

ABIP-LP

2.0.0

Generated by Doxygen 1.9.5

| | |
|---|----------|
| 1 README | 1 |
| 2 Data Structure Index | 3 |
| 2.1 Data Structures | 3 |
| 3 File Index | 5 |
| 3.1 File List | 5 |
| 4 Data Structure Documentation | 7 |
| 4.1 ABIP_A_DATA_MATRIX Struct Reference | 7 |
| 4.1.1 Detailed Description | 7 |
| 4.1.2 Field Documentation | 7 |
| 4.1.2.1 i | 7 |
| 4.1.2.2 m | 8 |
| 4.1.2.3 n | 8 |
| 4.1.2.4 p | 8 |
| 4.1.2.5 x | 8 |
| 4.2 ABIP_ADAPTIVE_WORK Struct Reference | 8 |
| 4.2.1 Detailed Description | 9 |
| 4.2.2 Field Documentation | 9 |
| 4.2.2.1 delta_u | 9 |
| 4.2.2.2 delta_ut | 9 |
| 4.2.2.3 delta_v | 9 |
| 4.2.2.4 k | 9 |
| 4.2.2.5 l | 9 |
| 4.2.2.6 total_adapt_time | 10 |
| 4.2.2.7 u | 10 |
| 4.2.2.8 u_next | 10 |
| 4.2.2.9 u_prev | 10 |
| 4.2.2.10 ut | 10 |
| 4.2.2.11 ut_next | 10 |
| 4.2.2.12 v | 11 |
| 4.2.2.13 v_next | 11 |
| 4.2.2.14 v_prev | 11 |
| 4.3 ABIP_INFO Struct Reference | 11 |
| 4.3.1 Detailed Description | 12 |
| 4.3.2 Field Documentation | 12 |
| 4.3.2.1 admm_iter | 12 |
| 4.3.2.2 dobj | 12 |
| 4.3.2.3 ipm_iter | 12 |
| 4.3.2.4 pobj | 12 |
| 4.3.2.5 rel_gap | 12 |
| 4.3.2.6 res_dual | 13 |

| | |
|--|----|
| 4.3.2.7 res_infeas | 13 |
| 4.3.2.8 res_pri | 13 |
| 4.3.2.9 res_unbdd | 13 |
| 4.3.2.10 setup_time | 13 |
| 4.3.2.11 solve_time | 13 |
| 4.3.2.12 status | 14 |
| 4.3.2.13 status_val | 14 |
| 4.4 ABIP_LIN_SYS_WORK Struct Reference | 14 |
| 4.4.1 Detailed Description | 14 |
| 4.4.2 Field Documentation | 14 |
| 4.4.2.1 At | 15 |
| 4.4.2.2 bp | 15 |
| 4.4.2.3 D | 15 |
| 4.4.2.4 Gp | 15 |
| 4.4.2.5 i | 15 |
| 4.4.2.6 j | 15 |
| 4.4.2.7 L | 16 |
| 4.4.2.8 M | 16 |
| 4.4.2.9 P | 16 |
| 4.4.2.10 p | 16 |
| 4.4.2.11 pardiso_work | 16 |
| 4.4.2.12 r | 16 |
| 4.4.2.13 tmp | 17 |
| 4.4.2.14 tot_cg_its | 17 |
| 4.4.2.15 total_solve_time | 17 |
| 4.4.2.16 z | 17 |
| 4.5 ABIP_PROBLEM_DATA Struct Reference | 17 |
| 4.5.1 Detailed Description | 18 |
| 4.5.2 Field Documentation | 18 |
| 4.5.2.1 A | 18 |
| 4.5.2.2 b | 18 |
| 4.5.2.3 c | 18 |
| 4.5.2.4 m | 18 |
| 4.5.2.5 n | 18 |
| 4.5.2.6 sp | 19 |
| 4.5.2.7 stgs | 19 |
| 4.6 ABIP_RESIDUALS Struct Reference | 19 |
| 4.6.1 Detailed Description | 19 |
| 4.6.2 Field Documentation | 19 |
| 4.6.2.1 bt_y_by_tau | 20 |
| 4.6.2.2 ct_x_by_tau | 20 |
| 4.6.2.3 kap | 20 |

