->lambdab) / log(10))) * 7.5);
00078         this_svm->sc_conel = this_svm->sc_cone2;
00079     } else {
00080         this_svm->sc_c = POWF(0.3, log(2 * d->lambdab) / log(10)) * 3;
00081         this_svm->sc_conel = MAX(0.4, log(2 * d->lambdab) / log(10) * 0.2 + 0.8);
00082     }
00083 } else if (d->m > 10 * d->n) {
00084     if (d->n < 10) {
00085         this_svm->sc = 1;
00086         this_svm->sc_c = 1 / this_svm->lambdab;
00087         // this_svm->sc_c = 1.0/d->n;
00088         this_svm->sc_b = 1;
00089         this_svm->sc_conel = 6;
00090         if (d->lambdab < 0.002) {
00091             this_svm->sc_cone2 =
00092                 this_svm->sc_cone2 - 3 * log(this_svm->lambdab * 500) / log(10);
00093         }
00094     } else if (d->lambdab >= 1) {
00095         this_svm->sc = 1;
00096         this_svm->sc_c = 1 / this_svm->lambdab;
00097         this_svm->sc_b = 1;
00098         this_svm->sc_conel = 6;
00099         this_svm->sc_cone2 = this_svm->lambdab;
00100     } else {
00101         this_svm->sc = 1;
00102         this_svm->sc_c = MIN(POWF(5, (-log(5 * d->lambdab) / log(10))) * 4, 300);
00103         this_svm->sc_b = MAX(0.1, log(5 * d->lambdab) / log(10) * 0.2 + 0.9);
00104         this_svm->sc_conel = MAX(0.05, log(5 * d->lambdab) / log(10) * 0.3 + 0.7);
00105         this_svm->sc_cone2 = -log(5 * d->lambdab) / log(10) * 2 + 6;
00106         if (d->lambdab < 0.002) {
00107             this_svm->sc_cone2 =
00108                 this_svm->sc_cone2 - 3 * log(this_svm->lambdab * 500) / log(10);
00109         }
00110     }
00111 }
00112
00113 ABIPMatrix *data_A = (ABIPMatrix *)abip_malloc(sizeof(ABIPMatrix));
00114 data_A->m = d->m;
00115 data_A->n = d->n + 1;
00116 data_A->p = (abip_int *)abip_malloc((data_A->n + 1) * sizeof(abip_int));
00117 data_A->i =
00118     (abip_int *)abip_malloc((d->A->p[d->n] + d->m) * sizeof(abip_int));
00119 data_A->x =
00120     (abip_float *)abip_malloc((d->A->p[d->n] + d->m) * sizeof(abip_float));
00121 for (int i = 0; i < d->A->p[d->n]; i++) {
00122     d->A->x[i] = d->b[d->A->i[i]];
00123 }
00124 memcpy(data_A->p, d->A->p, (d->n + 1) * sizeof(abip_int));
00125 data_A->p[data_A->n] = d->A->p[d->n] + d->m;
00126
00127 memcpy(data_A->i, d->A->i, d->A->p[d->n] * sizeof(abip_int));
00128 for (int i = d->A->p[d->n]; i < d->A->p[d->n] + d->m; i++) {
00129     data_A->i[i] = i - d->A->p[d->n];
00130 }
00131
00132 memcpy(data_A->x, d->A->x, d->A->p[d->n] * sizeof(abip_float));
00133 for (int i = d->A->p[d->n]; i < d->A->p[d->n] + d->m; i++) {
00134     data_A->x[i] = d->b[i - d->A->p[d->n]];
00135 }
00136 this_svm->data->A = data_A;
00137 this_svm->A = (ABIPMatrix *)abip_malloc(sizeof(ABIPMatrix));
00138 this_svm->b = (abip_float *)abip_malloc(this_svm->p * sizeof(abip_float));
00139 this_svm->c = (abip_float *)abip_malloc(this_svm->q * sizeof(abip_float));
00140 this_svm->sc_D = (abip_float *)abip_malloc(this_svm->m * sizeof(abip_float));
00141 this_svm->sc_E =
00142     (abip_float *)abip_malloc((this_svm->n + 1) * sizeof(abip_float));
00143 this_svm->sc_F =
00144     (abip_float *)abip_malloc((this_svm->n) * sizeof(abip_float));
00145
00146 this_svm->wA = (ABIPMatrix *)abip_malloc(sizeof(ABIPMatrix));
00147 this_svm->wy = (abip_float *)abip_malloc(this_svm->m * sizeof(abip_float));
00148 this_svm->wB = (abip_float *)abip_malloc(this_svm->m * sizeof(abip_float));
00149 this_svm->wC = (abip_float *)abip_malloc(this_svm->m * sizeof(abip_float));
00150 this_svm->wD = (abip_float *)abip_malloc((this_svm->n) * sizeof(abip_float));
00151 this_svm->wE = (abip_float *)abip_malloc((this_svm->n) * sizeof(abip_float));
00152 this_svm->wF = (abip_float *)abip_malloc(this_svm->m * sizeof(abip_float));
00153 this_svm->wG = (abip_float *)abip_malloc((this_svm->n) * sizeof(abip_float));

```

```

00154     this_svm->wH =
00155         (abip_float *)abip_malloc((this_svm->n + 1) * sizeof(abip_float));
00156     this_svm->wX = (ABIPMatrix *)abip_malloc(sizeof(ABIPMatrix));
00157
00158     this_svm->scaling_data = &scaling_svm_data;
00159     this_svm->un_scaling_sol = &un_scaling_svm_sol;
00160     this_svm->calc_residuals = &calc_svm_residuals;
00161     this_svm->init_spe_linsys_work = &init_svm_linsys_work;
00162     this_svm->solve_spe_linsys = &solve_svm_linsys;
00163     this_svm->free_spe_linsys_work = &free_svm_linsys_work;
00164     this_svm->spe_A_times = &svm_A_times;
00165     this_svm->spe_AT_times = &svm_AT_times;
00166     this_svm->inner_conv_check = &svm_inner_conv_check;
00167
00168     return 0;
00169 }
00170
00171 void svm_A_times(svm *self, const abip_float *x, abip_float *y) {
00172     y[0] += x[0];
00173     abip_int m = self->m;
00174     abip_int n = self->n;
00175
00176     abip_float *tmp = (abip_float *)abip_malloc(n * sizeof(abip_float));
00177     memcpy(tmp, &x[n + 2], n * sizeof(abip_float));
00178     ABIP(add_scaled_array)(tmp, &x[2 * n + 3], n, -1);
00179     ABIP(accum_by_A)(self->wA, tmp, &y[1]);
00180
00181     ABIP(add_scaled_array)(&y[1], self->wy, m, (x[2 * n + 2] - x[3 * n + 3]));
00182     for (int i = 0; i < m; i++) {
00183         y[i + 1] +=
00184             (self->wB[i] * x[i + 3 * n + 4] - self->wC[i] * x[i + 3 * n + 4 + m]);
00185     }
00186
00187     for (int i = 0; i < n; i++) {
00188         y[i + 1 + m] += (self->wD[i] * x[i + 2] - self->wE[i] * tmp[i]);
00189     }
00190
00191     abip_free(tmp);
00192 }
00193
00194 void svm_AT_times(svm *self, const abip_float *x, abip_float *y) {
00195     y[0] += x[0];
00196     abip_int m = self->m;
00197     abip_int n = self->n;
00198
00199     for (int i = 0; i < n; i++) {
00200         y[i + 2] += self->wD[i] * x[i + m + 1];
00201     }
00202
00203     abip_float *tmp = (abip_float *)abip_malloc(n * sizeof(abip_float));
00204     for (int i = 0; i < n; i++) {
00205         tmp[i] = -self->wE[i] * x[i + m + 1];
00206     }
00207     ABIP(accum_by_Atrans)(self->wA, &x[1], tmp);
00208
00209     ABIP(add_scaled_array)(&y[n + 2], tmp, n, 1);
00210     ABIP(add_scaled_array)(&y[2 * n + 3], tmp, n, -1);
00211
00212     y[2 * n + 2] += ABIP(dot)(self->wy, &x[1], m);
00213     y[3 * n + 3] -= ABIP(dot)(self->wy, &x[1], m);
00214
00215     for (int i = 0; i < m; i++) {
00216         y[i + 3 * n + 4] += self->wB[i] * x[i + 1];
00217         y[i + 3 * n + 4 + m] -= self->wC[i] * x[i + 1];
00218     }
00219
00220     abip_free(tmp);
00221 }
00222
00223 abip_float svm_inner_conv_check(svm *self, ABIPWork *w) {
00224     DEBUG_FUNC
00225
00226     abip_int m = w->m;
00227     abip_int n = w->n;
00228     abip_int l = m + n + 1;
00229
00230     abip_float *y = (abip_float *)abip_malloc(m * sizeof(abip_float));
00231     memcpy(y, w->u, m * sizeof(abip_float));
00232     abip_float *x = (abip_float *)abip_malloc(n * sizeof(abip_float));
00233     memcpy(x, &w->u[m], n * sizeof(abip_float));
00234     abip_float *s = (abip_float *)abip_malloc(n * sizeof(abip_float));
00235     memcpy(s, &w->v_origin[m], n * sizeof(abip_float));
00236
00237     abip_float tau = w->u[l - 1];
00238     abip_float kap = w->v_origin[l - 1];
00239
00240     abip_float *row1 = (abip_float *)abip_malloc(m * sizeof(abip_float));

```

```

00252 memcpy(row1, self->b, m * sizeof(abip_float));
00253 ABIP(scale_array)(row1, -tau, m);
00254 self->spe_A_times(self, x, row1);
00255
00256 abip_float *row2 = (abip_float *)abip_malloc(n * sizeof(abip_float));
00257 memcpy(row2, s, n * sizeof(abip_float));
00258 ABIP(add_scaled_array)(row2, self->c, n, -tau);
00259 self->spe_AT_times(self, y, row2);
00260 ABIP(scale_array)(row2, -1, n);
00261
00262 abip_float *Pu_v = (abip_float *)abip_malloc(1 * sizeof(abip_float));
00263 memcpy(Pu_v, row1, m * sizeof(abip_float));
00264 memcpy(&Pu_v[m], row2, n * sizeof(abip_float));
00265 Pu_v[1 - 1] = ABIP(dot)(self->b, y, m) - ABIP(dot)(self->c, x, n) - kap;
00266
00267 abip_float err_inner =
00268     ABIP(norm)(Pu_v, 1) /
00269     (1 + SQRTF(ABIP(norm_sq)(w->u, 1) + ABIP(norm_sq)(w->v_origin, 1)));
00270
00271 abip_free(x);
00272 abip_free(y);
00273 abip_free(s);
00274 abip_free(row1);
00275 abip_free(row2);
00276 abip_free(Pu_v);
00277 return err_inner;
00278 }
00279
00283 void scaling_svm_data(svm *self, ABIPConc *k) {
00284     if (!ABIP(copy_A_matrix)(&(self->A), self->data->A)) {
00285         abip_printf("ERROR: copy A matrix failed\n");
00286         RETURN;
00287     }
00288
00289     abip_int m = self->m;
00290     abip_int n = self->n + 1;
00291     ABIPMatrix *A = self->A;
00292
00293     abip_float min_row_scale = MIN_SCALE * SQRTF((abip_float)n);
00294     abip_float max_row_scale = MAX_SCALE * SQRTF((abip_float)n);
00295     abip_float min_col_scale = MIN_SCALE * SQRTF((abip_float)m);
00296     abip_float max_col_scale = MAX_SCALE * SQRTF((abip_float)m);
00297
00298     abip_float *E = self->sc_E;
00299     memset(E, 0, n * sizeof(abip_float));
00300
00301     abip_float *D = self->sc_D;
00302     memset(D, 0, m * sizeof(abip_float));
00303
00304     abip_float avg = 0;
00305
00306     if (self->stgs->scale_E) {
00307         for (int i = 0; i < n; i++) {
00308             for (int j = A->p[i]; j < A->p[i + 1]; j++) {
00309                 E[i] += A->x[j] * A->x[j];
00310             }
00311             E[i] = SQRTF(E[i]);
00312             avg += E[i];
00313         }
00314
00315         avg /= n;
00316
00317         for (int i = 0; i < n; i++) {
00318             E[i] = avg / E[i];
00319         }
00320
00321         for (int i = 0; i < n; i++) {
00322             for (int j = A->p[i]; j < A->p[i + 1]; j++) {
00323                 A->x[j] *= E[i];
00324             }
00325         }
00326     }
00327
00328     for (int i = 0; i < A->p[n]; i++) {
00329         D[A->i[i]] += A->x[i] * A->x[i];
00330     }
00331
00332     avg = 0;
00333     for (int i = 0; i < m; i++) {
00334         avg += SQRTF(D[i]);
00335     }
00336
00337     avg /= m;
00338
00339     for (int i = 0; i < m; i++) {
00340         D[i] = avg / SQRTF(D[i]);
00341     }

```



```

00342
00343     for (int i = 0; i < A->p[n]; i++) {
00344         A->x[i] *= D[A->i[i]];
00345     }
00346
00347     for (int i = 0; i < n - 1; i++) {
00348         self->sc_F[i] = 1 / SQRTF(1 + 2 * E[i] * E[i]);
00349     }
00350
00351     memcpy(self->b, self->data->b, self->p * sizeof(abip_float));
00352
00353     self->b[0] = self->sc_cone2;
00354
00355     for (int i = 1; i < m + 1; i++) {
00356         self->b[i] = D[i - 1];
00357     }
00358     ABIP(scale_array)(self->b, self->sc_b, self->p);
00359
00360     memcpy(self->c, self->data->c, self->q * sizeof(abip_float));
00361     self->c[0] = 0;
00362     self->c[1] = self->sc_c * self->sc_cone1 * self->sc_cone2;
00363
00364     for (int i = 0; i < m; i++) {
00365         self->c[i + 4 + 3 * self->n] = self->lambda * self->sc_c / self->sc;
00366     }
00367
00368     n -= 1;
00369
00370     ABIP(copy_A_matrix)(&(self->wA), self->A);
00371     self->wA->n -= 1;
00372
00373     memcpy(self->wy, &self->A->x[self->A->p[n]], m * sizeof(abip_float));
00374
00375     memcpy(self->wB, self->sc_D, m * sizeof(abip_float));
00376     ABIP(scale_array)(self->wB, 1 / self->sc, m);
00377
00378     memcpy(self->wC, self->sc_D, m * sizeof(abip_float));
00379
00380     memcpy(self->wD, self->sc_F, n * sizeof(abip_float));
00381     ABIP(scale_array)(self->wD, -SQRTF(self->sc_cone1), n);
00382
00383     for (int i = 0; i < n; i++) {
00384         self->wE[i] = self->sc_E[i] * self->sc_F[i];
00385     }
00386
00387     for (int i = 0; i < m; i++) {
00388         self->wF[i] = self->wB[i] * self->wB[i] + self->wC[i] * self->wC[i] +
00389             self->stgs->rho_y;
00390     }
00391
00392     for (int i = 0; i < n; i++) {
00393         self->wG[i] = self->wD[i] * self->wD[i] + 2 * self->wE[i] * self->wE[i] +
00394             self->stgs->rho_y;
00395     }
00396
00397     for (int i = 0; i < n; i++) {
00398         self->wH[i] = 2 - 4 / self->wG[i] * self->wE[i] * self->wE[i];
00399     }
00400     self->wH[n] = 2;
00401
00402     ABIP(copy_A_matrix)(&(self->wX), self->wA);
00403     for (int i = 0; i < n; i++) {
00404         for (int j = self->wX->p[i]; j < self->wX->p[i + 1]; j++) {
00405             self->wX->x[j] *= (-2 * self->wE[i]);
00406         }
00407     }
00408 }
00409
00413 void un_scaling_svm_sol(svm *self, ABIPSolution *sol) {
00414     abip_int m = self->m;
00415     abip_int n = self->n;
00416     abip_float *x = sol->x;
00417     abip_float *y = sol->y;
00418     abip_float *s = sol->s;
00419
00420     abip_float *w = (abip_float *)abip_malloc(n * sizeof(abip_float));
00421     memcpy(w, &x[n + 2], n * sizeof(abip_float));
00422     ABIP(add_scaled_array)(w, &x[2 * n + 3], n, -1);
00423     ABIP(c_dot)(w, self->sc_E, n);
00424     ABIP(scale_array)(w, 1 / self->sc_b, n);
00425
00426     abip_float *b = (abip_float *)abip_malloc(sizeof(abip_float));
00427     b[0] = (x[2 * n + 2] - x[3 * n + 3]) * self->sc_E[n] / self->sc_b;
00428
00429     abip_float *xi = (abip_float *)abip_malloc(m * sizeof(abip_float));
00430     memcpy(xi, &x[3 * n + 4], m * sizeof(abip_float));
00431     ABIP(scale_array)(xi, 1 / (self->sc_b * self->sc_c), m);

```

```

00432
00433     abip_free(x);
00434     abip_free(y);
00435     abip_free(s);
00436
00437     sol->x = w;
00438     sol->y = b;
00439     sol->s = xi;
00440 }
00441
00445 void calc_svm_residuals(svm *self, ABIPWork *w, ABIPResiduals *r,
00446                        abip_int ipm_iter, abip_int admm_iter) {
00447     abip_int p = w->m;
00448     abip_int q = w->n;
00449     abip_int m = self->m;
00450     abip_int n = self->n;
00451     abip_float C = self->lambda;
00452     abip_float this_pr;
00453     abip_float this_dr;
00454     abip_float this_gap;
00455
00456     r->tau = w->u[p + q];
00457
00458     abip_float *w1 = (abip_float *)abip_malloc((n + 1) * sizeof(abip_float));
00459     memcpy(w1, &w->u[m + 2 * n + 3], n * sizeof(abip_float));
00460     ABIP(add_scaled_array)(w1, &w->u[m + 3 * n + 4], n, -1);
00461     for (int i = 0; i < n; i++) {
00462         w1[i] *= (self->sc_E[i] / (r->tau * self->sc_b));
00463     }
00464
00465     abip_float b = (w->u[m + 3 * n + 3] - w->u[m + 4 * n + 4]) / r->tau *
00466                   self->sc_E[n] / self->sc_b;
00467     w1[n] = b;
00468
00469     abip_float *xi = (abip_float *)abip_malloc(m * sizeof(abip_float));
00470     memcpy(xi, &w->u[m + 4 * n + 5], m * sizeof(abip_float));
00471     ABIP(scale_array)(xi, 1 / (r->tau * self->sc * self->sc_b), m);
00472
00473     abip_float *t = (abip_float *)abip_malloc(m * sizeof(abip_float));
00474     memcpy(t, &w->u[2 * m + 4 * n + 5], m * sizeof(abip_float));
00475     ABIP(scale_array)(t, 1 / (r->tau * self->sc_b), m);
00476
00477     abip_float *y = (abip_float *)abip_malloc(m * sizeof(abip_float));
00478     memcpy(y, &w->u[1], m * sizeof(abip_float));
00479     for (int i = 0; i < m; i++) {
00480         y[i] = y[i] / r->tau * self->sc_D[i] / self->sc_c;
00481     }
00482
00483     abip_float *s2 = (abip_float *)abip_malloc(m * sizeof(abip_float));
00484     memcpy(s2, &w->v[2 * m + 4 * n + 5], m * sizeof(abip_float));
00485     ABIP(scale_array)(s2, 1 / (r->tau * self->sc_c), m);
00486
00487     abip_float *s1 = (abip_float *)abip_malloc(m * sizeof(abip_float));
00488     memcpy(s1, &w->v[m + 4 * n + 5], m * sizeof(abip_float));
00489     ABIP(scale_array)(s1, self->sc / (r->tau * self->sc_c), m);
00490
00491     abip_float *pr = (abip_float *)abip_malloc(m * sizeof(abip_float));
00492     memcpy(pr, xi, m * sizeof(abip_float));
00493     ABIP(accum_by_A)(self->data->A, w1, pr);
00494     for (int i = 0; i < m; i++) {
00495         pr[i] -= (t[i] + 1);
00496     }
00497     this_pr = ABIP(norm)(pr, m) / SQRTF(m);
00498
00499     abip_float *dr = (abip_float *)abip_malloc(2 * m * sizeof(abip_float));
00500     for (int i = 0; i < m; i++) {
00501         dr[i] = y[i] - s2[i];
00502         dr[i + m] = y[i] + s1[i] - C;
00503     }
00504     this_dr = ABIP(norm)(dr, 2 * m) / (SQRTF(m) * C);
00505
00506     ABIPMatrix *B = (ABIPMatrix *)abip_malloc(sizeof(ABIPMatrix));
00507     ABIP(copy_A_matrix)(B, self->data->A);
00508     B->n = 1;
00509     abip_float *BTy = (abip_float *)abip_malloc(n * sizeof(abip_float));
00510     memset(BTy, 0, n * sizeof(abip_float));
00511     ABIP(accum_by_Atrans)(B, y, BTy);
00512
00513     r->dobj = 0;
00514     r->pobj = 0;
00515     for (int i = 0; i < m; i++) {
00516         r->dobj += y[i];
00517         r->pobj += C * xi[i];
00518     }
00519
00520     r->pobj += 0.5 * ABIP(dot)(w1, w1, n);
00521     r->dobj -= 0.5 * ABIP(dot)(BTy, BTy, n);

```