| | |
|--|----|
| 4.6.2.4 last_admm_iter | 20 |
| 4.6.2.5 last_ipm_iter | 20 |
| 4.6.2.6 last_mu | 20 |
| 4.6.2.7 rel_gap | 21 |
| 4.6.2.8 res_dual | 21 |
| 4.6.2.9 res_infeas | 21 |
| 4.6.2.10 res_pri | 21 |
| 4.6.2.11 res_unbdd | 21 |
| 4.6.2.12 tau | 21 |
| 4.7 ABIP_SCALING Struct Reference | 22 |
| 4.7.1 Detailed Description | 22 |
| 4.7.2 Field Documentation | 22 |
| 4.7.2.1 D | 22 |
| 4.7.2.2 E | 22 |
| 4.7.2.3 mean_norm_col_A | 22 |
| 4.7.2.4 mean_norm_row_A | 23 |
| 4.8 ABIP_SETTINGS Struct Reference | 23 |
| 4.8.1 Detailed Description | 24 |
| 4.8.2 Field Documentation | 24 |
| 4.8.2.1 adaptive | 24 |
| 4.8.2.2 adaptive_lookback | 24 |
| 4.8.2.3 alpha | 24 |
| 4.8.2.4 avg_criterion | 24 |
| 4.8.2.5 cg_rate | 24 |
| 4.8.2.6 dynamic_eta | 25 |
| 4.8.2.7 dynamic_sigma | 25 |
| 4.8.2.8 dynamic_sigma_second | 25 |
| 4.8.2.9 dynamic_x | 25 |
| 4.8.2.10 eps | 25 |
| 4.8.2.11 eps_cor | 25 |
| 4.8.2.12 eps_pen | 26 |
| 4.8.2.13 half_update | 26 |
| 4.8.2.14 hybrid_mu | 26 |
| 4.8.2.15 hybrid_thresh | 26 |
| 4.8.2.16 max_admm_iters | 26 |
| 4.8.2.17 max_ipm_iters | 26 |
| 4.8.2.18 max_time | 27 |
| 4.8.2.19 normalize | 27 |
| 4.8.2.20 origin_rescale | 27 |
| 4.8.2.21 pc_ruiz_rescale | 27 |
| 4.8.2.22 pfeasopt | 27 |
| 4.8.2.23 qp_rescale | 27 |

| | |
|--|----|
| 4.8.2.24 restart_fre | 28 |
| 4.8.2.25 restart_thresh | 28 |
| 4.8.2.26 rho_y | 28 |
| 4.8.2.27 ruiz_iter | 28 |
| 4.8.2.28 scale | 28 |
| 4.8.2.29 sparsity_ratio | 28 |
| 4.8.2.30 verbose | 29 |
| 4.8.2.31 warm_start | 29 |
| 4.9 ABIP_SOL_VARS Struct Reference | 29 |
| 4.9.1 Detailed Description | 29 |
| 4.9.2 Field Documentation | 29 |
| 4.9.2.1 s | 29 |
| 4.9.2.2 x | 30 |
| 4.9.2.3 y | 30 |
| 4.10 ABIP_WORK Struct Reference | 30 |
| 4.10.1 Detailed Description | 31 |
| 4.10.2 Field Documentation | 31 |
| 4.10.2.1 A | 31 |
| 4.10.2.2 adapt | 31 |
| 4.10.2.3 b | 31 |
| 4.10.2.4 beta | 32 |
| 4.10.2.5 c | 32 |
| 4.10.2.6 double_check | 32 |
| 4.10.2.7 dr | 32 |
| 4.10.2.8 final_check | 32 |
| 4.10.2.9 fre_old | 32 |
| 4.10.2.10 g | 33 |
| 4.10.2.11 g_th | 33 |
| 4.10.2.12 gamma | 33 |
| 4.10.2.13 h | 33 |
| 4.10.2.14 m | 33 |
| 4.10.2.15 mu | 33 |
| 4.10.2.16 n | 34 |
| 4.10.2.17 nm_b | 34 |
| 4.10.2.18 nm_c | 34 |
| 4.10.2.19 p | 34 |
| 4.10.2.20 pr | 34 |
| 4.10.2.21 sc_b | 34 |
| 4.10.2.22 sc_c | 35 |
| 4.10.2.23 scal | 35 |
| 4.10.2.24 sigma | 35 |
| 4.10.2.25 sp | 35 |

| | |
|---|-----------|
| 4.10.2.26 stgs | 35 |
| 4.10.2.27 u | 35 |
| 4.10.2.28 u_avg | 36 |
| 4.10.2.29 u_avgcon | 36 |
| 4.10.2.30 u_prev | 36 |
| 4.10.2.31 u_sumcon | 36 |
| 4.10.2.32 u_t | 36 |
| 4.10.2.33 v | 36 |
| 4.10.2.34 v_avg | 37 |
| 4.10.2.35 v_avgcon | 37 |
| 4.10.2.36 v_prev | 37 |
| 4.10.2.37 v_sumcon | 37 |
| 4.11 SuiteSparse_config_struct Struct Reference | 37 |
| 4.11.1 Detailed Description | 38 |
| 4.11.2 Field Documentation | 38 |
| 4.11.2.1 calloc_func | 38 |
| 4.11.2.2 divcomplex_func | 38 |
| 4.11.2.3 free_func | 38 |
| 4.11.2.4 hypot_func | 38 |
| 4.11.2.5 malloc_func | 38 |
| 4.11.2.6 printf_func | 39 |
| 4.11.2.7 realloc_func | 39 |
| 5 File Documentation | 41 |
| 5.1 compile_direct.m File Reference | 41 |
| 5.1.1 Function Documentation | 42 |
| 5.1.1.1 compile_direct() | 42 |
| 5.1.1.2 eval() | 42 |
| 5.1.1.3 exist() | 42 |
| 5.1.1.4 fprintf() | 42 |
| 5.1.1.5 if() | 42 |
| 5.1.2 Variable Documentation | 42 |
| 5.1.2.1 abip_amd | 43 |
| 5.1.2.2 abip_ldl | 43 |
| 5.1.2.3 alternatively | 43 |
| 5.1.2.4 amd_files | 43 |
| 5.1.2.5 amd_path | 43 |
| 5.1.2.6 cmd | 44 |
| 5.1.2.7 example | 44 |
| 5.1.2.8 following | 44 |
| 5.1.2.9 intel64 | 44 |
| 5.1.2.10 ld_files | 44 |

| | |
|---|----|
| 5.1.2.11 ldl_path | 45 |
| 5.1.2.12 lib_path | 45 |
| 5.1.2.13 link | 45 |
| 5.1.2.14 linux | 45 |
| 5.1.2.15 mexext | 45 |
| 5.1.2.16 MKL | 46 |
| 5.1.2.17 mkl_macro | 46 |
| 5.1.2.18 oneapi | 46 |
| 5.1.2.19 pardiso_src | 46 |
| 5.1.2.20 platform | 46 |
| 5.1.2.21 self | 46 |
| 5.2 compile_direct.m | 47 |
| 5.3 compile_indirect.m File Reference | 48 |
| 5.3.1 Function Documentation | 48 |
| 5.3.1.1 compile_indirect() | 48 |
| 5.3.1.2 eval() | 48 |
| 5.3.1.3 if() | 48 |
| 5.3.2 Variable Documentation | 48 |
| 5.3.2.1 cmd | 49 |
| 5.4 compile_indirect.m | 49 |
| 5.5 external/amd/amd.h File Reference | 49 |
| 5.5.1 Macro Definition Documentation | 50 |
| 5.5.1.1 AMD_AGGRESSIVE | 51 |
| 5.5.1.2 AMD_CONTROL | 51 |
| 5.5.1.3 AMD_DATE | 51 |
| 5.5.1.4 AMD_DEFAULT_AGGRESSIVE | 51 |
| 5.5.1.5 AMD_DEFAULT_DENSE | 51 |
| 5.5.1.6 AMD_DENSE | 51 |
| 5.5.1.7 AMD_DMAX | 52 |
| 5.5.1.8 AMD_INFO | 52 |
| 5.5.1.9 AMD_INVALID | 52 |
| 5.5.1.10 AMD_LNZ | 52 |
| 5.5.1.11 AMD_MAIN_VERSION | 52 |
| 5.5.1.12 AMD_MEMORY | 52 |
| 5.5.1.13 AMD_N | 53 |
| 5.5.1.14 AMD_NCPA | 53 |
| 5.5.1.15 AMD_NDENSE | 53 |
| 5.5.1.16 AMD_NDIV | 53 |
| 5.5.1.17 AMD_NMULTSUBS_LDL | 53 |
| 5.5.1.18 AMD_NMULTSUBS_LU | 53 |
| 5.5.1.19 AMD_NZ | 54 |
| 5.5.1.20 AMD_NZ_A_PLUS_AT | 54 |