```

00522
00523     this_gap = r->dobj - r->pobj;
00524     this_gap = ABS(this_gap) / (1 + ABS(r->pobj));
00525
00526     r->last_ipm_iter = ipm_iter;
00527     r->last_admm_iter = admm_iter;
00528
00529     r->res_infeas = NAN;
00530     r->res_unbdd = NAN;
00531
00532     r->res_dif = MAX(MAX(ABS(this_pr - r->res_pri), ABS(this_dr - r->res_dual)),
00533                     ABS(this_gap - r->rel_gap));
00534     r->res_pri = this_pr;
00535     r->res_dual = this_dr;
00536     r->rel_gap = this_gap;
00537     r->error_ratio =
00538         MAX(r->res_pri / self->stgs->eps_p,
00539             MAX(r->res_dual / self->stgs->eps_d, r->rel_gap / self->stgs->eps_g));
00540
00541     if (ABIP(dot)(self->c, &w->u[p], q) < 0) {
00542         abip_float *Ax = (abip_float *)abip_malloc(p * sizeof(abip_float));
00543         memset(Ax, 0, p * sizeof(abip_float));
00544         self->spe_A_times(self, &w->u[p], Ax);
00545         r->res_unbdd = ABIP(norm)(Ax, p) / (-ABIP(dot)(self->c, &w->u[p], q));
00546         abip_free(Ax);
00547     } else {
00548         r->res_unbdd = INFINITY;
00549     }
00550
00551     if (ABIP(dot)(self->b, w->u, p) > 0) {
00552         abip_float *ATy_s = (abip_float *)abip_malloc(q * sizeof(abip_float));
00553         memset(ATy_s, 0, q * sizeof(abip_float));
00554         self->spe_AT_times(self, w->u, ATy_s);
00555         ABIP(add_scaled_array)(ATy_s, &w->v_origin[p], q, 1);
00556
00557         r->res_infeas = ABIP(norm)(ATy_s, q) / ABIP(dot)(self->b, w->u, p);
00558         abip_free(ATy_s);
00559     } else {
00560         r->res_infeas = INFINITY;
00561     }
00562
00563     ABIP(free_A_matrix)(B);
00564     abip_free(w1);
00565     abip_free(xi);
00566     abip_free(t);
00567     abip_free(y);
00568     abip_free(s1);
00569     abip_free(s2);
00570     abip_free(pr);
00571     abip_free(dr);
00572 }
00573
00577 cs *form_svm_kkt(svm *self) {
00578     abip_int n = self->n;
00579     abip_int m = self->m;
00580     cs *LTL;
00581
00582     cs *N1 = cs_spalloc(m, n + 1, self->A->p[n + 1], 1, 0);
00583     memcpy(N1->i, self->A->i, self->A->p[n + 1] * sizeof(abip_int));
00584     memcpy(N1->p, self->A->p, (n + 2) * sizeof(abip_int));
00585     memcpy(N1->x, self->A->x, self->A->p[n + 1] * sizeof(abip_float));
00586
00587     cs *N2 = cs_spalloc(m, n + 1, self->A->p[n + 1], 1, 0);
00588     memcpy(N2->i, self->A->i, self->A->p[n + 1] * sizeof(abip_int));
00589     memcpy(N2->p, self->A->p, (n + 2) * sizeof(abip_int));
00590     memcpy(N2->x, self->A->x, self->A->p[n + 1] * sizeof(abip_float));
00591
00592     if (m > n + 1) {
00593         for (int i = 0; i < N1->p[n + 1]; i++) {
00594             N1->x[i] /= self->wF[N1->i[i]];
00595         }
00596
00597         cs *diag = cs_spalloc(n + 1, n + 1, n + 1, 1, 1);
00598
00599         for (int i = 0; i < n + 1; i++) {
00600             cs_entry(diag, i, i, 1 / self->wH[i]);
00601         }
00602         cs *diag_H = cs_compress(diag);
00603         cs_spfree(diag);
00604
00605         LTL = cs_add(diag_H, cs_multiply(cs_transpose(N2, 1), N1), 1, 1);
00606         cs_spfree(diag_H);
00607     } else {
00608         for (int i = 0; i < N1->n; i++) {
00609             for (int j = N1->p[i]; j < N1->p[i + 1]; j++) {
00610                 N1->x[j] *= self->wH[i];
00611             }

```

```

00612     }
00613     cs *diag = cs_spalloc(m, m, m, 1, 1);
00614
00615     for (int i = 0; i < m; i++) {
00616         cs_entry(diag, i, i, self->wF[i]);
00617     }
00618     cs *diag_F = cs_compress(diag);
00619     cs_spfree(diag);
00620
00621     LTL = cs_add(diag_F, cs_multiply(N1, cs_transpose(N2, 1)), 1, 1);
00622
00623     cs_spfree(diag_F);
00624 }
00625
00626 cs_spfree(N1);
00627 cs_spfree(N2);
00628
00629 for (int i = 0; i < LTL->n; i++) {
00630     for (int j = LTL->p[i]; j < LTL->p[i + 1]; j++) {
00631         if (LTL->i[j] > i) LTL->x[j] = 0;
00632     }
00633 }
00634 cs_dropzeros(LTL);
00635 return LTL;
00636 }
00637
00642 void init_svm_precon(svm *self) {
00643     self->L->M = (abip_float *)abip_malloc(self->p * sizeof(abip_float));
00644     memset(self->L->M, 0, self->p * sizeof(abip_float));
00645
00646     abip_float *M = self->L->M;
00647
00648     M[0] = 1 / (self->stgs->rho_y + 1);
00649
00650     for (int i = 0; i < self->A->p[self->A->n]; i++) {
00651         M[self->A->i[i] + 1] += 2 * POWF(self->A->x[i], 2);
00652     }
00653
00654     for (int i = 1; i < self->m + 1; i++) {
00655         M[i] = 1 / (self->stgs->rho_y + M[i] + POWF(self->wB[i - 1], 2) +
00656             POWF(self->sc_D[i - 1], 2));
00657     }
00658
00659     for (int i = self->m + 1; i < self->m + self->n + 1; i++) {
00660         M[i] = 1 / (self->stgs->rho_y + 2 * POWF(self->wE[i - self->m + 1], 2) +
00661             POWF(self->wD[i - self->m + 1], 2));
00662     }
00663 }
00664
00669 abip_float get_svm_pcg_tol(abip_int k, abip_float error_ratio,
00670     abip_float norm_p) {
00671     if (k == -1) {
00672         return 1e-9 * norm_p;
00673     } else {
00674         if (error_ratio > 100000) {
00675             return MAX(1e-9, 3e-2 * norm_p / POWF((k + 1), 2));
00676         } else if (error_ratio > 30000) {
00677             return MAX(1e-9, 3e-2 * norm_p / POWF((k + 1), 2));
00678         } else if (error_ratio > 10000) {
00679             return MAX(1e-9, 3e-2 * norm_p / POWF((k + 1), 2));
00680         } else if (error_ratio > 3000) {
00681             return MAX(1e-9, 2.5e-2 * norm_p / POWF((k + 1), 2));
00682         } else if (error_ratio > 1000) {
00683             return MAX(1e-9, 2e-2 * norm_p / POWF((k + 1), 2));
00684         } else if (error_ratio > 300) {
00685             return MAX(1e-9, 1.6e-2 * norm_p / POWF((k + 1), 2));
00686         } else if (error_ratio > 100) {
00687             return MAX(1e-9, 1.3e-2 * norm_p / POWF((k + 1), 2));
00688         } else if (error_ratio > 30) {
00689             return MAX(1e-9, 1e-2 * norm_p / POWF((k + 1), 2));
00690         } else if (error_ratio > 10) {
00691             return MAX(1e-9, 7e-3 * norm_p / POWF((k + 1), 2));
00692         } else {
00693             return MAX(1e-9, 4e-3 * norm_p / POWF((k + 1), 2));
00694         }
00695     }
00696 }
00697
00702 abip_int init_svm_linsys_work(svm *self) {
00703     if (self->stgs->linsys_solver == 0) { // mkl-dss need lower triangle
00704         self->L->K = cs_transpose(form_svm_kkt(self), 1);
00705     } else if (self->stgs->linsys_solver == 1) { // qdldl need upper triangle
00706         self->L->K = form_svm_kkt(self);
00707     } else if (self->stgs->linsys_solver == 2) { // cholesky need upper triangle
00708         self->L->K = form_svm_kkt(self);
00709     } else if (self->stgs->linsys_solver == 3) { // pcg doesn't need kkt matrix
00710         init_svm_precon(self);

```

```

00711     self->L->K = ABIP_NULL;
00712 } else if (self->stgs->linsys_solver ==
00713     4) { // mkl-pardiso need lower triangle
00714     self->L->K = cs_transpose(form_svm_kkt(self), 1);
00715 } else if (self->stgs->linsys_solver ==
00716     5) { // dense cholesky need upper triangle
00717     self->L->K = form_svm_kkt(self);
00718 } else {
00719     printf("\nlinsys solver type error\n");
00720     return -1;
00721 }
00722 return ABIP(init_linsys_work)(self);
00723 }
00724
00728 abip_int solve_svm_linsys(svm *self, abip_float *b, abip_float *pcg_warm_start,
00729     abip_int iter, abip_float error_ratio) {
00730     ABIP(timer) linsys_timer;
00731     ABIP(tic)(&linsys_timer);
00732
00733     ABIP(scale_array)(&b[self->p], -1, self->q);
00734
00735     if (self->stgs->linsys_solver == 3) { // pcg
00736
00737         abip_int p = self->p;
00738         abip_float norm_p = ABIP(norm)(b, p);
00739
00740         self->spe_A_times(self, &b[p], b);
00741         abip_float pcg_tol = get_svm_pcg_tol(iter, error_ratio, norm_p);
00742         abip_int cg_its = ABIP(solve_linsys)(self, b, p, pcg_warm_start, pcg_tol);
00743
00744         if (iter >= 0) {
00745             self->L->total_cg_its += cg_its;
00746         }
00747     } else { // direct methods
00748         abip_int n = self->n;
00749         abip_int m = self->m;
00750         abip_float *b2 = (abip_float *)abip_malloc(self->p * sizeof(abip_float));
00751         memcpy(b2, b, self->p * sizeof(abip_float));
00752         self->spe_A_times(self, &b[self->p], b2);
00753         b[0] = b2[0] / (1 + self->stgs->rho_y);
00754
00755         for (int i = 0; i < n; i++) {
00756             b[i + m + 1] = b2[i + m + 1] / self->wG[i];
00757         }
00758
00759         abip_float *b3 = (abip_float *)abip_malloc(m * sizeof(abip_float));
00760         memcpy(b3, &b[1], m * sizeof(abip_float));
00761         ABIP(scale_array)(b3, -1, m);
00762         ABIP(accum_by_A)(self->wX, &b[m + 1], b3);
00763         ABIP(scale_array)(b3, -1, m);
00764
00765         abip_float *tmp = (abip_float *)abip_malloc((n + 1) * sizeof(abip_float));
00766         if (m > n + 1) {
00767             for (int i = 0; i < m; i++) {
00768                 b3[i] /= self->wF[i];
00769             }
00770             memset(tmp, 0, (n + 1) * sizeof(abip_float));
00771             ABIP(accum_by_Atrans)(self->A, b3, tmp);
00772             ABIP(solve_linsys)(self, tmp, n + 1, ABIP_NULL, 0);
00773
00774             abip_float *tmp2 = (abip_float *)abip_malloc(m * sizeof(abip_float));
00775             memset(tmp2, 0, m * sizeof(abip_float));
00776
00777             ABIP(accum_by_A)(self->A, tmp, tmp2);
00778             for (int i = 0; i < m; i++) {
00779                 tmp2[i] /= self->wF[i];
00780             }
00781             memcpy(&b[1], b3, m * sizeof(abip_float));
00782             ABIP(add_scaled_array)(&b[1], tmp2, m, -1);
00783             abip_free(tmp2);
00784         } else {
00785             ABIP(solve_linsys)(self, b3, m, ABIP_NULL, 0);
00786             memcpy(&b[1], b3, m * sizeof(abip_float));
00787         }
00788
00789         memset(tmp, 0, (n + 1) * sizeof(abip_float));
00790         ABIP(accum_by_Atrans)(self->wX, &b[1], tmp);
00791         for (int i = 0; i < n; i++) {
00792             tmp[i] /= self->wG[i];
00793         }
00794         ABIP(add_scaled_array)(&b[m + 1], tmp, n, -1);
00795
00796         abip_free(b2);
00797         abip_free(b3);
00798         abip_free(tmp);
00799     }
00800 }

```

```

00801  ABIP(scale_array)(&b[self->p], -1, self->q);
00802  self->spe_AT_times(self, b, &b[self->p]);
00803
00804  self->L->total_solve_time += ABIP(tocq)(&linsys_timer);
00805
00806  return 0;
00807 }
00808
00812 void free_svm_linsys_work(svm *self) { ABIP(free_linsys)(self); }

```

## 4.193 source/svm\_qp\_config.c File Reference

```
#include "svm_qp_config.h"
```

### Macros

- #define `MIN_SCALE` (1e-3)
- #define `MAX_SCALE` (1e3)

### Functions

- `abip_int init_svmqp (svmqp **self, ABIPData *d, ABIPSettings *stgs)`  
*Initialize the svm qp formulation structure.*
- void `svmqp_A_times (svmqp *self, const abip_float *x, abip_float *y)`  
*Customized matrix-vector multiplication for the svm qp formulation with A untransposed.*
- void `svmqp_AT_times (svmqp *self, const abip_float *x, abip_float *y)`  
*Customized matrix-vector multiplication for the svm qp formulation with A transposed.*
- `abip_float svmqp_inner_conv_check (svmqp *self, ABIPWork *w)`  
*Check whether the inner loop of the svm qp formulation has converged.*
- void `scaling_svmqp_data (svmqp *self, ABIPConc *k)`  
*Customized scaling procedure for the svm qp formulation.*
- void `un_scaling_svmqp_sol (svmqp *self, ABIPSolution *sol)`  
*Get the unscaled solution of the original svm problem.*
- void `calc_svmqp_residuals (svmqp *self, ABIPWork *w, ABIPResiduals *r, abip_int ipm_iter, abip_int admm_iter)`  
*Calculate the residuals of the svm qp formulation.*
- `cs * form_svmqp_kkt (svmqp *self)`  
*Formulate the qcp KKT matrix of the svm qp formulation.*
- void `init_svmqp_precon (svmqp *self)`  
*Initialize the preconditioner of conjugate gradient method for the svm qp formulation.*
- `abip_float get_svmqp_pcg_tol (abip_int k, abip_float error_ratio, abip_float norm_p)`  
*Get the tolerance of the conjugate gradient method for the svm qp formulation.*
- `abip_int init_svmqp_linsys_work (svmqp *self)`  
*Initialize the linear system solver work space for the svm qp formulation.*
- `abip_int solve_svmqp_linsys (svmqp *self, abip_float *b, abip_float *pcg_warm_start, abip_int iter, abip_float error_ratio)`  
*Customized linear system solver for the svm qp formulation.*
- void `free_svmqp_linsys_work (svmqp *self)`  
*Free the linear system solver work space for the svm qp formulation.*

## 4.193.1 Macro Definition Documentation

### 4.193.1.1 MAX\_SCALE

```
#define MAX_SCALE (1e3)
```

Definition at line 3 of file [svm\\_qp\\_config.c](#).

### 4.193.1.2 MIN\_SCALE

```
#define MIN_SCALE (1e-3)
```

Definition at line 2 of file [svm\\_qp\\_config.c](#).

## 4.193.2 Function Documentation

### 4.193.2.1 calc\_svmqp\_residuals()

```
void calc_svmqp_residuals (  
    svmqp * self,  
    ABIPWork * w,  
    ABIPResiduals * r,  
    abip_int ipm_iter,  
    abip_int admm_iter )
```

Calculate the residuals of the svm qp formulation.

Definition at line 628 of file [svm\\_qp\\_config.c](#).

### 4.193.2.2 form\_svmqp\_kkt()

```
cs * form_svmqp_kkt (  
    svmqp * self )
```

Formulate the qcp KKT matrix of the svm qp formulation.

Definition at line 761 of file [svm\\_qp\\_config.c](#).

#### 4.193.2.3 free\_svmqp\_linsys\_work()

```
void free_svmqp_linsys_work (
    svmqp * self )
```

Free the linear system solver work space for the svm qp formulation.

Definition at line 1002 of file [svm\\_qp\\_config.c](#).

#### 4.193.2.4 get\_svmqp\_pcg\_tol()

```
abip_float get_svmqp_pcg_tol (
    abip_int k,
    abip_float error_ratio,
    abip_float norm_p )
```

Get the tolerance of the conjugate gradient method for the svm qp formulation.

Definition at line 855 of file [svm\\_qp\\_config.c](#).

#### 4.193.2.5 init\_svmqp()

```
abip_int init_svmqp (
    svmqp ** self,
    ABIPData * d,
    ABIPSettings * stgs )
```

Initialize the svm qp formulation structure.

Definition at line 8 of file [svm\\_qp\\_config.c](#).

#### 4.193.2.6 init\_svmqp\_linsys\_work()

```
abip_int init_svmqp_linsys_work (
    svmqp * self )
```

Initialize the linear system solver work space for the svm qp formulation.

Definition at line 868 of file [svm\\_qp\\_config.c](#).



#### 4.193.2.7 init\_svmqp\_precon()

```
void init_svmqp_precon (
    svmqp * self )
```

Initialize the preconditioner of conjugate gradient method for the svm qp formulation.

Definition at line 826 of file [svm\\_qp\\_config.c](#).

#### 4.193.2.8 scaling\_svmqp\_data()

```
void scaling_svmqp_data (
    svmqp * self,
    ABIPConc * k )
```

Customized scaling procedure for the svm qp formulation.

Definition at line 196 of file [svm\\_qp\\_config.c](#).

#### 4.193.2.9 solve\_svmqp\_linsys()

```
abip_int solve_svmqp_linsys (
    svmqp * self,
    abip_float * b,
    abip_float * pcg_warm_start,
    abip_int iter,
    abip_float error_ratio )
```

Customized linear system solver for the svm qp formulation.

Definition at line 894 of file [svm\\_qp\\_config.c](#).

#### 4.193.2.10 svmqp\_A\_times()

```
void svmqp_A_times (
    svmqp * self,
    const abip_float * x,
    abip_float * y )
```

Customized matrix-vector multiplication for the svm qp formulation with A untransposed.

Definition at line 128 of file [svm\\_qp\\_config.c](#).

**4.193.2.11 svmqp\_AT\_times()**

```
void svmqp_AT_times (
    svmqp * self,
    const abip_float * x,
    abip_float * y )
```

Customized matrix-vector multiplication for the svm qp formulation with A transposed.

Definition at line 141 of file [svm\\_qp\\_config.c](#).

**4.193.2.12 svmqp\_inner\_conv\_check()**

```
abip_float svmqp_inner_conv_check (
    svmqp * self,
    ABIPWork * w )
```

Check whether the inner loop of the svm qp formulation has converged.

Definition at line 152 of file [svm\\_qp\\_config.c](#).

**4.193.2.13 un\_scaling\_svmqp\_sol()**

```
void un_scaling_svmqp_sol (
    svmqp * self,
    ABIPSolution * sol )
```

Get the unscaled solution of the original svm problem.

Definition at line 597 of file [svm\\_qp\\_config.c](#).

**4.194 svm\_qp\_config.c**

[Go to the documentation of this file.](#)

```
00001 #include "svm_qp_config.h"
00002 #define MIN_SCALE (1e-3)
00003 #define MAX_SCALE (1e3)
00004
00008 abip_int init_svmqp(svmqp **self, ABIPData *d, ABIPSettings *stgs) {
00009     svmqp *this_svm = (svmqp *)abip_malloc(sizeof(svmqp));
00010     *self = this_svm;
00011
00012     this_svm->pro_type = SVMQP;
00013     this_svm->m = d->m;
00014     this_svm->n = d->n;
00015     this_svm->p = d->m;
00016     this_svm->q = 1 + d->n + 2 * d->m;
00017     this_svm->lambda = d->lambda;
00018     abip_int m = this_svm->p;
00019     abip_int n = this_svm->q;
00020
00021     this_svm->Q = (ABIPMatrix *)abip_malloc(sizeof(ABIPMatrix));
00022     this_svm->Q->m = n;
00023     this_svm->Q->n = n;
00024     this_svm->Q->i = (abip_int *)abip_malloc(sizeof(abip_int) * d->n);
```

```

00025     this_svm->Q->p = (abip_int *)abip_malloc(sizeof(abip_int) * (n + 1));
00026     this_svm->Q->x = (abip_float *)abip_malloc(sizeof(abip_float) * d->n);
00027
00028     for (abip_int i = 0; i < d->n; i++) {
00029         this_svm->Q->i[i] = i;
00030         this_svm->Q->p[i] = i;
00031         this_svm->Q->x[i] = 1;
00032     }
00033
00034     for (abip_int i = 0; i < 2 * d->m + 2; i++) {
00035         this_svm->Q->p[d->n + i] = d->n;
00036     }
00037
00038     this_svm->sparsity = (((abip_float)d->A->p[d->n] / (d->m * d->n)) < 0.05);
00039     this_svm->rho_dr =
00040         (abip_float *)abip_malloc((m + n + 1) * sizeof(abip_float));
00041     for (int i = 0; i < m + n + 1; i++) {
00042         if (i < m) {
00043             this_svm->rho_dr[i] = stgs->rho_y;
00044         } else if (i < m + n) {
00045             this_svm->rho_dr[i] = stgs->rho_x;
00046         } else {
00047             this_svm->rho_dr[i] = stgs->rho_tau;
00048         }
00049     }
00050
00051     this_svm->L = (ABIPLinSysWork *)abip_malloc(sizeof(ABIPLinSysWork));
00052     this_svm->stgs = stgs;
00053
00054     this_svm->data = (ABIPData *)abip_malloc(sizeof(ABIPData));
00055
00056     abip_float *data_b = (abip_float *)abip_malloc(m * sizeof(abip_float));
00057     memset(data_b, 0, m * sizeof(abip_float));
00058     for (int i = 0; i < m; i++) {
00059         data_b[i] = 1;
00060     }
00061     this_svm->data->b = data_b;
00062
00063     abip_float *data_c = (abip_float *)abip_malloc(n * sizeof(abip_float));
00064     memset(data_c, 0, this_svm->q * sizeof(abip_float));
00065
00066     for (int i = 0; i < d->m; i++) {
00067         data_c[i + d->n + 1] = 1.0 / (this_svm->m * this_svm->lambda);
00068     }
00069     this_svm->data->c = data_c;
00070
00071     d->c = (abip_float *)abip_malloc(n * sizeof(abip_float));
00072     memcpy(d->c, data_c, n * sizeof(abip_float));
00073
00074     ABIPMatrix *data_A = (ABIPMatrix *)abip_malloc(sizeof(ABIPMatrix));
00075     data_A->m = d->m;
00076     data_A->n = d->n + 1;
00077     data_A->p = (abip_int *)abip_malloc((data_A->n + 1) * sizeof(abip_int));
00078     data_A->i =
00079         (abip_int *)abip_malloc((d->A->p[d->n] + d->m) * sizeof(abip_int));
00080     data_A->x =
00081         (abip_float *)abip_malloc((d->A->p[d->n] + d->m) * sizeof(abip_float));
00082     for (int i = 0; i < d->A->p[d->n]; i++) {
00083         d->A->x[i] *= d->b[d->A->i[i]];
00084     }
00085     memcpy(data_A->p, d->A->p, (d->n + 1) * sizeof(abip_int));
00086     data_A->p[data_A->n] = d->A->p[d->n] + d->m;
00087
00088     memcpy(data_A->i, d->A->i, d->A->p[d->n] * sizeof(abip_int));
00089     for (int i = d->A->p[d->n]; i < d->A->p[d->n] + d->m; i++) {
00090         data_A->i[i] = i - d->A->p[d->n];
00091     }
00092
00093     memcpy(data_A->x, d->A->x, d->A->p[d->n] * sizeof(abip_float));
00094     for (int i = d->A->p[d->n]; i < d->A->p[d->n] + d->m; i++) {
00095         data_A->x[i] = d->b[i - d->A->p[d->n]];
00096     }
00097     this_svm->data->A = data_A;
00098     this_svm->A = (ABIPMatrix *)abip_malloc(sizeof(ABIPMatrix));
00099     this_svm->b = (abip_float *)abip_malloc(this_svm->p * sizeof(abip_float));
00100     this_svm->c = (abip_float *)abip_malloc(this_svm->q * sizeof(abip_float));
00101     this_svm->D = (abip_float *)abip_malloc(this_svm->p * sizeof(abip_float));
00102
00103     this_svm->E = (abip_float *)abip_malloc(this_svm->q * sizeof(abip_float));
00104     for (int i = 0; i < this_svm->q; i++) {
00105         this_svm->E[i] = 1.0;
00106     }
00107
00108     this_svm->F = (abip_float *)abip_malloc(this_svm->m * sizeof(abip_float));
00109     this_svm->H = (abip_float *)abip_malloc(this_svm->q * sizeof(abip_float));
00110
00111     this_svm->scaling_data = &scaling_svmqp_data;

```

```

00112     this_svm->un_scaling_sol = &un_scaling_svmqp_sol;
00113     this_svm->calc_residuals = &calc_svmqp_residuals;
00114     this_svm->init_spe_linsys_work = &init_svmqp_linsys_work;
00115     this_svm->solve_spe_linsys = &solve_svmqp_linsys;
00116     this_svm->free_spe_linsys_work = &free_svmqp_linsys_work;
00117     this_svm->spe_A_times = &svmqp_A_times;
00118     this_svm->spe_AT_times = &svmqp_AT_times;
00119     this_svm->inner_conv_check = &svmqp_inner_conv_check;
00120
00121     return 0;
00122 }
00123
00128 void svmqp_A_times(svmqp *self, const abip_float *x, abip_float *y) {
00129     ABIP(accum_by_A)(self->A, x, y);
00130
00131     for (int i = 0; i < self->m; i++) {
00132         y[i] +=
00133             1 / self->D[i] * (x[self->n + 1 + i] - x[self->n + 1 + self->m + i]);
00134     }
00135 }
00136
00141 void svmqp_AT_times(svmqp *self, const abip_float *x, abip_float *y) {
00142     ABIP(accum_by_Atrans)(self->A, x, y);
00143     for (int i = 0; i < self->m; i++) {
00144         y[self->n + 1 + i] += 1 / self->D[i] * x[i];
00145         y[self->n + 1 + self->m + i] -= 1 / self->D[i] * x[i];
00146     }
00147 }
00148
00152 abip_float svmqp_inner_conv_check(svmqp *self, ABIPWork *w) {
00153     abip_int m = self->p;
00154     abip_int n = self->q;
00155
00156     abip_float *Qu = (abip_float *)abip_malloc((m + n + 1) * sizeof(abip_float));
00157     abip_float *Mu = (abip_float *)abip_malloc((m + n) * sizeof(abip_float));
00158
00159     memset(Mu, 0, (m + n) * sizeof(abip_float));
00160
00161     self->spe_A_times(self, &w->u[m], Mu);
00162
00163     self->spe_AT_times(self, w->u, &Mu[m]);
00164     ABIP(scale_array)(&Mu[m], -1, n);
00165
00166     if (self->Q != ABIP_NULL) {
00167         ABIP(accum_by_A)(self->Q, &w->u[m], &Mu[m]);
00168     }
00169
00170     memcpy(Qu, Mu, (m + n) * sizeof(abip_float));
00171
00172     ABIP(add_scaled_array)(Qu, self->b, m, -w->u[m + n]);
00173     ABIP(add_scaled_array)(&Qu[m], self->c, n, w->u[m + n]);
00174
00175     Qu[m + n] = -ABIP(dot)(w->u, Mu, m + n) / w->u[m + n] +
00176         ABIP(dot)(w->u, self->b, m) - ABIP(dot)(&w->u[m], self->c, n);
00177
00178     abip_float *tem = (abip_float *)abip_malloc((m + n + 1) * sizeof(abip_float));
00179     memcpy(tem, Qu, (m + n + 1) * sizeof(abip_float));
00180     ABIP(add_scaled_array)(tem, w->v_origin, m + n + 1, -1);
00181
00182     abip_float error_inner =
00183         ABIP(norm)(tem, m + n + 1) /
00184         (1 + ABIP(norm)(Qu, m + n + 1) + ABIP(norm)(w->v_origin, m + n + 1));
00185
00186     abip_free(Qu);
00187     abip_free(Mu);
00188     abip_free(tem);
00189
00190     return error_inner;
00191 }
00192
00196 void scaling_svmqp_data(svmqp *self, ABIPConc *k) {
00197     memcpy(self->b, self->data->b, self->p * sizeof(abip_float));
00198
00199     memcpy(self->c, self->data->c, self->q * sizeof(abip_float));
00200
00201     ABIP(copy_A_matrix)(&(self->A), self->data->A);
00202
00203     abip_int m = self->p;
00204     abip_int n = self->n + 1;
00205     ABIPMatrix *A = self->A;
00206     ABIPMatrix *Q = self->Q;
00207
00208     abip_float min_row_scale = MIN_SCALE * SQRTF((abip_float)n);
00209     abip_float max_row_scale = MAX_SCALE * SQRTF((abip_float)n);
00210     abip_float min_col_scale = MIN_SCALE * SQRTF((abip_float)m);
00211     abip_float max_col_scale = MAX_SCALE * SQRTF((abip_float)m);
00212

```

```

00213     abip_float *E_hat = self->E;
00214     abip_float *D_hat = self->D;
00215
00216     for (int i = 0; i < n; i++) {
00217         E_hat[i] = 1;
00218     }
00219     for (int i = 0; i < m; i++) {
00220         D_hat[i] = 1;
00221     }
00222
00223     abip_float *E = (abip_float *)abip_malloc(n * sizeof(abip_float));
00224     memset(E, 0, n * sizeof(abip_float));
00225
00226     abip_float *E1 = (abip_float *)abip_malloc(n * sizeof(abip_float));
00227     memset(E1, 0, n * sizeof(abip_float));
00228
00229     abip_float *E2 = (abip_float *)abip_malloc(n * sizeof(abip_float));
00230     memset(E2, 0, n * sizeof(abip_float));
00231
00232     abip_float *D = (abip_float *)abip_malloc(m * sizeof(abip_float));
00233     memset(D, 0, m * sizeof(abip_float));
00234
00235     abip_int origin_scaling = self->stgs->origin_scaling;
00236     abip_int ruiz_scaling = self->stgs->ruiz_scaling;
00237     abip_int pc_scaling = self->stgs->pc_scaling;
00238     abip_int count;
00239     abip_float mean_E;
00240
00241     if (A == ABIP_NULL && Q == ABIP_NULL) {
00242         origin_scaling = 0;
00243         ruiz_scaling = 0;
00244         pc_scaling = 0;
00245     }
00246
00247     if (ruiz_scaling) {
00248         abip_int n_ruiz = 10;
00249
00250         for (int ruiz_iter = 0; ruiz_iter < n_ruiz; ruiz_iter++) {
00251             count = 0;
00252             memset(E, 0, n * sizeof(abip_float));
00253             memset(E1, 0, n * sizeof(abip_float));
00254             memset(E2, 0, n * sizeof(abip_float));
00255             memset(D, 0, m * sizeof(abip_float));
00256
00257             if (A != ABIP_NULL) {
00258                 for (int j = 0; j < n; j++) {
00259                     if (A->p[j] == A->p[j + 1]) {
00260                         E1[j] = 0;
00261                     } else {
00262                         E1[j] =
00263                             SQRTEF(ABIP(norm_inf)(&A->x[A->p[j]], A->p[j + 1] - A->p[j]));
00264                     }
00265                 }
00266             }
00267
00268             if (Q != ABIP_NULL) {
00269                 for (int j = 0; j < n; j++) {
00270                     if (Q->p[j] == Q->p[j + 1]) {
00271                         E2[j] = 0;
00272                     } else {
00273                         E2[j] =
00274                             SQRTEF(ABIP(norm_inf)(&Q->x[Q->p[j]], Q->p[j + 1] - Q->p[j]));
00275                     }
00276                 }
00277             }
00278
00279             for (int i = 0; i < n; i++) {
00280                 E[i] = E1[i] < E2[i] ? E2[i] : E1[i];
00281             }
00282
00283             if (k->q) {
00284                 for (int i = 0; i < k->qsize; i++) {
00285                     mean_E = ABIP(vec_mean)(&E[count], k->q[i]);
00286                     for (int j = 0; j < k->q[i]; j++) {
00287                         E[j + count] = mean_E;
00288                     }
00289                     count += k->q[i];
00290                 }
00291             }
00292
00293             if (k->rq) {
00294                 for (int i = 0; i < k->rqsize; i++) {
00295                     mean_E = ABIP(vec_mean)(&E[count], k->rq[i]);
00296                     for (int j = 0; j < k->rq[i]; j++) {
00297                         E[j + count] = mean_E;
00298                     }
00299                     count += k->rq[i];

```

```

00300     }
00301 }
00302
00303 if (A != ABIP_NULL) {
00304     for (int i = 0; i < A->p[n]; i++) {
00305         if (D[A->i[i]] < ABS(A->x[i])) {
00306             D[A->i[i]] = ABS(A->x[i]);
00307         }
00308     }
00309     for (int i = 0; i < m; i++) {
00310         D[i] = SQRTF(D[i]);
00311         if (D[i] < min_row_scale)
00312             D[i] = 1;
00313         else if (D[i] > max_row_scale)
00314             D[i] = max_row_scale;
00315     }
00316
00317     for (int i = 0; i < n; i++) {
00318         if (E[i] < min_col_scale)
00319             E[i] = 1;
00320         else if (E[i] > max_col_scale)
00321             E[i] = max_col_scale;
00322         for (int j = A->p[i]; j < A->p[i + 1]; j++) {
00323             A->x[j] /= E[i];
00324         }
00325     }
00326 }
00327
00328 if (Q != ABIP_NULL) {
00329     for (int i = 0; i < n; i++) {
00330         for (int j = Q->p[i]; j < Q->p[i + 1]; j++) {
00331             Q->x[j] /= E[i];
00332         }
00333     }
00334     for (int i = 0; i < Q->p[n]; i++) {
00335         Q->x[i] /= E[Q->i[i]];
00336     }
00337 }
00338
00339 if (A != ABIP_NULL) {
00340     for (int i = 0; i < A->p[n]; i++) {
00341         A->x[i] /= D[A->i[i]];
00342     }
00343 }
00344
00345 for (int i = 0; i < n; i++) {
00346     E_hat[i] *= E[i];
00347 }
00348
00349 for (int i = 0; i < m; i++) {
00350     D_hat[i] *= D[i];
00351 }
00352 }
00353 }
00354
00355 if (origin_scaling) {
00356     memset(E, 0, n * sizeof(abip_float));
00357     memset(E1, 0, n * sizeof(abip_float));
00358     memset(E2, 0, n * sizeof(abip_float));
00359     memset(D, 0, m * sizeof(abip_float));
00360
00361     count = 0;
00362
00363     if (A != ABIP_NULL) {
00364         for (int i = 0; i < n; i++) {
00365             for (int j = A->p[i]; j < A->p[i + 1]; j++) {
00366                 E1[i] += A->x[j] * A->x[j];
00367             }
00368             E1[i] = SQRTF(E1[i]);
00369         }
00370     }
00371
00372     if (Q != ABIP_NULL) {
00373         for (int i = 0; i < n; i++) {
00374             for (int j = Q->p[i]; j < Q->p[i + 1]; j++) {
00375                 E2[i] += Q->x[j] * Q->x[j];
00376             }
00377             E2[i] = SQRTF(E2[i]);
00378         }
00379     }
00380
00381     for (int i = 0; i < n; i++) {
00382         E[i] = SQRTF(E1[i] < E2[i] ? E2[i] : E1[i]);
00383     }
00384
00385     if (k->q) {
00386         for (int i = 0; i < k->qsize; i++) {

```

```

00387     mean_E = ABIP(vec_mean)(&E[count], k->q[i]);
00388     for (int j = 0; j < k->q[i]; j++) {
00389         E[j + count] = mean_E;
00390     }
00391     count += k->q[i];
00392 }
00393 }
00394
00395 if (k->rq) {
00396     for (int i = 0; i < k->rsize; i++) {
00397         mean_E = ABIP(vec_mean)(&E[count], k->rq[i]);
00398         for (int j = 0; j < k->rq[i]; j++) {
00399             E[j + count] = mean_E;
00400         }
00401         count += k->rq[i];
00402     }
00403 }
00404
00405 if (A != ABIP_NULL) {
00406     for (int i = 0; i < A->p[n]; i++) {
00407         D[A->i[i]] += A->x[i] * A->x[i];
00408     }
00409     for (int i = 0; i < m; i++) {
00410         D[i] = SQRTF(SQRTF(D[i]));
00411         if (D[i] < min_row_scale)
00412             D[i] = 1;
00413         else if (D[i] > max_row_scale)
00414             D[i] = max_row_scale;
00415     }
00416
00417     for (int i = 0; i < n; i++) {
00418         if (E[i] < min_col_scale)
00419             E[i] = 1;
00420         else if (E[i] > max_col_scale)
00421             E[i] = max_col_scale;
00422         for (int j = A->p[i]; j < A->p[i + 1]; j++) {
00423             A->x[j] /= E[i];
00424         }
00425     }
00426 }
00427
00428 if (Q != ABIP_NULL) {
00429     for (int i = 0; i < n; i++) {
00430         for (int j = Q->p[i]; j < Q->p[i + 1]; j++) {
00431             Q->x[j] /= E[i];
00432         }
00433     }
00434     for (int i = 0; i < Q->p[n]; i++) {
00435         Q->x[i] /= E[Q->i[i]];
00436     }
00437 }
00438
00439 if (A != ABIP_NULL) {
00440     for (int i = 0; i < A->p[n]; i++) {
00441         A->x[i] /= D[A->i[i]];
00442     }
00443 }
00444
00445 for (int i = 0; i < n; i++) {
00446     E_hat[i] *= E[i];
00447 }
00448
00449 for (int i = 0; i < m; i++) {
00450     D_hat[i] *= D[i];
00451 }
00452 }
00453
00454 if (pc_scaling) {
00455     memset(E, 0, n * sizeof(abip_float));
00456     memset(E1, 0, n * sizeof(abip_float));
00457     memset(E2, 0, n * sizeof(abip_float));
00458     memset(D, 0, m * sizeof(abip_float));
00459     count = 0;
00460     abip_float alpha_pc = 1;
00461     if (A != ABIP_NULL) {
00462         for (int i = 0; i < n; i++) {
00463             for (int j = A->p[i]; j < A->p[i + 1]; j++) {
00464                 E1[i] += POWF(ABS(A->x[j]), alpha_pc);
00465             }
00466             E1[i] = SQRTF(POWF(E1[i], 1 / alpha_pc));
00467         }
00468     }
00469     if (Q != ABIP_NULL) {
00470         for (int i = 0; i < n; i++) {
00471             for (int j = Q->p[i]; j < Q->p[i + 1]; j++) {
00472                 E2[i] += POWF(ABS(Q->x[j]), alpha_pc);
00473             }

```

```

00474     E2[i] = SQRTF(POWF(E2[i], 1 / alpha_pc));
00475 }
00476 }
00477
00478 for (int i = 0; i < n; i++) {
00479     E[i] = E1[i] < E2[i] ? E2[i] : E1[i];
00480 }
00481
00482 if (k->q) {
00483     for (int i = 0; i < k->qsize; i++) {
00484         mean_E = ABIP(vec_mean)(&E[count], k->q[i]);
00485         for (int j = 0; j < k->q[i]; j++) {
00486             E[j + count] = mean_E;
00487         }
00488         count += k->q[i];
00489     }
00490 }
00491
00492 if (k->rq) {
00493     for (int i = 0; i < k->rqsize; i++) {
00494         mean_E = ABIP(vec_mean)(&E[count], k->rq[i]);
00495         for (int j = 0; j < k->rq[i]; j++) {
00496             E[j + count] = mean_E;
00497         }
00498         count += k->rq[i];
00499     }
00500 }
00501
00502 if (A != ABIP_NULL) {
00503     for (int i = 0; i < A->p[n]; i++) {
00504         D[A->i[i]] += POWF(ABS(A->x[i]), 2 - alpha_pc);
00505     }
00506     for (int i = 0; i < m; i++) {
00507         D[i] = SQRTF(POWF(D[i], 1 / (2 - alpha_pc)));
00508         if (D[i] < min_row_scale)
00509             D[i] = 1;
00510         else if (D[i] > max_row_scale)
00511             D[i] = max_row_scale;
00512     }
00513
00514     for (int i = 0; i < n; i++) {
00515         if (E[i] < min_col_scale)
00516             E[i] = 1;
00517         else if (E[i] > max_col_scale)
00518             E[i] = max_col_scale;
00519         for (int j = A->p[i]; j < A->p[i + 1]; j++) {
00520             A->x[j] /= E[i];
00521         }
00522     }
00523 }
00524
00525 if (Q != ABIP_NULL) {
00526     for (int i = 0; i < n; i++) {
00527         for (int j = Q->p[i]; j < Q->p[i + 1]; j++) {
00528             Q->x[j] /= E[i];
00529         }
00530     }
00531     for (int i = 0; i < Q->p[n]; i++) {
00532         Q->x[i] /= E[Q->i[i]];
00533     }
00534 }
00535
00536 if (A != ABIP_NULL) {
00537     for (int i = 0; i < A->p[n]; i++) {
00538         A->x[i] /= D[A->i[i]];
00539     }
00540 }
00541
00542 for (int i = 0; i < n; i++) {
00543     E_hat[i] *= E[i];
00544 }
00545
00546 for (int i = 0; i < m; i++) {
00547     D_hat[i] *= D[i];
00548 }
00549 }
00550
00551 abip_float sc =
00552     SQRTF(SQRTF(ABIP(norm_sq)(self->c, self->q) + ABIP(norm_sq)(self->b, m)));
00553
00554 if (self->b != ABIP_NULL) {
00555     for (int i = 0; i < m; i++) {
00556         self->b[i] /= D_hat[i];
00557     }
00558 }
00559
00560 for (int i = 0; i < n; i++) {

```



```

00561     self->c[i] /= E_hat[i];
00562 }
00563
00564 if (sc < MIN_SCALE)
00565     sc = 1;
00566 else if (sc > MAX_SCALE)
00567     sc = MAX_SCALE;
00568 self->sc_b = 1 / sc;
00569 self->sc_c = 1 / sc;
00570
00571 if (self->b != ABIP_NULL) {
00572     ABIP(scale_array)(self->b, self->sc_b * self->stgs->scale, m);
00573 }
00574 ABIP(scale_array)(self->c, self->sc_c * self->stgs->scale, self->q);
00575
00576 for (int i = 0; i < m; i++) {
00577     self->F[i] =
00578         self->stgs->rho_y + 2 / self->stgs->rho_x / POWF(self->D[i], 2);
00579 }
00580
00581 for (int i = 0; i < self->q; i++) {
00582     if (i < self->n)
00583         self->H[i] = self->stgs->rho_x + self->Q->x[i];
00584     else
00585         self->H[i] = self->stgs->rho_x;
00586 }
00587
00588 abip_free(E);
00589 abip_free(E1);
00590 abip_free(E2);
00591 abip_free(D);
00592 }
00593
00597 void un_scaling_svmqp_sol(svmqp *self, ABIPSolution *sol) {
00598     abip_int m = self->m;
00599     abip_int n = self->n;
00600     abip_float *x = sol->x;
00601     abip_float *y = sol->y;
00602     abip_float *s = sol->s;
00603
00604     abip_float *w = (abip_float *)abip_malloc(n * sizeof(abip_float));
00605     abip_float *b = (abip_float *)abip_malloc(sizeof(abip_float));
00606     abip_float *xi = (abip_float *)abip_malloc(m * sizeof(abip_float));
00607
00608     for (int i = 0; i < self->q; ++i) {
00609         sol->x[i] /= (self->E[i] * self->sc_b);
00610     }
00611
00612     memcpy(w, x, n * sizeof(abip_float));
00613     b[0] = x[n];
00614     memcpy(xi, &x[n + 1], m * sizeof(abip_float));
00615
00616     abip_free(x);
00617     abip_free(y);
00618     abip_free(s);
00619
00620     sol->x = w;
00621     sol->y = b;
00622     sol->s = xi;
00623 }
00624
00628 void calc_svmqp_residuals(svmqp *self, ABIPWork *w, ABIPResiduals *r,
00629                             abip_int ipm_iter, abip_int admm_iter) {
00630     DEBUG_FUNC
00631
00632     abip_int n = w->n;
00633     abip_int m = w->m;
00634
00635     abip_float *y = (abip_float *)abip_malloc(m * sizeof(abip_float));
00636     abip_float *x = (abip_float *)abip_malloc(n * sizeof(abip_float));
00637     abip_float *s = (abip_float *)abip_malloc(n * sizeof(abip_float));
00638
00639     abip_float this_pr;
00640     abip_float this_dr;
00641     abip_float this_gap;
00642
00643     if (admm_iter && r->last_admm_iter == admm_iter) {
00644         RETURN;
00645     }
00646
00647     r->last_ipm_iter = ipm_iter;
00648     r->last_admm_iter = admm_iter;
00649
00650     r->tau = ABS(w->u[n + m]);
00651     r->kappa =
00652         ABS(w->v_origin[n + m]) /
00653         (self->stgs->normalize ? (self->stgs->scale * self->sc_c * self->sc_b)

```

```

00654             : 1);
00655
00656 memcpy(y, w->u, m * sizeof(abip_float));
00657 memcpy(x, &w->u[m], n * sizeof(abip_float));
00658 memcpy(s, &w->v_origin[m], n * sizeof(abip_float));
00659
00660 ABIP(scale_array)(y, 1 / r->tau, m);
00661 ABIP(scale_array)(x, 1 / r->tau, n);
00662 ABIP(scale_array)(s, 1 / r->tau, n);
00663
00664 abip_float *Ax = (abip_float *)abip_malloc(m * sizeof(abip_float));
00665 abip_float *Ax_b = (abip_float *)abip_malloc(m * sizeof(abip_float));
00666
00667 memset(Ax, 0, m * sizeof(abip_float));
00668 self->spe_A_times(self, x, Ax);
00669
00670 memcpy(Ax_b, Ax, m * sizeof(abip_float));
00671 ABIP(add_scaled_array)(Ax_b, self->b, m, -1);
00672
00673 r->Ax_b_norm = ABIP(norm_inf)(Ax_b, m);
00674
00675 ABIP(c_dot)(Ax, self->D, m);
00676 ABIP(c_dot)(Ax_b, self->D, m);
00677
00678 this_pr = ABIP(norm_inf)(Ax_b, m) /
00679           (self->sc_b + MAX(ABIP(norm_inf)(Ax, m), self->sc_b * w->nm_inf_b));
00680
00681 abip_float *Qx = (abip_float *)abip_malloc(n * sizeof(abip_float));
00682 abip_float *ATy = (abip_float *)abip_malloc(n * sizeof(abip_float));
00683 abip_float *Qx_ATy_c_s = (abip_float *)abip_malloc(n * sizeof(abip_float));
00684
00685 memset(Qx, 0, n * sizeof(abip_float));
00686 abip_float xQx_2 = 0;
00687
00688 if (self->Q != ABIP_NULL) {
00689     ABIP(accum_by_A)(self->Q, x, Qx);
00690     xQx_2 = ABIP(dot)(x, Qx, n) / (2 * self->sc_b * self->sc_c);
00691 }
00692
00693 memset(ATy, 0, n * sizeof(abip_float));
00694 self->spe_AT_times(self, y, ATy);
00695
00696 memcpy(Qx_ATy_c_s, Qx, n * sizeof(abip_float));
00697 ABIP(add_scaled_array)(Qx_ATy_c_s, ATy, n, -1);
00698 ABIP(add_scaled_array)(Qx_ATy_c_s, self->c, n, 1);
00699 ABIP(add_scaled_array)(Qx_ATy_c_s, s, n, -1);
00700
00701 r->Qx_ATy_c_s_norm = ABIP(norm_inf)(Qx_ATy_c_s, n);
00702
00703 ABIP(c_dot)(Qx, self->E, n);
00704 ABIP(c_dot)(ATy, self->E, n);
00705 ABIP(c_dot)(Qx_ATy_c_s, self->E, n);
00706 ABIP(c_dot)(s, self->E, n);
00707
00708 this_dr = ABIP(norm_inf)(Qx_ATy_c_s, n) /
00709           (self->sc_c + MAX(self->sc_c * w->nm_inf_c, ABIP(norm_inf)(Qx, n)));
00710
00711 abip_float cTx = ABIP(dot)(self->c, x, n) / (self->sc_b * self->sc_c);
00712 abip_float bTy = ABIP(dot)(self->b, y, m) / (self->sc_b * self->sc_c);
00713
00714 this_gap = ABS(2 * xQx_2 + cTx - bTy) /
00715            (1 + MAX(2 * xQx_2, MAX(ABS(cTx), ABS(bTy))));
00716
00717 r->pobj = xQx_2 + cTx;
00718 r->dobj = -xQx_2 + bTy;
00719
00720 r->res_dif = MAX(MAX(ABS(this_pr - r->res_pri), ABS(this_dr - r->res_dual)),
00721                ABS(this_gap - r->rel_gap));
00722 r->res_pri = this_pr;
00723 r->res_dual = this_dr;
00724 r->rel_gap = this_gap;
00725 r->error_ratio =
00726     MAX(r->res_pri / self->stgs->eps_p,
00727         MAX(r->res_dual / self->stgs->eps_d, r->rel_gap / self->stgs->eps_g));
00728
00729 if (ABIP(dot)(self->c, &w->u[m], n) < 0) {
00730     ABIP(scale_array)(Qx, r->tau, n);
00731     ABIP(scale_array)(Ax, r->tau, m);
00732     r->res_unbdd = MAX(ABIP(norm)(Qx, n), ABIP(norm)(Ax, m)) /
00733                    (-ABIP(dot)(self->c, &w->u[m], n));
00734 } else {
00735     r->res_unbdd = INFINITY;
00736 }
00737
00738 if (ABIP(dot)(self->b, w->u, m) > 0) {
00739     ABIP(scale_array)(ATy, r->tau, n);
00740     ABIP(scale_array)(s, r->tau, n);

```

```

00741     ABIP(add_scaled_array) (ATy, s, n, 1);
00742
00743     r->res_infeas = ABIP(norm) (ATy, n) / ABIP(dot) (self->b, w->u, m);
00744 } else {
00745     r->res_infeas = INFINITY;
00746 }
00747
00748 abip_free(x);
00749 abip_free(y);
00750 abip_free(s);
00751 abip_free(Ax);
00752 abip_free(Ax_b);
00753 abip_free(Qx);
00754 abip_free(ATy);
00755 abip_free(Qx_ATy_c_s);
00756 }
00757
00761 cs *form_svmqp_kkt(svmqp *self) {
00762     abip_int n = self->n + 1;
00763     abip_int m = self->m;
00764     cs *LTL;
00765
00766     cs *B1 = cs_spalloc(m, n, self->A->p[n], 1, 0);
00767     memcpy(B1->i, self->A->i, self->A->p[n] * sizeof(abip_int));
00768     memcpy(B1->p, self->A->p, (n + 1) * sizeof(abip_int));
00769     memcpy(B1->x, self->A->x, self->A->p[n] * sizeof(abip_float));
00770
00771     cs *B2 = cs_spalloc(m, n, self->A->p[n], 1, 0);
00772     memcpy(B2->i, B1->i, B1->p[n] * sizeof(abip_int));
00773     memcpy(B2->p, B1->p, (n + 1) * sizeof(abip_int));
00774     memcpy(B2->x, B1->x, B1->p[n] * sizeof(abip_float));
00775
00776     if (m > n) {
00777         cs *T1 = cs_spalloc(n, n, n, 1, 1);
00778
00779         for (int i = 0; i < n; i++) {
00780             cs_entry(T1, i, i, self->H[i]);
00781         }
00782
00783         cs *eye = cs_compress(T1);
00784
00785         cs_spfree(T1);
00786
00787         for (int i = 0; i < B1->p[n]; i++) {
00788             B1->x[i] /= self->F[B1->i[i]];
00789         }
00790
00791         LTL = cs_add(eye, cs_multiply(cs_transpose(B2, 1), B1), 1, 1);
00792
00793         cs_spfree(eye);
00794     } else {
00795         for (int i = 0; i < B2->n; i++) {
00796             for (int j = B2->p[i]; j < B2->p[i + 1]; j++) {
00797                 B2->x[j] /= self->H[i];
00798             }
00799         }
00800         cs *diag = cs_spalloc(m, m, m, 1, 1);
00801
00802         for (int i = 0; i < m; i++) {
00803             cs_entry(diag, i, i, self->F[i]);
00804         }
00805         cs *diag_F = cs_compress(diag);
00806         cs_spfree(diag);
00807
00808         LTL = cs_add(diag_F, cs_multiply(B2, cs_transpose(B1, 1)), 1, 1);
00809     }
00810     cs_spfree(B2);
00811     cs_spfree(B1);
00812
00813     for (int i = 0; i < LTL->n; i++) {
00814         for (int j = LTL->p[i]; j < LTL->p[i + 1]; j++) {
00815             if (LTL->i[j] > i) LTL->x[j] = 0;
00816         }
00817     }
00818     cs_dropzeros(LTL);
00819     return LTL;
00820 }
00821
00826 void init_svmqp_precon(svmqp *self) {
00827     abip_int i;
00828
00829     self->L->M = (abip_float *)abip_malloc(self->p * sizeof(abip_float));
00830     memset(self->L->M, 0, self->p * sizeof(abip_float));
00831
00832     for (i = 0; i < self->A->n; i++) {
00833         for (int j = self->A->p[i]; j < self->A->p[i + 1]; j++) {
00834             self->L->M[self->A->i[j]] += self->A->x[j] * self->A->x[j] / self->H[i];

```

```

00835     }
00836 }
00837
00838 for (i = 0; i < self->p; i++) {
00839     self->L->M[i] += 1 / (self->D[i] * self->D[i] * self->H[self->A->n + i]);
00840     self->L->M[i] +=
00841         1 / (self->E[i] * self->E[i] * self->H[self->A->n + self->p + i]);
00842 }
00843
00844 ABIP(add_scaled_array) (self->L->M, self->rho_dr, self->p, 1.0);
00845
00846 for (i = 0; i < self->p; i++) {
00847     self->L->M[i] = 1.0 / self->L->M[i];
00848 }
00849 }
00850
00855 abip_float get_svmqp_pcg_tol(abip_int k, abip_float error_ratio,
00856                             abip_float norm_p) {
00857     if (k == -1) {
00858         return 1e-9 * norm_p;
00859     } else {
00860         return MAX(1e-9, 1e-5 * norm_p / POWF((k + 1), 2));
00861     }
00862 }
00863
00868 abip_int init_svmqp_linsys_work(svmqp *self) {
00869     if (self->stgs->linsys_solver == 0) { // mkl need lower triangle
00870         self->L->K = cs_transpose(form_svmqp_kkt(self), 1);
00871     } else if (self->stgs->linsys_solver == 1) { // qdldl need upper triangle
00872         self->L->K = form_svmqp_kkt(self);
00873     } else if (self->stgs->linsys_solver == 2) { // cholesky need upper triangle
00874         self->L->K = form_svmqp_kkt(self);
00875     } else if (self->stgs->linsys_solver == 3) { // pcg doesn't need kkt matrix
00876         init_svmqp_precon(self);
00877         self->L->K = ABIP_NULL;
00878     } else if (self->stgs->linsys_solver ==
00879         4) { // mkl-pardiso need lower triangle
00880         self->L->K = cs_transpose(form_svmqp_kkt(self), 1);
00881     } else if (self->stgs->linsys_solver ==
00882         5) { // dense cholesky need upper triangle
00883         self->L->K = form_svmqp_kkt(self);
00884     } else {
00885         printf("\nlinsys solver type error\n");
00886         return -1;
00887     }
00888     return ABIP(init_linsys_work)(self);
00889 }
00890
00894 abip_int solve_svmqp_linsys(svmqp *self, abip_float *b,
00895                             abip_float *pcg_warm_start, abip_int iter,
00896                             abip_float error_ratio) {
00897     ABIP(timer) linsys_timer;
00898     ABIP(tic) (&linsys_timer);
00899     abip_int n = self->n;
00900     abip_int m = self->m;
00901
00902     abip_int p = self->p;
00903     abip_int q = self->q;
00904
00905     if (self->stgs->linsys_solver == 3) { // pcg
00906
00907         abip_int n = self->q;
00908         abip_int m = self->p;
00909         abip_int i;
00910
00911         abip_float norm_p = ABIP(norm) (&b[m], n);
00912
00913         abip_float *tem = (abip_float *)abip_malloc(sizeof(abip_float) * m);
00914         memcpy(tem, b, m * sizeof(abip_float));
00915         for (i = 0; i < m; i++) {
00916             tem[i] /= self->rho_dr[i];
00917         }
00918         self->spe_AT_times(self, tem, &b[m]);
00919
00920         abip_free(tem);
00921
00922         abip_float pcg_tol = get_svmqp_pcg_tol(iter, error_ratio, norm_p);
00923         abip_int cg_its =
00924             ABIP(solve_linsys)(self, &b[m], n, &pcg_warm_start[m], pcg_tol);
00925
00926         if (iter >= 0) {
00927             self->L->total_cg_iters += cg_its;
00928         }
00929
00930         ABIP(scale_array) (b, -1, m);
00931         self->spe_A_times(self, &b[m], b);
00932         for (i = 0; i < m; i++) {

```

```

00933     b[i] /= -self->rho_dr[i];
00934 }
00935 } else { // direct methods
00936
00937     abip_float *b2 = (abip_float *)abip_malloc(p * sizeof(abip_float));
00938     memcpy(b2, b, p * sizeof(abip_float));
00939     abip_float *tmp = (abip_float *)abip_malloc(q * sizeof(abip_float));
00940     memcpy(tmp, &b[p], sizeof(abip_float) * q);
00941
00942     for (int i = 0; i < q; i++) {
00943         tmp[i] /= -self->H[i];
00944     }
00945
00946     self->spe_A_times(self, tmp, b2);
00947
00948     if (m > n + 1 && self->stgs->linsys_solver != 3) {
00949         for (int i = 0; i < p; i++) {
00950             b2[i] /= self->F[i];
00951         }
00952
00953         abip_float *tmp1 =
00954             (abip_float *)abip_malloc((n + 1) * sizeof(abip_float));
00955         memset(tmp1, 0, (n + 1) * sizeof(abip_float));
00956
00957         ABIP(accum_by_Atrans)(self->A, b2, tmp1);
00958
00959         ABIP(solve_linsys)(self, tmp1, n + 1, ABIP_NULL, 0);
00960
00961         abip_float *tmp2 = (abip_float *)abip_malloc(m * sizeof(abip_float));
00962         memset(tmp2, 0, m * sizeof(abip_float));
00963
00964         ABIP(accum_by_A)(self->A, tmp1, tmp2);
00965
00966         for (int i = 0; i < m; i++) {
00967             tmp2[i] /= self->F[i];
00968         }
00969
00970         ABIP(add_scaled_array)(b2, tmp2, m, -1);
00971
00972         abip_free(tmp1);
00973         abip_free(tmp2);
00974     } else {
00975         abip_int cg_its =
00976             ABIP(solve_linsys)(self, b2, m, pcg_warm_start, error_ratio);
00977         if (iter >= 0) {
00978             self->L->total_cg_its += cg_its;
00979         }
00980     }
00981
00982     memcpy(b, b2, m * sizeof(abip_float));
00983     abip_free(b2);
00984     abip_free(tmp);
00985
00986 } // if pcg
00987
00988 self->spe_AT_times(self, b, &b[p]);
00989
00990 for (int i = 0; i < q; i++) {
00991     b[p + i] /= self->H[i];
00992 }
00993
00994 self->L->total_solve_time += ABIP(tocq)(&linsys_timer);
00995
00996 return 0;
00997 }
00998
01002 void free_svmqp_linsys_work(svmqp *self) { ABIP(free_linsys)(self); }

```

## 4.195 source/util.c File Reference

```

#include "util.h"
#include "glbopts.h"
#include "linsys.h"

```

### Macros

- `#define _CRT_SECURE_NO_WARNINGS`

## Functions

- void [ABIP\(\)](#) [tic](#) ([ABIP](#)(timer) \*t)
- [abip\\_float](#) [ABIP\(\)](#) [tocq](#) ([ABIP](#)(timer) \*t)
- [abip\\_float](#) [ABIP\(\)](#) [toc](#) ([ABIP](#)(timer) \*t)
- [abip\\_float](#) [ABIP\(\)](#) [str\\_toc](#) (char \*str, [ABIP](#)(timer) \*t)
- void [ABIP\(\)](#) [print\\_work](#) (const [ABIPWork](#) \*w)
- void [ABIP\(\)](#) [print\\_data](#) (const [ABIPData](#) \*d)
- void [ABIP\(\)](#) [print\\_array](#) (const [abip\\_float](#) \*arr, [abip\\_int](#) n, const char \*name)
- void [ABIP\(\)](#) [free\\_info](#) ([ABIPInfo](#) \*info)
- void [ABIP\(\)](#) [free\\_cone](#) ([ABIPCone](#) \*k)
- void [ABIP\(\)](#) [free\\_data](#) ([ABIPData](#) \*d)
- void [ABIP\(\)](#) [free\\_sol](#) ([ABIPSolution](#) \*sol)
- void [ABIP\(\)](#) [set\\_default\\_settings](#) ([ABIPData](#) \*d)

*Default parameter settings.*

## 4.195.1 Macro Definition Documentation

### 4.195.1.1 `_CRT_SECURE_NO_WARNINGS`

```
#define _CRT_SECURE_NO_WARNINGS
```

Definition at line 1 of file [util.c](#).

## 4.195.2 Function Documentation

### 4.195.2.1 `free_cone()`

```
void ABIP\(\) free\_cone (
    ABIPCone * k )
```

Definition at line 148 of file [util.c](#).

### 4.195.2.2 `free_data()`

```
void ABIP\(\) free\_data (
    ABIPData * d )
```

Definition at line 160 of file [util.c](#).

#### 4.195.2.3 free\_info()

```
void ABIP() free_info (
    ABIPInfo * info )
```

Definition at line 142 of file [util.c](#).

#### 4.195.2.4 free\_sol()

```
void ABIP() free_sol (
    ABIPSolution * sol )
```

Definition at line 182 of file [util.c](#).

#### 4.195.2.5 print\_array()

```
void ABIP() print_array (
    const abip_float * arr,
    abip_int n,
    const char * name )
```

Definition at line 119 of file [util.c](#).

#### 4.195.2.6 print\_data()

```
void ABIP() print_data (
    const ABIPData * d )
```

Definition at line 100 of file [util.c](#).

#### 4.195.2.7 print\_work()

```
void ABIP() print_work (
    const ABIPWork * w )
```

Definition at line 80 of file [util.c](#).

#### 4.195.2.8 `set_default_settings()`

```
void ABIP() set_default_settings (
    ABIPData * d )
```

Default parameter settings.

Definition at line 203 of file [util.c](#).

#### 4.195.2.9 `str_toc()`

```
abip_float ABIP() str_toc (
    char * str,
    ABIP(timer) * t )
```

Definition at line 74 of file [util.c](#).

#### 4.195.2.10 `tic()`

```
void ABIP() tic (
    ABIP(timer) * t )
```

Definition at line 48 of file [util.c](#).

#### 4.195.2.11 `toc()`

```
abip_float ABIP() toc (
    ABIP(timer) * t )
```

Definition at line 68 of file [util.c](#).

#### 4.195.2.12 `tocq()`

```
abip_float ABIP() tocq (
    ABIP(timer) * t )
```

Definition at line 50 of file [util.c](#).



## 4.196 util.c

[Go to the documentation of this file.](#)

```

00001 #define _CRT_SECURE_NO_WARNINGS
00002 #include "util.h"
00003
00004 #include "glbopts.h"
00005 #include "linsys.h"
00006
00007 #if (defined NOTIMER)
00008
00009 void ABIP(tic) (ABIP(timer) * t) {}
00010
00011 abip_float ABIP(tocq) (ABIP(timer) * t) { return NAN; }
00012
00013 #elif (defined _WIN32 || _WIN64 || defined _WINDLL)
00014
00015 void ABIP(tic) (ABIP(timer) * t) {
00016     QueryPerformanceFrequency(&t->freq);
00017     QueryPerformanceCounter(&t->tic);
00018 }
00019
00020 abip_float ABIP(tocq) (ABIP(timer) * t) {
00021     QueryPerformanceCounter(&t->toc);
00022     return (1e3 * (t->toc.QuadPart - t->tic.QuadPart) /
00023         (abip_float)t->freq.QuadPart);
00024 }
00025
00026 #elif (defined __APPLE__)
00027
00028 void ABIP(tic) (ABIP(timer) * t) {
00029     /* read current clock cycles */
00030     t->tic = mach_absolute_time();
00031 }
00032
00033 abip_float ABIP(tocq) (ABIP(timer) * t) {
00034     uint64_t duration;
00035
00036     t->toc = mach_absolute_time();
00037     duration = t->toc - t->tic;
00038
00039     mach_timebase_info(&(t->tinfo));
00040     duration *= t->tinfo.numer;
00041     duration /= t->tinfo.denom;
00042
00043     return (abip_float)duration / 1e6;
00044 }
00045
00046 #else
00047
00048 void ABIP(tic) (ABIP(timer) * t) { clock_gettime(CLOCK_MONOTONIC, &t->tic); }
00049
00050 abip_float ABIP(tocq) (ABIP(timer) * t) {
00051     struct timespec temp;
00052
00053     clock_gettime(CLOCK_MONOTONIC, &t->toc);
00054
00055     if ((t->toc.tv_nsec - t->tic.tv_nsec) < 0) {
00056         temp.tv_sec = t->toc.tv_sec - t->tic.tv_sec - 1;
00057         temp.tv_nsec = 1e9 + t->toc.tv_nsec - t->tic.tv_nsec;
00058     } else {
00059         temp.tv_sec = t->toc.tv_sec - t->tic.tv_sec;
00060         temp.tv_nsec = t->toc.tv_nsec - t->tic.tv_nsec;
00061     }
00062
00063     return (abip_float)temp.tv_sec * 1e3 + (abip_float)temp.tv_nsec / 1e6;
00064 }
00065
00066 #endif
00067
00068 abip_float ABIP(toc) (ABIP(timer) * t) {
00069     abip_float time = ABIP(tocq) (t);
00070     abip_printf("time: %8.4f milli-seconds.\n", time);
00071     return time;
00072 }
00073
00074 abip_float ABIP(str_toc) (char *str, ABIP(timer) * t) {
00075     abip_float time = ABIP(tocq) (t);
00076     abip_printf("%s - time: %8.4f milli-seconds.\n", str, time);
00077     return time;
00078 }
00079
00080 void ABIP(print_work) (const ABIPWork *w) {
00081     abip_int i;
00082     abip_int l = w->n + w->m;

```

```

00083
00084     abip_printf("\n u_t is \n");
00085     for (i = 0; i < l; i++) {
00086         abip_printf("%f\n", w->u_t[i]);
00087     }
00088
00089     abip_printf("\n u is \n");
00090     for (i = 0; i < l; i++) {
00091         abip_printf("%f\n", w->u[i]);
00092     }
00093
00094     abip_printf("\n v is \n");
00095     for (i = 0; i < l; i++) {
00096         abip_printf("%f\n", w->v[i]);
00097     }
00098 }
00099
00100 void ABIP(print_data)(const ABIPData *d) {
00101     abip_printf("m = %i\n", (int)d->m);
00102     abip_printf("n = %i\n", (int)d->n);
00103
00104     abip_printf("max_ipm_iters = %i\n", (int)d->stgs->max_ipm_iters);
00105     abip_printf("max_admm_iters = %i\n", (int)d->stgs->max_admm_iters);
00106
00107     abip_printf("verbose = %i\n", (int)d->stgs->verbose);
00108     abip_printf("normalize = %i\n", (int)d->stgs->normalize);
00109
00110     abip_printf("eps_p = %4f\n", d->stgs->eps_p);
00111     abip_printf("eps_d = %4f\n", d->stgs->eps_d);
00112     abip_printf("eps_g = %4f\n", d->stgs->eps_g);
00113     abip_printf("eps_inf = %4f\n", d->stgs->eps_inf);
00114     abip_printf("eps_unb = %4f\n", d->stgs->eps_unb);
00115     abip_printf("alpha = %4f\n", d->stgs->alpha);
00116     abip_printf("rho_y = %4f\n", d->stgs->rho_y);
00117 }
00118
00119 void ABIP(print_array)(const abip_float *arr, abip_int n, const char *name) {
00120     abip_int i;
00121     abip_int j;
00122     abip_int k = 0;
00123
00124     abip_int num_on_one_line = 10;
00125
00126     abip_printf("\n");
00127     for (i = 0; i < n / num_on_one_line; ++i) {
00128         for (j = 0; j < num_on_one_line; ++j) {
00129             abip_printf("%s[%li] = %4f, ", name, (long)k, arr[k]);
00130             k++;
00131         }
00132         abip_printf("\n");
00133     }
00134
00135     for (j = k; j < n; ++j) {
00136         abip_printf("%s[%li] = %4f, ", name, (long)j, arr[j]);
00137     }
00138
00139     abip_printf("\n");
00140 }
00141
00142 void ABIP(free_info)(ABIPInfo *info) {
00143     if (info) {
00144         abip_free(info);
00145     }
00146 }
00147
00148 void ABIP(free_cone)(ABIPCone *k) {
00149     if (k) {
00150         if (k->q) {
00151             abip_free(k->q);
00152         }
00153         if (k->rq) {
00154             abip_free(k->rq);
00155         }
00156         abip_free(k);
00157     }
00158 }
00159
00160 void ABIP(free_data)(ABIPData *d) {
00161     if (d) {
00162         if (d->b) {
00163             abip_free(d->b);
00164         }
00165
00166         if (d->c) {
00167             abip_free(d->c);
00168         }
00169     }

```

```

00170     if (d->stgs) {
00171         abip_free(d->stgs);
00172     }
00173
00174     if (d->A) {
00175         ABIP(free_A_matrix) (d->A);
00176     }
00177     abip_free(d);
00178 }
00179 }
00180 }
00181
00182 void ABIP(free_sol) (ABIPSolution *sol) {
00183     if (sol) {
00184         if (sol->x) {
00185             abip_free(sol->x);
00186         }
00187
00188         if (sol->y) {
00189             abip_free(sol->y);
00190         }
00191
00192         if (sol->s) {
00193             abip_free(sol->s);
00194         }
00195
00196         abip_free(sol);
00197     }
00198 }
00199
00203 void ABIP(set_default_settings) (ABIPData *d) {
00204     abip_int n = d->n;
00205     abip_int m = d->m;
00206     abip_int nz = d->A == ABIP_NULL ? 0 : d->A->p[n];
00207     abip_float sparsity = (abip_float)nz / (m * n);
00208
00209     d->stgs->normalize = 1;
00210     d->stgs->scale_E = 1;
00211     d->stgs->scale_bc = 1;
00212     d->stgs->max_ipm_iters = MAX_IPM_ITERS;
00213     d->stgs->max_admm_iters = MAX_ADMM_ITERS;
00214     d->stgs->eps = EPS;
00215     d->stgs->eps_p = EPS;
00216     d->stgs->eps_d = EPS;
00217     d->stgs->eps_g = EPS;
00218     d->stgs->eps_inf = EPS;
00219     d->stgs->eps_unb = EPS;
00220     d->stgs->alpha = ALPHA;
00221     d->stgs->cg_rate = CG_RATE;
00222
00223     d->stgs->use_indirect = 0;
00224
00225     d->stgs->scale = SCALE;
00226     d->stgs->rho_y = 1e-6;
00227     d->stgs->rho_x = 1;
00228     d->stgs->rho_tau = 1;
00229     d->stgs->verbose = VERBOSE;
00230
00231     d->stgs->err_dif =
00232         0; // tol between max(dres,pres,dgap) of two consecutive inters
00233
00234     d->stgs->inner_check_period = 500;
00235     d->stgs->outer_check_period = 1;
00236
00237     // 0:mkl_dss, 1:qlddl, 2:sparse cholesky, 3:pcg, 4:pardiso, 5:dense cholesky
00238     if (m * n > 1e12) {
00239         d->stgs->linsys_solver = 3;
00240     } else if (sparsity > 0.4) {
00241         d->stgs->linsys_solver = 5;
00242     } else {
00243         d->stgs->linsys_solver = 1;
00244     }
00245
00246     d->stgs->prob_type = 3; // 0:general_qp, 1:lasso, 2:svm, 3:QCP
00247     d->stgs->time_limit = INFINITY; // in s
00248
00249     d->stgs->psi = 1; // for qp&socp
00250     // d->stgs->psi = 1.5; //for ml
00251
00252     d->stgs->origin_scaling = 1;
00253     d->stgs->ruiz_scaling = 1;
00254     d->stgs->pc_scaling = 0;
00255 }

```



# Index

- `_CRT_SECURE_NO_WARNINGS`
    - `abip.c`, [354](#)
    - `cones.c`, [373](#)
    - `linsys.c`, [403](#)
    - `util.c`, [466](#)
  - `_abip_calloc`
    - `glbopts.h`, [268](#)
  - `_abip_free`
    - `glbopts.h`, [268](#)
  - `_abip_malloc`
    - `glbopts.h`, [268](#)
  - `_abip_realloc`
    - `glbopts.h`, [268](#)
- A
  - `ABIP_PROBLEM_DATA`, [18](#)
  - `ABIP_WORK`, [31](#)
  - Lasso, [41](#)
  - qcp, [47](#)
  - `solve_specific_problem`, [52](#)
  - Svm, [59](#)
  - SVMqp, [67](#)
- a
  - `ABIP_WORK`, [31](#)
- ABIP
  - `abip.c`, [355](#)
  - `glbopts.h`, [268](#)
  - `util.h`, [311](#)
- abip
  - `abip.c`, [354](#)
  - `abip.h`, [256](#)
- abip.c
  - `_CRT_SECURE_NO_WARNINGS`, [354](#)
  - ABIP, [355](#)
  - abip, [354](#)
  - `adjust_barrier`, [355](#)
  - finish, [355](#)
  - init, [355](#)
  - `init_problem`, [356](#)
  - solve, [356](#)
  - `update_work`, [356](#)
- abip.h
  - abip, [256](#)
  - ABIPAdaptWork, [253](#)
  - ABIPCone, [253](#)
  - ABIPData, [254](#)
  - ABIPInfo, [254](#)
  - ABIPLinSysWork, [254](#)
  - ABIPMatrix, [254](#)
  - ABIPResiduals, [254](#)
  - ABIPSettings, [254](#)
  - ABIPSolution, [255](#)
  - ABIPWork, [255](#)
  - finish, [256](#)
  - init, [256](#)
  - LASSO, [256](#)
  - main, [256](#)
  - MKLLinsys, [255](#)
  - `problem_type`, [255](#)
  - QCP, [256](#)
  - solve, [257](#)
  - `spe_problem`, [255](#)
  - SVM, [256](#)
  - SVMQP, [256](#)
  - version, [257](#)
- ABIP\_A\_DATA\_MATRIX, [7](#)
  - i, [7](#)
  - m, [7](#)
  - n, [8](#)
  - p, [8](#)
  - x, [8](#)
- `abip_calloc`
  - `glbopts.h`, [268](#)
- `abip_cholsol`
  - `linsys.c`, [403](#)
- ABIP\_CONE, [8](#)
  - f, [9](#)
  - l, [9](#)
  - q, [9](#)
  - qsize, [9](#)
  - rq, [9](#)
  - rqsize, [9](#)
  - z, [10](#)
- `abip_end_interrupt_listener`
  - `ctrlc.h`, [265](#)
- ABIP\_FAILED
  - `glbopts.h`, [269](#)
- `abip_float`
  - `glbopts.h`, [277](#)
- `abip_free`
  - `glbopts.h`, [269](#)
- ABIP\_INDETERMINATE
  - `glbopts.h`, [269](#)
- ABIP\_INFEASIBLE
  - `glbopts.h`, [269](#)
- ABIP\_INFEASIBLE\_INACCURATE
  - `glbopts.h`, [269](#)
- ABIP\_INFO, [10](#)
  - `admm_iter`, [10](#)

- avg\_cg\_iters, 11
- avg\_linsys\_time, 11
- dobj, 11
- ipm\_iter, 11
- pobj, 11
- rel\_gap, 11
- res\_dual, 12
- res\_infeas, 12
- res\_pri, 12
- res\_unbdd, 12
- setup\_time, 12
- solve\_time, 12
- status, 13
- status\_val, 13
- abip\_int
  - glbopts.h, 277
- abip\_is\_interrupted
  - ctrlc.h, 265
- ABIP\_LIN\_SYS\_WORK, 13
  - bp, 14
  - ddum, 14
  - Dinv, 14
  - error, 14
  - handle, 14
  - idum, 14
  - iparm, 15
  - K, 15
  - L, 15
  - M, 15
  - maxfct, 15
  - mnum, 15
  - msglvl, 16
  - mtype, 16
  - N, 16
  - nnz\_LDL, 16
  - P, 16
  - pt, 16
  - S, 17
  - total\_cg\_iters, 17
  - total\_solve\_time, 17
  - U, 17
- abip\_make\_iso\_compilers\_happy
  - ctrlc.h, 265
- abip\_malloc
  - glbopts.h, 270
- abip\_ml\_mex.c
  - mexFunction, 327
- ABIP\_NULL
  - amd\_global.c, 124
  - amd\_internal.h, 130
  - glbopts.h, 270
- abip\_printf
  - glbopts.h, 270
- ABIP\_PROBLEM\_DATA, 17
  - A, 18
  - b, 18
  - c, 18
  - lambda, 18
  - m, 18
  - n, 19
  - Q, 19
  - stgs, 19
- abip\_qcp\_mex.c
  - mexFunction, 333
- abip\_realloc
  - glbopts.h, 270
- ABIP\_RESIDUALS, 19
  - Ax\_b\_norm, 20
  - bt\_y\_by\_tau, 20
  - ct\_x\_by\_tau, 20
  - dobj, 21
  - error\_ratio, 21
  - kap, 21
  - last\_admm\_iter, 21
  - last\_ipm\_iter, 21
  - last\_mu, 21
  - pobj, 22
  - Qx\_ATy\_c\_s\_norm, 22
  - rel\_gap, 22
  - res\_dif, 22
  - res\_dual, 22
  - res\_infeas, 22
  - res\_pri, 23
  - res\_unbdd, 23
  - tau, 23
- ABIP\_SETTINGS, 23
  - alpha, 24
  - cg\_rate, 24
  - eps, 25
  - eps\_d, 25
  - eps\_g, 25
  - eps\_inf, 25
  - eps\_p, 25
  - eps\_unb, 25
  - err\_dif, 26
  - inner\_check\_period, 26
  - linsys\_solver, 26
  - max\_admm\_iters, 26
  - max\_ipm\_iters, 26
  - normalize, 26
  - origin\_scaling, 27
  - outer\_check\_period, 27
  - pc\_scaling, 27
  - prob\_type, 27
  - psi, 27
  - rho\_tau, 27
  - rho\_x, 28
  - rho\_y, 28
  - ruiz\_scaling, 28
  - scale, 28
  - scale\_bc, 28
  - scale\_E, 28
  - time\_limit, 29
  - use\_indirect, 29
  - verbose, 29
- ABIP\_SIGINT

- glbopts.h, 270
- ABIP\_SOL\_VARS, 29
  - s, 30
  - x, 30
  - y, 30
- ABIP\_SOLVED
  - glbopts.h, 270
- ABIP\_SOLVED\_INACCURATE
  - glbopts.h, 271
- abip\_start\_interrupt\_listener
  - ctrlc.h, 265
- ABIP\_UNBOUNDED
  - glbopts.h, 271
- ABIP\_UNBOUNDED\_INACCURATE
  - glbopts.h, 271
- ABIP\_UNFINISHED
  - glbopts.h, 271
- ABIP\_UNSOLVED
  - glbopts.h, 271
- ABIP\_VERSION
  - glbopts.h, 271
- abip\_version.c
  - version, 372
- ABIP\_WORK, 30
  - A, 31
  - a, 31
  - beta, 31
  - gamma, 31
  - m, 31
  - mu, 31
  - n, 32
  - nm\_inf\_b, 32
  - nm\_inf\_c, 32
  - r, 32
  - rel\_ut, 32
  - sigma, 32
  - u, 33
  - u\_t, 33
  - v, 33
  - v\_origin, 33
- ABIPAdaptWork
  - abip.h, 253
- ABIPCone
  - abip.h, 253
- ABIPData
  - abip.h, 254
- ABIPInfo
  - abip.h, 254
- ABIPLinSysWork
  - abip.h, 254
- ABIPMatrix
  - abip.h, 254
- ABIPResiduals
  - abip.h, 254
- ABIPSettings
  - abip.h, 254
- ABIPSolution
  - abip.h, 255
- ABIPWork
  - abip.h, 255
- ABS
  - glbopts.h, 272
- accum\_by\_A
  - linsys.c, 403
  - linsys.h, 292
- accum\_by\_Atrans
  - linsys.c, 404
  - linsys.h, 293
- ADAPTIVE
  - glbopts.h, 272
- ADAPTIVE\_LOOKBACK
  - glbopts.h, 272
- add\_array
  - linalg.c, 394
  - linalg.h, 286
- add\_scaled\_array
  - linalg.c, 395
  - linalg.h, 286
- addpath
  - make\_abip\_qcp.m, 316
- adjust\_barrier
  - abip.c, 355
- admm\_iter
  - ABIP\_INFO, 10
- ALPHA
  - glbopts.h, 272
- alpha
  - ABIP\_SETTINGS, 24
- alternatively
  - make\_abip\_qcp.m, 318
- amd
  - make\_abip\_qcp.m, 318
- amd.h
  - amd\_2, 79
  - AMD\_AGGRESSIVE, 74
  - amd\_calloc, 82
  - AMD\_CONTROL, 74
  - amd\_control, 80
  - AMD\_DATE, 74
  - AMD\_DEFAULT\_AGGRESSIVE, 75
  - AMD\_DEFAULT\_DENSE, 75
  - amd\_defaults, 80
  - AMD\_DENSE, 75
  - AMD\_DMAX, 75
  - amd\_free, 82
  - AMD\_INFO, 75
  - amd\_info, 80
  - AMD\_INVALID, 75
  - amd\_l2, 80
  - amd\_l\_control, 81
  - amd\_l\_defaults, 81
  - amd\_l\_info, 81
  - amd\_l\_order, 81
  - amd\_l\_valid, 81
  - AMD\_LNZ, 76
  - AMD\_MAIN\_VERSION, 76

- amd\_malloc, 82
- AMD\_MEMORY, 76
- AMD\_N, 76
- AMD\_NCOMP, 76
- AMD\_NDENSE, 76
- AMD\_NDIV, 77
- AMD\_NMULTSUBS\_LDL, 77
- AMD\_NMULTSUBS\_LU, 77
- AMD\_NZ, 77
- AMD\_NZ\_A\_PLUS\_AT, 77
- AMD\_NZDIAG, 77
- AMD\_OK, 78
- AMD\_OK\_BUT\_JUMBLED, 78
- amd\_order, 82
- AMD\_OUT\_OF\_MEMORY, 78
- amd\_printf, 83
- amd\_realloc, 83
- AMD\_STATUS, 78
- AMD\_SUB\_VERSION, 78
- AMD\_SUBSUB\_VERSION, 78
- AMD\_SYMMETRY, 79
- amd\_valid, 82
- AMD\_VERSION, 79
- AMD\_VERSION\_CODE, 79
- EXTERN, 79
- amd/amd.h, 73, 83
- amd/amd\_1.c, 88, 89
- amd/amd\_2.c, 91, 92
- amd/amd\_aat.c, 115, 116
- amd/amd\_control.c, 118, 119
- amd/amd\_defaults.c, 119, 120
- amd/amd\_dump.c, 120, 122
- amd/amd\_global.c, 124, 126
- amd/amd\_info.c, 127, 128
- amd/amd\_internal.h, 129, 138
- amd/amd\_order.c, 142
- amd/amd\_post\_tree.c, 145, 146
- amd/amd\_postorder.c, 147, 148
- amd/amd\_preprocess.c, 151
- amd/amd\_valid.c, 153
- amd/SuiteSparse\_config.c, 155, 158
- amd/SuiteSparse\_config.h, 164, 170
- AMD\_1
  - amd\_1.c, 88
  - amd\_internal.h, 130, 136
- amd\_1.c
  - AMD\_1, 88
- AMD\_2
  - amd\_2.c, 91
  - amd\_internal.h, 130
- amd\_2
  - amd.h, 79
- amd\_2.c
  - AMD\_2, 91
- AMD\_aat
  - amd\_aat.c, 115
  - amd\_internal.h, 131, 137
- amd\_aat.c
  - AMD\_aat, 115
- AMD\_AGGRESSIVE
  - amd.h, 74
- amd\_calloc
  - amd.h, 82
  - amd\_global.c, 125
- AMD\_CONTROL
  - amd.h, 74
- AMD\_control
  - amd\_control.c, 118
  - amd\_internal.h, 131
- amd\_control
  - amd.h, 80
- amd\_control.c
  - AMD\_control, 118
- AMD\_DATE
  - amd.h, 74
- AMD\_debug
  - amd\_dump.c, 121
  - amd\_internal.h, 131
- AMD\_DEBUG0
  - amd\_internal.h, 131
- AMD\_DEBUG1
  - amd\_internal.h, 131
- AMD\_DEBUG2
  - amd\_internal.h, 131
- AMD\_DEBUG3
  - amd\_internal.h, 132
- AMD\_DEBUG4
  - amd\_internal.h, 132
- AMD\_debug\_init
  - amd\_dump.c, 121
  - amd\_internal.h, 132
- AMD\_DEFAULT\_AGGRESSIVE
  - amd.h, 75
- AMD\_DEFAULT\_DENSE
  - amd.h, 75
- AMD\_defaults
  - amd\_defaults.c, 120
  - amd\_internal.h, 132
- amd\_defaults
  - amd.h, 80
- amd\_defaults.c
  - AMD\_defaults, 120
- AMD\_DENSE
  - amd.h, 75
- AMD\_DMAX
  - amd.h, 75
- AMD\_dump
  - amd\_dump.c, 121
  - amd\_internal.h, 132
- amd\_dump.c
  - AMD\_debug, 121
  - AMD\_debug\_init, 121
  - AMD\_dump, 121
- amd\_files
  - make\_abip\_qcp.m, 318
- amd\_free



- amd.h, 82
- amd\_global.c, 125
- amd\_global.c
  - ABIP\_NULL, 124
  - amd\_calloc, 125
  - amd\_free, 125
  - amd\_malloc, 125
  - amd\_printf, 125
  - amd\_realloc, 125
- amd\_include
  - make\_abip\_qcp.m, 318
- AMD\_INFO
  - amd.h, 75
- AMD\_info
  - amd\_info.c, 127
  - amd\_internal.h, 132
- amd\_info
  - amd.h, 80
- amd\_info.c
  - AMD\_info, 127
  - PRI, 127
- amd\_internal.h
  - ABIP\_NULL, 130
  - AMD\_1, 130, 136
  - AMD\_2, 130
  - AMD\_aat, 131, 137
  - AMD\_control, 131
  - AMD\_debug, 131
  - AMD\_DEBUG0, 131
  - AMD\_DEBUG1, 131
  - AMD\_DEBUG2, 131
  - AMD\_DEBUG3, 132
  - AMD\_DEBUG4, 132
  - AMD\_debug\_init, 132
  - AMD\_defaults, 132
  - AMD\_dump, 132
  - AMD\_info, 132
  - AMD\_order, 133
  - AMD\_post\_tree, 133, 137
  - AMD\_postorder, 133, 137
  - AMD\_preprocess, 133, 137
  - AMD\_valid, 133
  - ASSERT, 133
  - EMPTY, 134
  - FALSE, 134
  - FLIP, 134
  - GLOBAL, 134
  - ID, 134
  - IMPLIES, 135
  - Int, 135
  - Int\_MAX, 135
  - MAX, 135
  - MIN, 135
  - PRINTF, 135
  - PRIVATE, 136
  - SIZE\_T\_MAX, 136
  - TRUE, 136
  - UNFLIP, 136
- AMD\_INVALID
  - amd.h, 75
- amd\_l2
  - amd.h, 80
- amd\_l\_control
  - amd.h, 81
- amd\_l\_defaults
  - amd.h, 81
- amd\_l\_info
  - amd.h, 81
- amd\_l\_order
  - amd.h, 81
- amd\_l\_valid
  - amd.h, 81
- AMD\_LNZ
  - amd.h, 76
- AMD\_MAIN\_VERSION
  - amd.h, 76
- amd\_malloc
  - amd.h, 82
  - amd\_global.c, 125
- AMD\_MEMORY
  - amd.h, 76
- AMD\_N
  - amd.h, 76
- AMD\_NCMPA
  - amd.h, 76
- AMD\_NDENSE
  - amd.h, 76
- AMD\_NDIV
  - amd.h, 77
- AMD\_NMULTSUBS\_LDL
  - amd.h, 77
- AMD\_NMULTSUBS\_LU
  - amd.h, 77
- AMD\_NZ
  - amd.h, 77
- AMD\_NZ\_A\_PLUS\_AT
  - amd.h, 77
- AMD\_NZDIAG
  - amd.h, 77
- AMD\_OK
  - amd.h, 78
- AMD\_OK\_BUT\_JUMBLED
  - amd.h, 78
- AMD\_order
  - amd\_internal.h, 133
  - amd\_order.c, 142
- amd\_order
  - amd.h, 82
- amd\_order.c
  - AMD\_order, 142
- AMD\_OUT\_OF\_MEMORY
  - amd.h, 78
- AMD\_post\_tree
  - amd\_internal.h, 133, 137
  - amd\_post\_tree.c, 145
- amd\_post\_tree.c

- AMD\_post\_tree, 145
- AMD\_postorder
  - amd\_internal.h, 133, 137
  - amd\_postorder.c, 147
- amd\_postorder.c
  - AMD\_postorder, 147
- AMD\_preprocess
  - amd\_internal.h, 133, 137
  - amd\_preprocess.c, 151
- amd\_preprocess.c
  - AMD\_preprocess, 151
- amd\_printf
  - amd.h, 83
  - amd\_global.c, 125
- amd\_realloc
  - amd.h, 83
  - amd\_global.c, 125
- AMD\_STATUS
  - amd.h, 78
- AMD\_SUB\_VERSION
  - amd.h, 78
- AMD\_SUBSUB\_VERSION
  - amd.h, 78
- AMD\_SYMMETRY
  - amd.h, 79
- AMD\_valid
  - amd\_internal.h, 133
  - amd\_valid.c, 153
- amd\_valid
  - amd.h, 82
- amd\_valid.c
  - AMD\_valid, 153
- AMD\_VERSION
  - amd.h, 79
- AMD\_VERSION\_CODE
  - amd.h, 79
- amdlist
  - make\_abip\_qcp.m, 319
- arr\_ind
  - linalg.c, 395
- ASSERT
  - amd\_internal.h, 133
- avg\_cg\_iters
  - ABIP\_INFO, 11
- avg\_linsys\_time
  - ABIP\_INFO, 11
- Ax\_b\_norm
  - ABIP\_RESIDUALS, 20
- B
  - cs\_numeric, 35
- b
  - ABIP\_PROBLEM\_DATA, 18
  - Lasso, 41
  - qcp, 47
  - solve\_specific\_problem, 52
  - Svm, 59
  - SVMqp, 67
- beta
  - ABIP\_WORK, 31
- bp
  - ABIP\_LIN\_SYS\_WORK, 14
- bt\_y\_by\_tau
  - ABIP\_RESIDUALS, 20
- c
  - ABIP\_PROBLEM\_DATA, 18
  - Lasso, 41
  - qcp, 47
  - solve\_specific\_problem, 52
  - Svm, 59
  - SVMqp, 67
- c\_dot
  - linalg.c, 395
  - linalg.h, 286
- calc\_lasso\_residuals
  - lasso\_config.c, 382
  - lasso\_config.h, 281
- calc\_qcp\_residuals
  - qcp\_config.c, 423
  - qcp\_config.h, 298
- calc\_residuals
  - Lasso, 41
  - qcp, 47
  - solve\_specific\_problem, 53
  - Svm, 59
  - SVMqp, 67
- calc\_svm\_residuals
  - svm\_config.c, 438
  - svm\_config.h, 302
- calc\_svmqp\_residuals
  - svm\_qp\_config.c, 451
  - svm\_qp\_config.h, 307
- calloc\_func
  - SuiteSparse\_config\_struct, 57
- cc
  - cs\_dmperm\_results, 34
- CG\_BEST\_TOL
  - glbopts.h, 272
- CG\_MIN\_TOL
  - glbopts.h, 272
- CG\_RATE
  - glbopts.h, 273
- cg\_rate
  - ABIP\_SETTINGS, 24
- ColMajor
  - linalg.h, 285
- cone\_norm\_1
  - linalg.c, 395
  - linalg.h, 286
- CONE\_TOL
  - glbopts.h, 273
- cones.c
  - \_CRT\_SECURE\_NO\_WARNINGS, 373
  - free\_barrier\_subproblem, 374
  - get\_cone\_dims, 374
  - get\_cone\_header, 374
  - positive\_orthant\_barrier\_subproblem, 374

- rsoc\_barrier\_subproblem, 375
- soc\_barrier\_subproblem, 375
- validate\_cones, 375
- zero\_barrier\_subproblem, 375
- cones.h
  - free\_barrier\_subproblem, 261
  - get\_cone\_dims, 262
  - get\_cone\_header, 262
  - positive\_orthant\_barrier\_subproblem, 262
  - rsoc\_barrier\_subproblem, 262
  - soc\_barrier\_subproblem, 263
  - validate\_cones, 263
  - zero\_barrier\_subproblem, 263
- CONVERGED\_INTERVAL
  - glbopts.h, 273
- copy\_A\_matrix
  - linsys.c, 404
  - linsys.h, 293
- cp
  - cs\_symbolic, 38
- cs
  - cs.h, 178
  - make\_abip\_qcp.m, 319
- cs.h
  - cs, 178
  - cs\_add, 178
  - cs\_amd, 178
  - cs\_calloc, 179
  - cs\_chol, 179
  - cs\_cholsol, 179
  - cs\_compress, 179
  - CS\_COPYRIGHT, 175
  - cs\_counts, 179
  - CS\_CSC, 175
  - cs\_cumsum, 180
  - cs\_dalloc, 180
  - CS\_DATE, 175
  - cs\_ddone, 180
  - cs\_dfree, 180
  - cs\_dfs, 180
  - cs\_dmperm, 181
  - cs\_done, 181
  - cs\_droptol, 181
  - cs\_dropzeros, 181
  - cs\_dupl, 181
  - cs\_entry, 182
  - cs\_ereach, 182
  - cs\_etree, 182
  - cs\_fkeep, 182
  - CS\_FLIP, 176
  - cs\_free, 182
  - cs\_gaxpy, 183
  - cs\_happly, 183
  - cs\_house, 183
  - cs\_idone, 183
  - cs\_ipvec, 183
  - cs\_leaf, 184
  - cs\_load, 184
  - cs\_lsolve, 184
  - cs\_ltsolve, 184
  - cs\_lu, 184
  - cs\_lusol, 185
  - cs\_malloc, 185
  - CS\_MARK, 176
  - CS\_MARKED, 176
  - CS\_MAX, 176
  - cs\_maxtrans, 185
  - CS\_MIN, 176
  - cs\_multiply, 185
  - cs\_ndone, 185
  - cs\_nfree, 186
  - cs\_norm, 186
  - cs\_permute, 186
  - cs\_pinv, 186
  - cs\_post, 186
  - cs\_print, 187
  - cs\_pvec, 187
  - cs\_qr, 187
  - cs\_qrsol, 187
  - cs\_randperm, 187
  - cs\_reach, 188
  - cs\_realloc, 188
  - cs\_scatter, 188
  - cs\_scc, 188
  - cs\_schol, 189
  - cs\_sfree, 189
  - cs\_spalloc, 189
  - cs\_spfree, 189
  - cs\_sprealloc, 189
  - cs\_spsolve, 190
  - cs\_sqr, 190
  - CS\_SUBSUB, 177
  - CS\_SUBVER, 177
  - cs\_symperm, 190
  - cs\_tdfs, 190
  - cs\_transpose, 191
  - CS\_TRIPLET, 177
  - CS\_UNFLIP, 177
  - cs\_updown, 191
  - cs\_usolve, 191
  - cs\_utsolve, 191
  - CS\_VER, 177
  - csd, 178
  - csi, 177
  - csn, 178
  - css, 178
- cs\_add
  - cs.h, 178
  - cs\_add.c, 194
- cs\_add.c
  - cs\_add, 194
- cs\_amd
  - cs.h, 178
  - cs\_amd.c, 195
- cs\_amd.c
  - cs\_amd, 195

- cs\_calloc
  - cs.h, 179
  - cs\_malloc.c, 223
- cs\_chol
  - cs.h, 179
  - cs\_chol.c, 199
- cs\_chol.c
  - cs\_chol, 199
- cs\_cholsol
  - cs.h, 179
  - cs\_cholsol.c, 201
- cs\_cholsol.c
  - cs\_cholsol, 201
- cs\_compress
  - cs.h, 179
  - cs\_compress.c, 201
- cs\_compress.c
  - cs\_compress, 201
- CS\_COPYRIGHT
  - cs.h, 175
- cs\_counts
  - cs.h, 179
  - cs\_counts.c, 203
- cs\_counts.c
  - cs\_counts, 203
  - HEAD, 202
  - NEXT, 203
- CS\_CSC
  - cs.h, 175
- cs\_cumsum
  - cs.h, 180
  - cs\_cumsum.c, 204
- cs\_cumsum.c
  - cs\_cumsum, 204
- cs\_dalloc
  - cs.h, 180
  - cs\_util.c, 248
- CS\_DATE
  - cs.h, 175
- cs\_ddone
  - cs.h, 180
  - cs\_util.c, 248
- cs\_dfree
  - cs.h, 180
  - cs\_util.c, 248
- cs\_dfs
  - cs.h, 180
  - cs\_dfs.c, 205
- cs\_dfs.c
  - cs\_dfs, 205
- cs\_dmperm
  - cs.h, 181
  - cs\_dmperm.c, 206
- cs\_dmperm.c
  - cs\_dmperm, 206
- cs\_dmperm\_results, 33
  - cc, 34
  - nb, 34
  - p, 34
  - q, 34
  - r, 34
  - rr, 35
  - s, 35
- cs\_done
  - cs.h, 181
  - cs\_util.c, 248
- cs\_droptol
  - cs.h, 181
  - cs\_droptol.c, 208
- cs\_droptol.c
  - cs\_droptol, 208
- cs\_dropzeros
  - cs.h, 181
  - cs\_dropzeros.c, 209
- cs\_dropzeros.c
  - cs\_dropzeros, 209
- cs\_dupl
  - cs.h, 181
  - cs\_dupl.c, 210
- cs\_dupl.c
  - cs\_dupl, 210
- cs\_entry
  - cs.h, 182
  - cs\_entry.c, 211
- cs\_entry.c
  - cs\_entry, 211
- cs\_ereach
  - cs.h, 182
  - cs\_ereach.c, 212
- cs\_ereach.c
  - cs\_ereach, 212
- cs\_etree
  - cs.h, 182
  - cs\_etree.c, 212
- cs\_etree.c
  - cs\_etree, 212
- cs\_files
  - make\_abip\_qcp.m, 319
- cs\_fkeep
  - cs.h, 182
  - cs\_fkeep.c, 213
- cs\_fkeep.c
  - cs\_fkeep, 213
- CS\_FLIP
  - cs.h, 176
- cs\_free
  - cs.h, 182
  - cs\_malloc.c, 223
- cs\_gaxpy
  - cs.h, 183
  - cs\_gaxpy.c, 214
- cs\_gaxpy.c
  - cs\_gaxpy, 214
- cs\_happly
  - cs.h, 183
  - cs\_happly.c, 215

- cs\_happly.c
  - cs\_happly, 215
- cs\_house
  - cs.h, 183
  - cs\_house.c, 216
- cs\_house.c
  - cs\_house, 216
- cs\_idone
  - cs.h, 183
  - cs\_util.c, 248
- cs\_include
  - make\_abip\_qcp.m, 319
- cs\_ipvec
  - cs.h, 183
  - cs\_ipvec.c, 217
- cs\_ipvec.c
  - cs\_ipvec, 217
- cs\_leaf
  - cs.h, 184
  - cs\_leaf.c, 217
- cs\_leaf.c
  - cs\_leaf, 217
- cs\_load
  - cs.h, 184
  - cs\_load.c, 218
- cs\_load.c
  - cs\_load, 218
- cs\_lsolve
  - cs.h, 184
  - cs\_lsolve.c, 219
- cs\_lsolve.c
  - cs\_lsolve, 219
- cs\_ltsolve
  - cs.h, 184
  - cs\_ltsolve.c, 220
- cs\_ltsolve.c
  - cs\_ltsolve, 220
- cs\_lu
  - cs.h, 184
  - cs\_lu.c, 221
- cs\_lu.c
  - cs\_lu, 221
- cs\_lusol
  - cs.h, 185
  - cs\_lusol.c, 222
- cs\_lusol.c
  - cs\_lusol, 222
- cs\_malloc
  - cs.h, 185
  - cs\_malloc.c, 224
- cs\_malloc.c
  - cs\_calloc, 223
  - cs\_free, 223
  - cs\_malloc, 224
  - cs\_realloc, 224
- CS\_MARK
  - cs.h, 176
- CS\_MARKED
  - cs.h, 176
- CS\_MAX
  - cs.h, 176
- cs\_maxtrans
  - cs.h, 185
  - cs\_maxtrans.c, 225
- cs\_maxtrans.c
  - cs\_maxtrans, 225
- CS\_MIN
  - cs.h, 176
- cs\_multiply
  - cs.h, 185
  - cs\_multiply.c, 226
- cs\_multiply.c
  - cs\_multiply, 226
- cs\_ndone
  - cs.h, 185
  - cs\_util.c, 249
- cs\_nfree
  - cs.h, 186
  - cs\_util.c, 249
- cs\_norm
  - cs.h, 186
  - cs\_norm.c, 227
- cs\_norm.c
  - cs\_norm, 227
- cs\_numeric, 35
  - B, 35
  - L, 36
  - pinv, 36
  - U, 36
- cs\_permute
  - cs.h, 186
  - cs\_permute.c, 228
- cs\_permute.c
  - cs\_permute, 228
- cs\_pinv
  - cs.h, 186
  - cs\_pinv.c, 229
- cs\_pinv.c
  - cs\_pinv, 229
- cs\_post
  - cs.h, 186
  - cs\_post.c, 230
- cs\_post.c
  - cs\_post, 230
- cs\_print
  - cs.h, 187
  - cs\_print.c, 231
- cs\_print.c
  - cs\_print, 231
- cs\_pvec
  - cs.h, 187
  - cs\_pvec.c, 232
- cs\_pvec.c
  - cs\_pvec, 232
- cs\_qr
  - cs.h, 187

- cs\_qr.c, [232](#)
- cs\_qr.c
  - cs\_qr, [232](#)
- cs\_qrsol
  - cs.h, [187](#)
  - cs\_qrsol.c, [234](#)
- cs\_qrsol.c
  - cs\_qrsol, [234](#)
- cs\_randperm
  - cs.h, [187](#)
  - cs\_randperm.c, [235](#)
- cs\_randperm.c
  - cs\_randperm, [235](#)
- cs\_reach
  - cs.h, [188](#)
  - cs\_reach.c, [236](#)
- cs\_reach.c
  - cs\_reach, [236](#)
- cs\_realloc
  - cs.h, [188](#)
  - cs\_malloc.c, [224](#)
- cs\_scatter
  - cs.h, [188](#)
  - cs\_scatter.c, [237](#)
- cs\_scatter.c
  - cs\_scatter, [237](#)
- cs\_scc
  - cs.h, [188](#)
  - cs\_scc.c, [238](#)
- cs\_scc.c
  - cs\_scc, [238](#)
- cs\_schol
  - cs.h, [189](#)
  - cs\_schol.c, [239](#)
- cs\_schol.c
  - cs\_schol, [239](#)
- cs\_sfree
  - cs.h, [189](#)
  - cs\_util.c, [249](#)
- cs\_spalloc
  - cs.h, [189](#)
  - cs\_util.c, [249](#)
- cs\_sparse, [36](#)
  - i, [37](#)
  - m, [37](#)
  - n, [37](#)
  - nz, [37](#)
  - nzmax, [37](#)
  - p, [37](#)
  - x, [38](#)
- cs\_spfree
  - cs.h, [189](#)
  - cs\_util.c, [249](#)
- cs\_sprealloc
  - cs.h, [189](#)
  - cs\_util.c, [250](#)
- cs\_spsolve
  - cs.h, [190](#)
- cs\_spsolve.c, [240](#)
- cs\_spsolve.c
  - cs\_spsolve, [240](#)
- cs\_sqr
  - cs.h, [190](#)
  - cs\_sqr.c, [241](#)
- cs\_sqr.c
  - cs\_sqr, [241](#)
- CS\_SUBSUB
  - cs.h, [177](#)
- CS\_SUBVER
  - cs.h, [177](#)
- cs\_symbolic, [38](#)
  - cp, [38](#)
  - leftmost, [38](#)
  - lnz, [39](#)
  - m2, [39](#)
  - parent, [39](#)
  - pinv, [39](#)
  - q, [39](#)
  - unz, [39](#)
- cs\_symperm
  - cs.h, [190](#)
  - cs\_symperm.c, [243](#)
- cs\_symperm.c
  - cs\_symperm, [243](#)
- cs\_tdfs
  - cs.h, [190](#)
  - cs\_tdfs.c, [244](#)
- cs\_tdfs.c
  - cs\_tdfs, [244](#)
- cs\_transpose
  - cs.h, [191](#)
  - cs\_transpose.c, [245](#)
- cs\_transpose.c
  - cs\_transpose, [245](#)
- CS\_TRIPLET
  - cs.h, [177](#)
- CS\_UNFLIP
  - cs.h, [177](#)
- cs\_updown
  - cs.h, [191](#)
  - cs\_updown.c, [246](#)
- cs\_updown.c
  - cs\_updown, [246](#)
- cs\_usolve
  - cs.h, [191](#)
  - cs\_usolve.c, [247](#)
- cs\_usolve.c
  - cs\_usolve, [247](#)
- cs\_util.c
  - cs\_dalloc, [248](#)
  - cs\_ddone, [248](#)
  - cs\_dfree, [248](#)
  - cs\_done, [248](#)
  - cs\_idone, [248](#)
  - cs\_ndone, [249](#)
  - cs\_nfree, [249](#)

- cs\_sfree, [249](#)
  - cs\_spallo, [249](#)
  - cs\_spfree, [249](#)
  - cs\_sprealloc, [250](#)
- cs\_utsolve
  - cs.h, [191](#)
  - cs\_utsolve.c, [252](#)
- cs\_utsolve.c
  - cs\_utsolve, [252](#)
- CS\_VER
  - cs.h, [177](#)
- csc\_to\_dense
  - linalg.c, [396](#)
  - linalg.h, [287](#)
- csd
  - cs.h, [178](#)
- csi
  - cs.h, [177](#)
- cslist
  - make\_abip\_qcp.m, [319](#)
- csn
  - cs.h, [178](#)
- csparse/Include/cs.h, [173](#), [192](#)
- csparse/Source/cs\_add.c, [193](#), [194](#)
- csparse/Source/cs\_amd.c, [194](#), [195](#)
- csparse/Source/cs\_chol.c, [199](#), [200](#)
- csparse/Source/cs\_cholsol.c, [200](#), [201](#)
- csparse/Source/cs\_compress.c, [201](#), [202](#)
- csparse/Source/cs\_counts.c, [202](#), [203](#)
- csparse/Source/cs\_cumsum.c, [204](#), [205](#)
- csparse/Source/cs\_dfs.c, [205](#), [206](#)
- csparse/Source/cs\_dmpm.c, [206](#), [207](#)
- csparse/Source/cs\_droptol.c, [208](#), [209](#)
- csparse/Source/cs\_dropzeros.c, [209](#)
- csparse/Source/cs\_dupl.c, [210](#)
- csparse/Source/cs\_entry.c, [211](#)
- csparse/Source/cs\_ereach.c, [211](#), [212](#)
- csparse/Source/cs\_etree.c, [212](#), [213](#)
- csparse/Source/cs\_fkeep.c, [213](#), [214](#)
- csparse/Source/cs\_gaxpy.c, [214](#), [215](#)
- csparse/Source/cs\_happly.c, [215](#)
- csparse/Source/cs\_house.c, [216](#)
- csparse/Source/cs\_ipvec.c, [216](#), [217](#)
- csparse/Source/cs\_leaf.c, [217](#), [218](#)
- csparse/Source/cs\_load.c, [218](#), [219](#)
- csparse/Source/cs\_lsolve.c, [219](#)
- csparse/Source/cs\_ltsolve.c, [220](#)
- csparse/Source/cs\_lu.c, [220](#), [221](#)
- csparse/Source/cs\_lusol.c, [222](#), [223](#)
- csparse/Source/cs\_malloc.c, [223](#), [224](#)
- csparse/Source/cs\_maxtrans.c, [225](#)
- csparse/Source/cs\_multiply.c, [226](#), [227](#)
- csparse/Source/cs\_norm.c, [227](#), [228](#)
- csparse/Source/cs\_permute.c, [228](#), [229](#)
- csparse/Source/cs\_pinv.c, [229](#)
- csparse/Source/cs\_post.c, [230](#)
- csparse/Source/cs\_print.c, [230](#), [231](#)
- csparse/Source/cs\_pvec.c, [231](#), [232](#)
- csparse/Source/cs\_qr.c, [232](#), [233](#)
- csparse/Source/cs\_qrsol.c, [234](#)
- csparse/Source/cs\_randperm.c, [235](#), [236](#)
- csparse/Source/cs\_reach.c, [236](#), [237](#)
- csparse/Source/cs\_scatter.c, [237](#), [238](#)
- csparse/Source/cs\_scc.c, [238](#), [239](#)
- csparse/Source/cs\_schol.c, [239](#), [240](#)
- csparse/Source/cs\_spsolve.c, [240](#), [241](#)
- csparse/Source/cs\_sqr.c, [241](#), [242](#)
- csparse/Source/cs\_symperm.c, [243](#)
- csparse/Source/cs\_tdfs.c, [244](#)
- csparse/Source/cs\_transpose.c, [245](#)
- csparse/Source/cs\_updown.c, [245](#), [246](#)
- csparse/Source/cs\_usolve.c, [246](#), [247](#)
- csparse/Source/cs\_util.c, [247](#), [250](#)
- csparse/Source/cs\_utsolve.c, [251](#), [252](#)
- css
  - cs.h, [178](#)
- ct\_x\_by\_tau
  - ABIP\_RESIDUALS, [20](#)
- ctrlc.h
  - abip\_end\_interrupt\_listener, [265](#)
  - abip\_is\_interrupted, [265](#)
  - abip\_make\_iso\_compilers\_happy, [265](#)
  - abip\_start\_interrupt\_listener, [265](#)
- D
  - Lasso, [41](#)
  - qcp, [47](#)
  - SVMqp, [68](#)
- D\_hat
  - Lasso, [41](#)
- data
  - Lasso, [42](#)
  - qcp, [47](#)
  - solve\_specific\_problem, [53](#)
  - Svm, [60](#)
  - SVMqp, [68](#)
- ddum
  - ABIP\_LIN\_SYS\_WORK, [14](#)
- debug
  - make\_abip\_qcp.m, [319](#)
- DEBUG\_FUNC
  - glbopts.h, [273](#)
- debugcommand
  - make\_abip\_qcp.m, [320](#)
- dense\_chol\_free
  - linsys.c, [404](#)
- dense\_chol\_sol
  - linsys.c, [404](#)
- Dinv
  - ABIP\_LIN\_SYS\_WORK, [14](#)
- divcomplex\_func
  - SuiteSparse\_config\_struct, [57](#)
- dojb
  - ABIP\_INFO, [11](#)
  - ABIP\_RESIDUALS, [21](#)
- dot
  - linalg.c, [396](#)

- linalg.h, [287](#)
- E
  - Lasso, [42](#)
  - qcp, [48](#)
  - SVMqp, [68](#)
- EMPTY
  - amd\_internal.h, [134](#)
- EPS
  - glbopts.h, [273](#)
- eps
  - ABIP\_SETTINGS, [25](#)
- EPS\_COR
  - glbopts.h, [273](#)
- eps\_d
  - ABIP\_SETTINGS, [25](#)
- eps\_g
  - ABIP\_SETTINGS, [25](#)
- eps\_inf
  - ABIP\_SETTINGS, [25](#)
- eps\_p
  - ABIP\_SETTINGS, [25](#)
- EPS\_PEN
  - glbopts.h, [274](#)
- EPS\_TOL
  - glbopts.h, [274](#)
- eps\_unb
  - ABIP\_SETTINGS, [25](#)
- err\_dif
  - ABIP\_SETTINGS, [26](#)
- error
  - ABIP\_LIN\_SYS\_WORK, [14](#)
  - make\_abip\_qcp.m, [316](#)
- error\_ratio
  - ABIP\_RESIDUALS, [21](#)
- eval
  - make\_abip\_qcp.m, [316](#)
- example
  - make\_abip\_qcp.m, [320](#)
- EXTERN
  - amd.h, [79](#)
- F
  - SVMqp, [68](#)
- f
  - ABIP\_CONE, [9](#)
- FALSE
  - amd\_internal.h, [134](#)
- finish
  - abip.c, [355](#)
  - abip.h, [256](#)
- FLIP
  - amd\_internal.h, [134](#)
- form\_lasso\_kkt
  - lasso\_config.c, [382](#)
- form\_qcp\_kkt
  - qcp\_config.c, [423](#)
- form\_svm\_kkt
  - svm\_config.c, [438](#)
- form\_svmqp\_kkt
  - svm\_qp\_config.c, [451](#)
- fprintf
  - make\_abip\_qcp.m, [316](#), [317](#)
- free\_A\_matrix
  - linsys.c, [405](#)
  - linsys.h, [293](#)
- free\_barrier\_subproblem
  - cones.c, [374](#)
  - cones.h, [261](#)
- free\_cone
  - util.c, [466](#)
  - util.h, [311](#)
- free\_data
  - util.c, [466](#)
  - util.h, [312](#)
- free\_func
  - SuiteSparse\_config\_struct, [57](#)
- free\_info
  - util.c, [466](#)
  - util.h, [312](#)
- free\_lasso\_linsys\_work
  - lasso\_config.c, [382](#)
  - lasso\_config.h, [281](#)
- free\_linsys
  - linsys.c, [405](#)
  - linsys.h, [293](#)
- free\_qcp\_linsys\_work
  - qcp\_config.c, [424](#)
  - qcp\_config.h, [298](#)
- free\_sol
  - util.c, [467](#)
  - util.h, [312](#)
- free\_spe\_linsys\_work
  - Lasso, [42](#)
  - qcp, [48](#)
  - solve\_specific\_problem, [53](#)
  - Svm, [60](#)
  - SVMqp, [68](#)
- free\_svm\_linsys\_work
  - svm\_config.c, [438](#)
  - svm\_config.h, [303](#)
- free\_svmqp\_linsys\_work
  - svm\_qp\_config.c, [451](#)
  - svm\_qp\_config.h, [307](#)
- GAMMA
  - glbopts.h, [274](#)
- gamma
  - ABIP\_WORK, [31](#)
- get\_cone\_dims
  - cones.c, [374](#)
  - cones.h, [262](#)
- get\_cone\_header
  - cones.c, [374](#)
  - cones.h, [262](#)
- get\_lasso\_pcg\_tol
  - lasso\_config.c, [382](#)
- get\_lin\_sys\_method



- linsys.c, [405](#)
- linsys.h, [294](#)
- get\_lin\_sys\_summary
  - linsys.c, [405](#)
  - linsys.h, [294](#)
- get\_qcp\_pcg\_tol
  - qcp\_config.c, [424](#)
- get\_svm\_pcg\_tol
  - svm\_config.c, [438](#)
- get\_svmqp\_pcg\_tol
  - svm\_qp\_config.c, [452](#)
- get\_unscaled\_s
  - lasso\_config.c, [383](#)
- get\_unscaled\_x
  - lasso\_config.c, [383](#)
- get\_unscaled\_y
  - lasso\_config.c, [383](#)
- glbopts.h
  - \_abip\_calloc, [268](#)
  - \_abip\_free, [268](#)
  - \_abip\_malloc, [268](#)
  - \_abip\_realloc, [268](#)
  - ABIP, [268](#)
  - abip\_calloc, [268](#)
  - ABIP\_FAILED, [269](#)
  - abip\_float, [277](#)
  - abip\_free, [269](#)
  - ABIP\_INDETERMINATE, [269](#)
  - ABIP\_INFEASIBLE, [269](#)
  - ABIP\_INFEASIBLE\_INACCURATE, [269](#)
  - abip\_int, [277](#)
  - abip\_malloc, [270](#)
  - ABIP\_NULL, [270](#)
  - abip\_printf, [270](#)
  - abip\_realloc, [270](#)
  - ABIP\_SIGINT, [270](#)
  - ABIP\_SOLVED, [270](#)
  - ABIP\_SOLVED\_INACCURATE, [271](#)
  - ABIP\_UNBOUNDED, [271](#)
  - ABIP\_UNBOUNDED\_INACCURATE, [271](#)
  - ABIP\_UNFINISHED, [271](#)
  - ABIP\_UNSOLVED, [271](#)
  - ABIP\_VERSION, [271](#)
  - ABS, [272](#)
  - ADAPTIVE, [272](#)
  - ADAPTIVE\_LOOKBACK, [272](#)
  - ALPHA, [272](#)
  - CG\_BEST\_TOL, [272](#)
  - CG\_MIN\_TOL, [272](#)
  - CG\_RATE, [273](#)
  - CONE\_TOL, [273](#)
  - CONVERGED\_INTERVAL, [273](#)
  - DEBUG\_FUNC, [273](#)
  - EPS, [273](#)
  - EPS\_COR, [273](#)
  - EPS\_PEN, [274](#)
  - EPS\_TOL, [274](#)
  - GAMMA, [274](#)
  - INDETERMINATE\_TOL, [274](#)
  - INFINITY, [274](#)
  - MAX, [274](#)
  - MAX\_ADMM\_ITERS, [275](#)
  - MAX\_IPM\_ITERS, [275](#)
  - MIN, [275](#)
  - NAN, [275](#)
  - NORMALIZE, [275](#)
  - POWF, [275](#)
  - RETURN, [276](#)
  - RHO\_Y, [276](#)
  - SAFEDIV\_POS, [276](#)
  - SCALE, [276](#)
  - SIGMA, [276](#)
  - SPARSITY\_RATIO, [276](#)
  - SQRTF, [277](#)
  - VERBOSE, [277](#)
  - WARM\_START, [277](#)
- GLOBAL
  - amd\_internal.h, [134](#)
- H
  - SVMqp, [68](#)
- handle
  - ABIP\_LIN\_SYS\_WORK, [14](#)
- HEAD
  - cs\_counts.c, [202](#)
- hypot\_func
  - SuiteSparse\_config\_struct, [57](#)
- i
  - ABIP\_A\_DATA\_MATRIX, [7](#)
  - cs\_sparse, [37](#)
  - make\_abip\_qcp.m, [320](#)
- ID
  - amd\_internal.h, [134](#)
- idum
  - ABIP\_LIN\_SYS\_WORK, [14](#)
- if
  - make\_abip\_qcp.m, [318](#)
- IMPLIES
  - amd\_internal.h, [135](#)
- inc
  - make\_abip\_qcp.m, [320](#)
- include/abip.h, [252](#), [257](#)
- include/amatrix.h, [260](#)
- include/cones.h, [261](#), [264](#)
- include/ctrlc.h, [264](#), [266](#)
- include/glbopts.h, [266](#), [278](#)
- include/lasso\_config.h, [280](#), [283](#)
- include/linalg.h, [284](#), [290](#)
- include/linsys.h, [292](#), [295](#)
- include/qcp\_config.h, [297](#), [300](#)
- include/svm\_config.h, [301](#), [305](#)
- include/svm\_qp\_config.h, [306](#), [310](#)
- include/util.h, [311](#), [314](#)
- INDETERMINATE\_TOL
  - glbopts.h, [274](#)
- INFINITY

- glbopts.h, 274
- init
  - abip.c, 355
  - abip.h, 256
- init\_dense\_chol
  - linsys.c, 405
- init\_lasso
  - lasso\_config.c, 383
  - lasso\_config.h, 281
- init\_lasso\_linsys\_work
  - lasso\_config.c, 384
  - lasso\_config.h, 281
- init\_lasso\_precon
  - lasso\_config.c, 384
- init\_linsys\_work
  - linsys.c, 406
  - linsys.h, 294
- init\_mkl\_work
  - linsys.c, 406
- init\_pardiso
  - linsys.c, 406
- init\_problem
  - abip.c, 356
- init\_qcp
  - qcp\_config.c, 424
  - qcp\_config.h, 298
- init\_qcp\_linsys\_work
  - qcp\_config.c, 424
  - qcp\_config.h, 298
- init\_qcp\_precon
  - qcp\_config.c, 425
- init\_spe\_linsys\_work
  - Lasso, 42
  - qcp, 48
  - solve\_specific\_problem, 53
  - Svm, 60
  - SVMqp, 69
- init\_svm
  - svm\_config.c, 438
  - svm\_config.h, 303
- init\_svm\_linsys\_work
  - svm\_config.c, 439
  - svm\_config.h, 303
- init\_svm\_precon
  - svm\_config.c, 439
- init\_svmqp
  - svm\_qp\_config.c, 452
  - svm\_qp\_config.h, 308
- init\_svmqp\_linsys\_work
  - svm\_qp\_config.c, 452
  - svm\_qp\_config.h, 308
- init\_svmqp\_precon
  - svm\_qp\_config.c, 452
- inner\_check\_period
  - ABIP\_SETTINGS, 26
- inner\_conv\_check
  - Lasso, 42
  - qcp, 48
  - solve\_specific\_problem, 53
  - Svm, 60
  - SVMqp, 69
- Int
  - amd\_internal.h, 135
- Int\_MAX
  - amd\_internal.h, 135
- intel64
  - make\_abip\_qcp.m, 320
- iparm
  - ABIP\_LIN\_SYS\_WORK, 15
- ipm\_iter
  - ABIP\_INFO, 11
- K
  - ABIP\_LIN\_SYS\_WORK, 15
- kap
  - ABIP\_RESIDUALS, 21
- L
  - ABIP\_LIN\_SYS\_WORK, 15
  - cs\_numeric, 36
  - Lasso, 42
  - qcp, 48
  - solve\_specific\_problem, 53
  - Svm, 60
  - SVMqp, 69
- I
  - ABIP\_CONE, 9
- lambda
  - ABIP\_PROBLEM\_DATA, 18
  - Lasso, 43
  - Svm, 60
  - SVMqp, 69
- LASSO
  - abip.h, 256
- Lasso, 40
  - A, 41
  - b, 41
  - c, 41
  - calc\_residuals, 41
  - D, 41
  - D\_hat, 41
  - data, 42
  - E, 42
  - free\_spe\_linsys\_work, 42
  - init\_spe\_linsys\_work, 42
  - inner\_conv\_check, 42
  - L, 42
  - lambda, 43
  - m, 43
  - n, 43
  - p, 43
  - pro\_type, 43
  - Q, 44
  - q, 43
  - rho\_dr, 44
  - sc, 44
  - sc\_b, 44

- sc\_c, 44
- sc\_cone1, 44
- sc\_cone2, 45
- scaling\_data, 45
- solve\_spe\_linsys, 45
- sparsity, 45
- spe\_A\_times, 45
- spe\_AT\_times, 45
- stgs, 46
- un\_scaling\_sol, 46
- lasso
  - lasso\_config.h, 281
- lasso\_A\_times
  - lasso\_config.c, 384
  - lasso\_config.h, 282
- lasso\_AT\_times
  - lasso\_config.c, 384
  - lasso\_config.h, 282
- lasso\_config.c
  - calc\_lasso\_residuals, 382
  - form\_lasso\_kkt, 382
  - free\_lasso\_linsys\_work, 382
  - get\_lasso\_pcg\_tol, 382
  - get\_unscaled\_s, 383
  - get\_unscaled\_x, 383
  - get\_unscaled\_y, 383
  - init\_lasso, 383
  - init\_lasso\_linsys\_work, 384
  - init\_lasso\_precon, 384
  - lasso\_A\_times, 384
  - lasso\_AT\_times, 384
  - lasso\_inner\_conv\_check, 385
  - MAX\_SCALE, 381
  - MIN\_SCALE, 381
  - scaling\_lasso\_data, 385
  - solve\_lasso\_linsys, 385
  - un\_scaling\_lasso\_sol, 385
- lasso\_config.h
  - calc\_lasso\_residuals, 281
  - free\_lasso\_linsys\_work, 281
  - init\_lasso, 281
  - init\_lasso\_linsys\_work, 281
  - lasso, 281
  - lasso\_A\_times, 282
  - lasso\_AT\_times, 282
  - lasso\_inner\_conv\_check, 282
  - scaling\_lasso\_data, 282
  - solve\_lasso\_linsys, 283
  - un\_scaling\_lasso\_sol, 283
- lasso\_inner\_conv\_check
  - lasso\_config.c, 385
  - lasso\_config.h, 282
- last\_admm\_iter
  - ABIP\_RESIDUALS, 21
- last\_ipm\_iter
  - ABIP\_RESIDUALS, 21
- last\_mu
  - ABIP\_RESIDUALS, 21
- ldl
  - make\_abip\_qcp.m, 320
- LDL\_factor
  - linsys.c, 406
  - linsys.h, 294
- ldl\_files
  - make\_abip\_qcp.m, 321
- ldl\_include
  - make\_abip\_qcp.m, 321
- ldlilst
  - make\_abip\_qcp.m, 321
- leftmost
  - cs\_symbolic, 38
- lib\_path
  - make\_abip\_qcp.m, 321
- linalg.c
  - add\_array, 394
  - add\_scaled\_array, 395
  - arr\_ind, 395
  - c\_dot, 395
  - cone\_norm\_1, 395
  - csc\_to\_dense, 396
  - dot, 396
  - norm, 396
  - norm\_1, 396
  - norm\_diff, 397
  - norm\_inf, 397
  - norm\_inf\_diff, 397
  - norm\_sq, 397
  - scale\_array, 398
  - set\_as\_scaled\_array, 398
  - set\_as\_sq, 398
  - set\_as\_sqrt, 398
  - vec\_mean, 399
- linalg.h
  - add\_array, 286
  - add\_scaled\_array, 286
  - c\_dot, 286
  - ColMajor, 285
  - cone\_norm\_1, 286
  - csc\_to\_dense, 287
  - dot, 287
  - norm, 287
  - norm\_1, 287
  - norm\_diff, 288
  - norm\_inf, 288
  - norm\_inf\_diff, 288
  - norm\_sq, 288
  - RowMajor, 285
  - scale\_array, 289
  - set\_as\_scaled\_array, 289
  - set\_as\_sq, 289
  - set\_as\_sqrt, 289
  - vec\_mean, 290
- link
  - make\_abip\_qcp.m, 321
- linsys.c
  - \_CRT\_SECURE\_NO\_WARNINGS, 403

- abip\_cholsol, [403](#)
- accum\_by\_A, [403](#)
- accum\_by\_Atrans, [404](#)
- copy\_A\_matrix, [404](#)
- dense\_chol\_free, [404](#)
- dense\_chol\_sol, [404](#)
- free\_A\_matrix, [405](#)
- free\_linsys, [405](#)
- get\_lin\_sys\_method, [405](#)
- get\_lin\_sys\_summary, [405](#)
- init\_dense\_chol, [405](#)
- init\_linsys\_work, [406](#)
- init\_mkl\_work, [406](#)
- init\_pardiso, [406](#)
- LDL\_factor, [406](#)
- MAX\_SCALE, [403](#)
- MIN\_SCALE, [403](#)
- mkl\_solve\_linsys, [406](#)
- pardiso\_free, [407](#)
- pardiso\_solve, [407](#)
- pcg, [407](#)
- permute\_kkt, [407](#)
- qcp\_pcg, [407](#)
- solve\_linsys, [408](#)
- svmqp\_pcg, [408](#)
- validate\_lin\_sys, [408](#)
- linsys.h
  - accum\_by\_A, [292](#)
  - accum\_by\_Atrans, [293](#)
  - copy\_A\_matrix, [293](#)
  - free\_A\_matrix, [293](#)
  - free\_linsys, [293](#)
  - get\_lin\_sys\_method, [294](#)
  - get\_lin\_sys\_summary, [294](#)
  - init\_linsys\_work, [294](#)
  - LDL\_factor, [294](#)
  - solve\_linsys, [294](#)
  - validate\_lin\_sys, [295](#)
- linsys\_solver
  - ABIP\_SETTINGS, [26](#)
- linux
  - make\_abip\_qcp.m, [321](#)
- lnz
  - cs\_symbolic, [39](#)
- M
  - ABIP\_LIN\_SYS\_WORK, [15](#)
- m
  - ABIP\_A\_DATA\_MATRIX, [7](#)
  - ABIP\_PROBLEM\_DATA, [18](#)
  - ABIP\_WORK, [31](#)
  - cs\_sparse, [37](#)
  - Lasso, [43](#)
  - qcp, [48](#)
  - solve\_specific\_problem, [54](#)
  - Svm, [61](#)
  - SVMqp, [69](#)
- m2
  - cs\_symbolic, [39](#)
- main
  - abip.h, [256](#)
- make\_abip\_qcp.m, [315](#)
- addpath, [316](#)
- alternatively, [318](#)
- amd, [318](#)
- amd\_files, [318](#)
- amd\_include, [318](#)
- amdlist, [319](#)
- cs, [319](#)
- cs\_files, [319](#)
- cs\_include, [319](#)
- cslist, [319](#)
- debug, [319](#)
- debugcommand, [320](#)
- error, [316](#)
- eval, [316](#)
- example, [320](#)
- fprintf, [316](#), [317](#)
- i, [320](#)
- if, [318](#)
- inc, [320](#)
- intel64, [320](#)
- ldl, [320](#)
- ldl\_files, [321](#)
- ldl\_include, [321](#)
- ldllist, [321](#)
- lib\_path, [321](#)
- link, [321](#)
- linux, [321](#)
- mex\_file, [322](#)
- mex\_type, [322](#)
- mexcommand, [322](#)
- mexfname, [322](#)
- MKL, [322](#)
- mkl\_include, [322](#)
- mkl\_lib\_path, [323](#)
- MKLROOT, [323](#)
- oneapi, [323](#)
- pamd, [323](#)
- pcs, [323](#)
- pinc, [323](#)
- platform, [324](#)
- pdl, [324](#)
- pmex, [324](#)
- psrc, [324](#)
- self, [324](#)
- src, [324](#)
- src\_files, [325](#)
- srclist, [325](#)
- malloc\_func
  - SuiteSparse\_config\_struct, [57](#)
- MAX
  - amd\_internal.h, [135](#)
  - glbopts.h, [274](#)
- MAX\_ADMM\_ITERS
  - glbopts.h, [275](#)
- max\_admm\_iters

- ABIP\_SETTINGS, 26
- MAX\_IPM\_ITERS
  - glbopts.h, 275
- max\_ipm\_iters
  - ABIP\_SETTINGS, 26
- MAX\_SCALE
  - lasso\_config.c, 381
  - linsys.c, 403
  - qcp\_config.c, 423
  - svm\_config.c, 437
  - svm\_qp\_config.c, 451
- maxfct
  - ABIP\_LIN\_SYS\_WORK, 15
- mex/abip\_ml\_mex.c, 327
- mex/abip\_qcp\_mex.c, 332, 333
- mex\_file
  - make\_abip\_qcp.m, 322
- mex\_type
  - make\_abip\_qcp.m, 322
- mexcommand
  - make\_abip\_qcp.m, 322
- mexfname
  - make\_abip\_qcp.m, 322
- mexFunction
  - abip\_ml\_mex.c, 327
  - abip\_qcp\_mex.c, 333
- MIN
  - amd\_internal.h, 135
  - glbopts.h, 275
- MIN\_SCALE
  - lasso\_config.c, 381
  - linsys.c, 403
  - qcp\_config.c, 423
  - svm\_config.c, 437
  - svm\_qp\_config.c, 451
- MKL
  - make\_abip\_qcp.m, 322
- mkl\_include
  - make\_abip\_qcp.m, 322
- mkl\_lib\_path
  - make\_abip\_qcp.m, 323
- mkl\_solve\_linsys
  - linsys.c, 406
- MKLlinsys
  - abip.h, 255
- MKLROOT
  - make\_abip\_qcp.m, 323
- mnum
  - ABIP\_LIN\_SYS\_WORK, 15
- msglvl
  - ABIP\_LIN\_SYS\_WORK, 16
- mtype
  - ABIP\_LIN\_SYS\_WORK, 16
- mu
  - ABIP\_WORK, 31
- N
  - ABIP\_LIN\_SYS\_WORK, 16
- n
  - ABIP\_A\_DATA\_MATRIX, 8
  - ABIP\_PROBLEM\_DATA, 19
  - ABIP\_WORK, 32
  - cs\_sparse, 37
  - Lasso, 43
  - qcp, 49
  - solve\_specific\_problem, 54
  - Svm, 61
  - SVMqp, 69
- NAN
  - glbopts.h, 275
- nb
  - cs\_dmperm\_results, 34
- NEXT
  - cs\_counts.c, 203
- nm\_inf\_b
  - ABIP\_WORK, 32
- nm\_inf\_c
  - ABIP\_WORK, 32
- nnz\_LDL
  - ABIP\_LIN\_SYS\_WORK, 16
- norm
  - linalg.c, 396
  - linalg.h, 287
- norm\_1
  - linalg.c, 396
  - linalg.h, 287
- norm\_diff
  - linalg.c, 397
  - linalg.h, 288
- norm\_inf
  - linalg.c, 397
  - linalg.h, 288
- norm\_inf\_diff
  - linalg.c, 397
  - linalg.h, 288
- norm\_sq
  - linalg.c, 397
  - linalg.h, 288
- NORMALIZE
  - glbopts.h, 275
- normalize
  - ABIP\_SETTINGS, 26
- nz
  - cs\_sparse, 37
- nzmax
  - cs\_sparse, 37
- oneapi
  - make\_abip\_qcp.m, 323
- origin\_scaling
  - ABIP\_SETTINGS, 27
- outer\_check\_period
  - ABIP\_SETTINGS, 27
- P
  - ABIP\_LIN\_SYS\_WORK, 16
- p
  - ABIP\_A\_DATA\_MATRIX, 8

- cs\_dmperm\_results, 34
  - cs\_sparse, 37
  - Lasso, 43
  - qcp, 49
  - solve\_specific\_problem, 54
  - Svm, 61
  - SVMqp, 70
- pamd
  - make\_abip\_qcp.m, 323
- pardiso\_free
  - linsys.c, 407
- pardiso\_solve
  - linsys.c, 407
- parent
  - cs\_symbolic, 39
- pc\_scaling
  - ABIP\_SETTINGS, 27
- pcg
  - linsys.c, 407
- pcs
  - make\_abip\_qcp.m, 323
- permute\_kkt
  - linsys.c, 407
- pinc
  - make\_abip\_qcp.m, 323
- pinv
  - cs\_numeric, 36
  - cs\_symbolic, 39
- platform
  - make\_abip\_qcp.m, 324
- pdl
  - make\_abip\_qcp.m, 324
- pmex
  - make\_abip\_qcp.m, 324
- pobj
  - ABIP\_INFO, 11
  - ABIP\_RESIDUALS, 22
- positive\_orthant\_barrier\_subproblem
  - cones.c, 374
  - cones.h, 262
- POWF
  - glbopts.h, 275
- PRI
  - amd\_info.c, 127
- print\_array
  - util.c, 467
  - util.h, 312
- print\_data
  - util.c, 467
  - util.h, 312
- print\_work
  - util.c, 467
  - util.h, 313
- PRINTF
  - amd\_internal.h, 135
- printf\_func
  - SuiteSparse\_config\_struct, 57
- PRIVATE
  - amd\_internal.h, 136
- pro\_type
  - Lasso, 43
  - qcp, 49
  - solve\_specific\_problem, 54
  - Svm, 61
  - SVMqp, 70
- prob\_type
  - ABIP\_SETTINGS, 27
- problem\_type
  - abip.h, 255
- psi
  - ABIP\_SETTINGS, 27
- psrc
  - make\_abip\_qcp.m, 324
- pt
  - ABIP\_LIN\_SYS\_WORK, 16
- Q
  - ABIP\_PROBLEM\_DATA, 19
  - Lasso, 44
  - qcp, 49
  - solve\_specific\_problem, 54
  - Svm, 61
  - SVMqp, 70
- q
  - ABIP\_CONE, 9
  - cs\_dmperm\_results, 34
  - cs\_symbolic, 39
  - Lasso, 43
  - qcp, 49
  - solve\_specific\_problem, 54
  - Svm, 61
  - SVMqp, 70
- QCP
  - abip.h, 256
- qcp, 46
  - A, 47
  - b, 47
  - c, 47
  - calc\_residuals, 47
  - D, 47
  - data, 47
  - E, 48
  - free\_spe\_linsys\_work, 48
  - init\_spe\_linsys\_work, 48
  - inner\_conv\_check, 48
  - L, 48
  - m, 48
  - n, 49
  - p, 49
  - pro\_type, 49
  - Q, 49
  - q, 49
  - qcp\_config.h, 297
  - rho\_dr, 49
  - sc\_b, 50
  - sc\_c, 50
  - scaling\_data, 50

- solve\_spe\_linsys, 50
  - sparsity, 50
  - spe\_A\_times, 50
  - spe\_AT\_times, 51
  - stgs, 51
  - un\_scaling\_sol, 51
- qcp\_A\_times
  - qcp\_config.c, 425
  - qcp\_config.h, 299
- qcp\_AT\_times
  - qcp\_config.c, 425
  - qcp\_config.h, 299
- qcp\_config.c
  - calc\_qcp\_residuals, 423
  - form\_qcp\_kkt, 423
  - free\_qcp\_linsys\_work, 424
  - get\_qcp\_pcg\_tol, 424
  - init\_qcp, 424
  - init\_qcp\_linsys\_work, 424
  - init\_qcp\_precon, 425
  - MAX\_SCALE, 423
  - MIN\_SCALE, 423
  - qcp\_A\_times, 425
  - qcp\_AT\_times, 425
  - qcp\_inner\_conv\_check, 425
  - scaling\_qcp\_data, 426
  - solve\_qcp\_linsys, 426
  - un\_scaling\_qcp\_sol, 426
- qcp\_config.h
  - calc\_qcp\_residuals, 298
  - free\_qcp\_linsys\_work, 298
  - init\_qcp, 298
  - init\_qcp\_linsys\_work, 298
  - qcp, 297
  - qcp\_A\_times, 299
  - qcp\_AT\_times, 299
  - qcp\_inner\_conv\_check, 299
  - scaling\_qcp\_data, 299
  - solve\_qcp\_linsys, 300
  - un\_scaling\_qcp\_sol, 300
- qcp\_inner\_conv\_check
  - qcp\_config.c, 425
  - qcp\_config.h, 299
- qcp\_pcg
  - linsys.c, 407
- qddl.c
  - QDLDL\_etree, 347
  - QDLDL\_factor, 347
  - QDLDL\_Lsolve, 348
  - QDLDL\_Ltsolve, 349
  - QDLDL\_solve, 349
  - QDLDL\_UNKNOWN, 346
  - QDLDL\_UNUSED, 346
  - QDLDL\_USED, 346
- qddl.h
  - QDLDL\_etree, 340
  - QDLDL\_factor, 340
  - QDLDL\_Lsolve, 341
  - QDLDL\_Ltsolve, 342
  - QDLDL\_solve, 342
  - qddl/include/qddl.h, 339, 343
  - qddl/include/qddl\_types.h, 344, 345
  - qddl/src/qddl.c, 345, 350
  - QDLDL\_bool
    - qddl\_types.h, 344
  - QDLDL\_etree
    - qddl.c, 347
    - qddl.h, 340
  - QDLDL\_factor
    - qddl.c, 347
    - qddl.h, 340
  - QDLDL\_float
    - qddl\_types.h, 345
  - QDLDL\_int
    - qddl\_types.h, 345
  - QDLDL\_INT\_MAX
    - qddl\_types.h, 344
  - QDLDL\_Lsolve
    - qddl.c, 348
    - qddl.h, 341
  - QDLDL\_Ltsolve
    - qddl.c, 349
    - qddl.h, 342
  - QDLDL\_solve
    - qddl.c, 349
    - qddl.h, 342
  - qddl\_types.h
    - QDLDL\_bool, 344
    - QDLDL\_float, 345
    - QDLDL\_int, 345
    - QDLDL\_INT\_MAX, 344
  - QDLDL\_UNKNOWN
    - qddl.c, 346
  - QDLDL\_UNUSED
    - qddl.c, 346
  - QDLDL\_USED
    - qddl.c, 346
- qsize
  - ABIP\_CONE, 9
- Qx\_ATy\_c\_s\_norm
  - ABIP\_RESIDUALS, 22
- r
  - ABIP\_WORK, 32
  - cs\_dmpmperm\_results, 34
- realloc\_func
  - SuiteSparse\_config\_struct, 58
- rel\_gap
  - ABIP\_INFO, 11
  - ABIP\_RESIDUALS, 22
- rel\_ut
  - ABIP\_WORK, 32
- res\_dif
  - ABIP\_RESIDUALS, 22
- res\_dual
  - ABIP\_INFO, 12
  - ABIP\_RESIDUALS, 22

- res\_infeas
  - ABIP\_INFO, 12
  - ABIP\_RESIDUALS, 22
- res\_pri
  - ABIP\_INFO, 12
  - ABIP\_RESIDUALS, 23
- res\_unbdd
  - ABIP\_INFO, 12
  - ABIP\_RESIDUALS, 23
- RETURN
  - glbopts.h, 276
- rho\_dr
  - Lasso, 44
  - qcp, 49
  - solve\_specific\_problem, 55
  - Svm, 62
  - SVMqp, 70
- rho\_tau
  - ABIP\_SETTINGS, 27
- rho\_x
  - ABIP\_SETTINGS, 28
- RHO\_Y
  - glbopts.h, 276
- rho\_y
  - ABIP\_SETTINGS, 28
- RowMajor
  - linalg.h, 285
- rq
  - ABIP\_CONE, 9
- rqsize
  - ABIP\_CONE, 9
- rr
  - cs\_dmpm\_results, 35
- rsoc\_barrier\_subproblem
  - cones.c, 375
  - cones.h, 262
- ruiz\_scaling
  - ABIP\_SETTINGS, 28
- S
  - ABIP\_LIN\_SYS\_WORK, 17
- s
  - ABIP\_SOL\_VARS, 30
  - cs\_dmpm\_results, 35
- SAFEDIV\_POS
  - glbopts.h, 276
- sc
  - Lasso, 44
  - Svm, 62
- sc\_b
  - Lasso, 44
  - qcp, 50
  - Svm, 62
  - SVMqp, 70
- sc\_c
  - Lasso, 44
  - qcp, 50
  - Svm, 62
  - SVMqp, 71
- sc\_cone1
  - Lasso, 44
  - Svm, 62
- sc\_cone2
  - Lasso, 45
  - Svm, 62
- sc\_D
  - Svm, 63
- sc\_E
  - Svm, 63
- sc\_F
  - Svm, 63
- SCALE
  - glbopts.h, 276
- scale
  - ABIP\_SETTINGS, 28
- scale\_array
  - linalg.c, 398
  - linalg.h, 289
- scale\_bc
  - ABIP\_SETTINGS, 28
- scale\_E
  - ABIP\_SETTINGS, 28
- scaling\_data
  - Lasso, 45
  - qcp, 50
  - solve\_specific\_problem, 55
  - Svm, 63
  - SVMqp, 71
- scaling\_lasso\_data
  - lasso\_config.c, 385
  - lasso\_config.h, 282
- scaling\_qcp\_data
  - qcp\_config.c, 426
  - qcp\_config.h, 299
- scaling\_svm\_data
  - svm\_config.c, 439
  - svm\_config.h, 303
- scaling\_svmqp\_data
  - svm\_qp\_config.c, 453
  - svm\_qp\_config.h, 308
- self
  - make\_abip\_qcp.m, 324
- set\_as\_scaled\_array
  - linalg.c, 398
  - linalg.h, 289
- set\_as\_sq
  - linalg.c, 398
  - linalg.h, 289
- set\_as\_sqrt
  - linalg.c, 398
  - linalg.h, 289
- set\_default\_settings
  - util.c, 467
  - util.h, 313
- setup\_time
  - ABIP\_INFO, 12
- SIGMA



- glbopts.h, 276
- sigma
  - ABIP\_WORK, 32
- SIZE\_T\_MAX
  - amd\_internal.h, 136
- soc\_barrier\_subproblem
  - cones.c, 375
  - cones.h, 263
- solve
  - abip.c, 356
  - abip.h, 257
- solve\_lasso\_linsys
  - lasso\_config.c, 385
  - lasso\_config.h, 283
- solve\_linsys
  - linsys.c, 408
  - linsys.h, 294
- solve\_qcp\_linsys
  - qcp\_config.c, 426
  - qcp\_config.h, 300
- solve\_spe\_linsys
  - Lasso, 45
  - qcp, 50
  - solve\_specific\_problem, 55
  - Svm, 63
  - SVMqp, 71
- solve\_specific\_problem, 51
  - A, 52
  - b, 52
  - c, 52
  - calc\_residuals, 53
  - data, 53
  - free\_spe\_linsys\_work, 53
  - init\_spe\_linsys\_work, 53
  - inner\_conv\_check, 53
  - L, 53
  - m, 54
  - n, 54
  - p, 54
  - pro\_type, 54
  - Q, 54
  - q, 54
  - rho\_dr, 55
  - scaling\_data, 55
  - solve\_spe\_linsys, 55
  - sparsity, 55
  - spe\_A\_times, 55
  - spe\_AT\_times, 55
  - stgs, 56
  - un\_scaling\_sol, 56
- solve\_svm\_linsys
  - svm\_config.c, 439
  - svm\_config.h, 304
- solve\_svmqp\_linsys
  - svm\_qp\_config.c, 453
  - svm\_qp\_config.h, 308
- solve\_time
  - ABIP\_INFO, 12
- source/abip.c, 353, 357
- source/abip\_version.c, 372, 373
- source/cones.c, 373, 376
- source/ctrlc.c, 379
- source/lasso\_config.c, 380, 386
- source/linalg.c, 394, 399
- source/linsys.c, 402, 409
- source/qcp\_config.c, 422, 427
- source/svm\_config.c, 436, 441
- source/svm\_qp\_config.c, 450, 454
- source/util.c, 465, 469
- sparsity
  - Lasso, 45
  - qcp, 50
  - solve\_specific\_problem, 55
  - Svm, 63
  - SVMqp, 71
- SPARSITY\_RATIO
  - glbopts.h, 276
- spe\_A\_times
  - Lasso, 45
  - qcp, 50
  - solve\_specific\_problem, 55
  - Svm, 64
  - SVMqp, 71
- spe\_AT\_times
  - Lasso, 45
  - qcp, 51
  - solve\_specific\_problem, 55
  - Svm, 64
  - SVMqp, 71
- spe\_problem
  - abip.h, 255
- SQRTF
  - glbopts.h, 277
- src
  - make\_abip\_qcp.m, 324
- src\_files
  - make\_abip\_qcp.m, 325
- srclist
  - make\_abip\_qcp.m, 325
- status
  - ABIP\_INFO, 13
- status\_val
  - ABIP\_INFO, 13
- stgs
  - ABIP\_PROBLEM\_DATA, 19
  - Lasso, 46
  - qcp, 51
  - solve\_specific\_problem, 56
  - Svm, 64
  - SVMqp, 72
- str\_toc
  - util.c, 468
  - util.h, 313
- SuiteSparse\_malloc
  - SuiteSparse\_config.c, 155
  - SuiteSparse\_config.h, 167

- SuiteSparse\_config
  - SuiteSparse\_config.c, [158](#)
  - SuiteSparse\_config.h, [170](#)
- SuiteSparse\_config.c
  - SuiteSparse\_calloc, [155](#)
  - SuiteSparse\_config, [158](#)
  - SuiteSparse\_divcomplex, [155](#)
  - SuiteSparse\_finish, [156](#)
  - SuiteSparse\_free, [156](#)
  - SuiteSparse\_hypot, [156](#)
  - SuiteSparse\_malloc, [156](#)
  - SuiteSparse\_realloc, [156](#)
  - SuiteSparse\_start, [157](#)
  - SuiteSparse\_tic, [157](#)
  - SuiteSparse\_time, [157](#)
  - SuiteSparse\_toc, [157](#)
  - SuiteSparse\_version, [157](#)
- SuiteSparse\_config.h
  - SuiteSparse\_calloc, [167](#)
  - SuiteSparse\_config, [170](#)
  - SUITESPARSE\_DATE, [165](#)
  - SuiteSparse\_divcomplex, [168](#)
  - SuiteSparse\_finish, [168](#)
  - SuiteSparse\_free, [168](#)
  - SUITESPARSE\_HAS\_VERSION\_FUNCTION, [165](#)
  - SuiteSparse\_hypot, [168](#)
  - SuiteSparse\_long, [166](#)
  - SuiteSparse\_long\_id, [166](#)
  - SuiteSparse\_long\_idd, [166](#)
  - SuiteSparse\_long\_max, [166](#)
  - SUITESPARSE\_MAIN\_VERSION, [166](#)
  - SuiteSparse\_malloc, [168](#)
  - SUITESPARSE\_PRINTF, [166](#)
  - SuiteSparse\_realloc, [169](#)
  - SuiteSparse\_start, [169](#)
  - SUITESPARSE\_SUB\_VERSION, [167](#)
  - SUITESPARSE\_SUBSUB\_VERSION, [167](#)
  - SuiteSparse\_tic, [169](#)
  - SuiteSparse\_time, [169](#)
  - SuiteSparse\_toc, [169](#)
  - SUITESPARSE\_VER\_CODE, [167](#)
  - SUITESPARSE\_VERSION, [167](#)
  - SuiteSparse\_version, [170](#)
- SuiteSparse\_config\_struct, [56](#)
  - calloc\_func, [57](#)
  - divcomplex\_func, [57](#)
  - free\_func, [57](#)
  - hypot\_func, [57](#)
  - malloc\_func, [57](#)
  - printf\_func, [57](#)
  - realloc\_func, [58](#)
- SUITESPARSE\_DATE
  - SuiteSparse\_config.h, [165](#)
- SuiteSparse\_divcomplex
  - SuiteSparse\_config.c, [155](#)
  - SuiteSparse\_config.h, [168](#)
- SuiteSparse\_finish
  - SuiteSparse\_config.c, [156](#)
- SuiteSparse\_free
  - SuiteSparse\_config.c, [156](#)
  - SuiteSparse\_config.h, [168](#)
- SUITESPARSE\_HAS\_VERSION\_FUNCTION
  - SuiteSparse\_config.h, [165](#)
- SuiteSparse\_hypot
  - SuiteSparse\_config.c, [156](#)
  - SuiteSparse\_config.h, [168](#)
- SuiteSparse\_long
  - SuiteSparse\_config.h, [166](#)
- SuiteSparse\_long\_id
  - SuiteSparse\_config.h, [166](#)
- SuiteSparse\_long\_idd
  - SuiteSparse\_config.h, [166](#)
- SuiteSparse\_long\_max
  - SuiteSparse\_config.h, [166](#)
- SUITESPARSE\_MAIN\_VERSION
  - SuiteSparse\_config.h, [166](#)
- SuiteSparse\_malloc
  - SuiteSparse\_config.c, [156](#)
  - SuiteSparse\_config.h, [168](#)
- SUITESPARSE\_PRINTF
  - SuiteSparse\_config.h, [166](#)
- SuiteSparse\_realloc
  - SuiteSparse\_config.c, [156](#)
  - SuiteSparse\_config.h, [169](#)
- SuiteSparse\_start
  - SuiteSparse\_config.c, [157](#)
  - SuiteSparse\_config.h, [169](#)
- SUITESPARSE\_SUB\_VERSION
  - SuiteSparse\_config.h, [167](#)
- SUITESPARSE\_SUBSUB\_VERSION
  - SuiteSparse\_config.h, [167](#)
- SuiteSparse\_tic
  - SuiteSparse\_config.c, [157](#)
  - SuiteSparse\_config.h, [169](#)
- SuiteSparse\_time
  - SuiteSparse\_config.c, [157](#)
  - SuiteSparse\_config.h, [169](#)
- SuiteSparse\_toc
  - SuiteSparse\_config.c, [157](#)
  - SuiteSparse\_config.h, [169](#)
- SUITESPARSE\_VER\_CODE
  - SuiteSparse\_config.h, [167](#)
- SUITESPARSE\_VERSION
  - SuiteSparse\_config.h, [167](#)
- SuiteSparse\_version
  - SuiteSparse\_config.c, [157](#)
  - SuiteSparse\_config.h, [170](#)
- SVM
  - abip.h, [256](#)
- Svm, [58](#)
  - A, [59](#)
  - b, [59](#)
  - c, [59](#)
  - calc\_residuals, [59](#)
  - data, [60](#)

- free\_spe\_linsys\_work, 60
- init\_spe\_linsys\_work, 60
- inner\_conv\_check, 60
- L, 60
- lambda, 60
- m, 61
- n, 61
- p, 61
- pro\_type, 61
- Q, 61
- q, 61
- rho\_dr, 62
- sc, 62
- sc\_b, 62
- sc\_c, 62
- sc\_cone1, 62
- sc\_cone2, 62
- sc\_D, 63
- sc\_E, 63
- sc\_F, 63
- scaling\_data, 63
- solve\_spe\_linsys, 63
- sparsity, 63
- spe\_A\_times, 64
- spe\_AT\_times, 64
- stgs, 64
- un\_scaling\_sol, 64
- wA, 64
- wB, 64
- wC, 65
- wD, 65
- wE, 65
- wF, 65
- wG, 65
- wH, 65
- wX, 66
- wy, 66
- svm
  - svm\_config.h, 302
- svm\_A\_times
  - svm\_config.c, 440
  - svm\_config.h, 304
- svm\_AT\_times
  - svm\_config.c, 440
  - svm\_config.h, 304
- svm\_config.c
  - calc\_svm\_residuals, 438
  - form\_svm\_kkt, 438
  - free\_svm\_linsys\_work, 438
  - get\_svm\_pcg\_tol, 438
  - init\_svm, 438
  - init\_svm\_linsys\_work, 439
  - init\_svm\_precon, 439
  - MAX\_SCALE, 437
  - MIN\_SCALE, 437
  - scaling\_svm\_data, 439
  - solve\_svm\_linsys, 439
  - svm\_A\_times, 440
  - svm\_AT\_times, 440
  - svm\_inner\_conv\_check, 440
  - un\_scaling\_svm\_sol, 440
- svm\_config.h
  - calc\_svm\_residuals, 302
  - free\_svm\_linsys\_work, 303
  - init\_svm, 303
  - init\_svm\_linsys\_work, 303
  - scaling\_svm\_data, 303
  - solve\_svm\_linsys, 304
  - svm, 302
  - svm\_A\_times, 304
  - svm\_AT\_times, 304
  - svm\_inner\_conv\_check, 304
  - un\_scaling\_svm\_sol, 305
- svm\_inner\_conv\_check
  - svm\_config.c, 440
  - svm\_config.h, 304
- svm\_qp\_config.c
  - calc\_svmqp\_residuals, 451
  - form\_svmqp\_kkt, 451
  - free\_svmqp\_linsys\_work, 451
  - get\_svmqp\_pcg\_tol, 452
  - init\_svmqp, 452
  - init\_svmqp\_linsys\_work, 452
  - init\_svmqp\_precon, 452
  - MAX\_SCALE, 451
  - MIN\_SCALE, 451
  - scaling\_svmqp\_data, 453
  - solve\_svmqp\_linsys, 453
  - svmqp\_A\_times, 453
  - svmqp\_AT\_times, 453
  - svmqp\_inner\_conv\_check, 454
  - un\_scaling\_svmqp\_sol, 454
- svm\_qp\_config.h
  - calc\_svmqp\_residuals, 307
  - free\_svmqp\_linsys\_work, 307
  - init\_svmqp, 308
  - init\_svmqp\_linsys\_work, 308
  - scaling\_svmqp\_data, 308
  - solve\_svmqp\_linsys, 308
  - svmqp, 307
  - svmqp\_A\_times, 309
  - svmqp\_AT\_times, 309
  - svmqp\_inner\_conv\_check, 309
  - un\_scaling\_svmqp\_sol, 309
- SVMQP
  - abip.h, 256
- SVMqp, 66
  - A, 67
  - b, 67
  - c, 67
  - calc\_residuals, 67
  - D, 68
  - data, 68
  - E, 68
  - F, 68
  - free\_spe\_linsys\_work, 68

- H, 68
- init\_spe\_linsys\_work, 69
- inner\_conv\_check, 69
- L, 69
- lambda, 69
- m, 69
- n, 69
- p, 70
- pro\_type, 70
- Q, 70
- q, 70
- rho\_dr, 70
- sc\_b, 70
- sc\_c, 71
- scaling\_data, 71
- solve\_spe\_linsys, 71
- sparsity, 71
- spe\_A\_times, 71
- spe\_AT\_times, 71
- stgs, 72
- un\_scaling\_sol, 72
- svmqp
  - svm\_qp\_config.h, 307
- svmqp\_A\_times
  - svm\_qp\_config.c, 453
  - svm\_qp\_config.h, 309
- svmqp\_AT\_times
  - svm\_qp\_config.c, 453
  - svm\_qp\_config.h, 309
- svmqp\_inner\_conv\_check
  - svm\_qp\_config.c, 454
  - svm\_qp\_config.h, 309
- svmqp\_pcg
  - linsys.c, 408
- tau
  - ABIP\_RESIDUALS, 23
- tic
  - util.c, 468
  - util.h, 313
- time\_limit
  - ABIP\_SETTINGS, 29
- toc
  - util.c, 468
  - util.h, 313
- tocq
  - util.c, 468
  - util.h, 314
- total\_cg\_iters
  - ABIP\_LIN\_SYS\_WORK, 17
- total\_solve\_time
  - ABIP\_LIN\_SYS\_WORK, 17
- TRUE
  - amd\_internal.h, 136
- U
  - ABIP\_LIN\_SYS\_WORK, 17
  - cs\_numeric, 36
- u
  - ABIP\_WORK, 33
- u\_t
  - ABIP\_WORK, 33
- un\_scaling\_lasso\_sol
  - lasso\_config.c, 385
  - lasso\_config.h, 283
- un\_scaling\_qcp\_sol
  - qcp\_config.c, 426
  - qcp\_config.h, 300
- un\_scaling\_sol
  - Lasso, 46
  - qcp, 51
  - solve\_specific\_problem, 56
  - Svm, 64
  - SVMqp, 72
- un\_scaling\_svm\_sol
  - svm\_config.c, 440
  - svm\_config.h, 305
- un\_scaling\_svmqp\_sol
  - svm\_qp\_config.c, 454
  - svm\_qp\_config.h, 309
- UNFLIP
  - amd\_internal.h, 136
- unz
  - cs\_symbolic, 39
- update\_work
  - abip.c, 356
- use\_indirect
  - ABIP\_SETTINGS, 29
- util.c
  - \_CRT\_SECURE\_NO\_WARNINGS, 466
  - free\_cone, 466
  - free\_data, 466
  - free\_info, 466
  - free\_sol, 467
  - print\_array, 467
  - print\_data, 467
  - print\_work, 467
  - set\_default\_settings, 467
  - str\_toc, 468
  - tic, 468
  - toc, 468
  - tocq, 468
- util.h
  - ABIP, 311
  - free\_cone, 311
  - free\_data, 312
  - free\_info, 312
  - free\_sol, 312
  - print\_array, 312
  - print\_data, 312
  - print\_work, 313
  - set\_default\_settings, 313
  - str\_toc, 313
  - tic, 313
  - toc, 313
  - tocq, 314
- v

- ABIP\_WORK, [33](#)
- v\_origin
  - ABIP\_WORK, [33](#)
- validate\_cones
  - cones.c, [375](#)
  - cones.h, [263](#)
- validate\_lin\_sys
  - linsys.c, [408](#)
  - linsys.h, [295](#)
- vec\_mean
  - linalg.c, [399](#)
  - linalg.h, [290](#)
- VERBOSE
  - glbopts.h, [277](#)
- verbose
  - ABIP\_SETTINGS, [29](#)
- version
  - abip.h, [257](#)
  - abip\_version.c, [372](#)
- wA
  - Svm, [64](#)
- WARM\_START
  - glbopts.h, [277](#)
- wB
  - Svm, [64](#)
- wC
  - Svm, [65](#)
- wD
  - Svm, [65](#)
- wE
  - Svm, [65](#)
- wF
  - Svm, [65](#)
- wG
  - Svm, [65](#)
- wH
  - Svm, [65](#)
- wX
  - Svm, [66](#)
- wy
  - Svm, [66](#)
- x
  - ABIP\_A\_DATA\_MATRIX, [8](#)
  - ABIP\_SOL\_VARS, [30](#)
  - cs\_sparse, [38](#)
- y
  - ABIP\_SOL\_VARS, [30](#)
- z
  - ABIP\_CONE, [10](#)
- zero\_barrier\_subproblem
  - cones.c, [375](#)
  - cones.h, [263](#)