| | |
|--|----|
| 5.5.1.21 AMD_NZDIAG | 54 |
| 5.5.1.22 AMD_OK | 54 |
| 5.5.1.23 AMD_OK_BUT_JUMBLED | 54 |
| 5.5.1.24 AMD_OUT_OF_MEMORY | 54 |
| 5.5.1.25 AMD_STATUS | 55 |
| 5.5.1.26 AMD_SUB_VERSION | 55 |
| 5.5.1.27 AMD_SUBSUB_VERSION | 55 |
| 5.5.1.28 AMD_SYMMETRY | 55 |
| 5.5.1.29 AMD_VERSION | 55 |
| 5.5.1.30 AMD_VERSION_CODE | 55 |
| 5.5.1.31 EXTERN | 56 |
| 5.5.2 Function Documentation | 56 |
| 5.5.2.1 amd_2() | 56 |
| 5.5.2.2 amd_control() | 56 |
| 5.5.2.3 amd_defaults() | 56 |
| 5.5.2.4 amd_info() | 56 |
| 5.5.2.5 amd_l2() | 57 |
| 5.5.2.6 amd_l_control() | 57 |
| 5.5.2.7 amd_l_defaults() | 57 |
| 5.5.2.8 amd_l_info() | 57 |
| 5.5.2.9 amd_l_order() | 57 |
| 5.5.2.10 amd_l_valid() | 58 |
| 5.5.2.11 amd_order() | 58 |
| 5.5.2.12 amd_valid() | 58 |
| 5.5.3 Variable Documentation | 58 |
| 5.5.3.1 amd_calloc | 58 |
| 5.5.3.2 amd_free | 58 |
| 5.5.3.3 amd_malloc | 59 |
| 5.5.3.4 amd_printf | 59 |
| 5.5.3.5 amd_realloc | 59 |
| 5.6 amd.h | 59 |
| 5.7 external/amd/amd_1.c File Reference | 64 |
| 5.7.1 Function Documentation | 64 |
| 5.7.1.1 AMD_1() | 64 |
| 5.8 amd_1.c | 65 |
| 5.9 external/amd/amd_2.c File Reference | 67 |
| 5.9.1 Function Documentation | 67 |
| 5.9.1.1 AMD_2() | 67 |
| 5.10 amd_2.c | 68 |
| 5.11 external/amd/amd_aat.c File Reference | 91 |
| 5.11.1 Function Documentation | 91 |
| 5.11.1.1 AMD_aat() | 91 |

| | |
|---|-----|
| 5.12 amd_aat.c | 92 |
| 5.13 external/amd/amd_control.c File Reference | 94 |
| 5.13.1 Function Documentation | 94 |
| 5.13.1.1 AMD_control() | 94 |
| 5.14 amd_control.c | 95 |
| 5.15 external/amd/amd_defaults.c File Reference | 95 |
| 5.15.1 Function Documentation | 96 |
| 5.15.1.1 AMD_defaults() | 96 |
| 5.16 amd_defaults.c | 96 |
| 5.17 external/amd/amd_dump.c File Reference | 96 |
| 5.17.1 Function Documentation | 97 |
| 5.17.1.1 AMD_debug_init() | 97 |
| 5.17.1.2 AMD_dump() | 97 |
| 5.17.2 Variable Documentation | 97 |
| 5.17.2.1 AMD_debug | 97 |
| 5.18 amd_dump.c | 98 |
| 5.19 external/amd/amd_global.c File Reference | 100 |
| 5.19.1 Macro Definition Documentation | 100 |
| 5.19.1.1 ABIP_NULL | 100 |
| 5.19.2 Variable Documentation | 101 |
| 5.19.2.1 amd_calloc | 101 |
| 5.19.2.2 amd_free | 101 |
| 5.19.2.3 amd_malloc | 101 |
| 5.19.2.4 amd_printf | 101 |
| 5.19.2.5 amd_realloc | 101 |
| 5.20 amd_global.c | 102 |
| 5.21 external/amd/amd_info.c File Reference | 103 |
| 5.21.1 Macro Definition Documentation | 103 |
| 5.21.1.1 PRI | 103 |
| 5.21.2 Function Documentation | 103 |
| 5.21.2.1 AMD_info() | 103 |
| 5.22 amd_info.c | 104 |
| 5.23 external/amd/amd_internal.h File Reference | 105 |
| 5.23.1 Macro Definition Documentation | 106 |
| 5.23.1.1 ABIP_NULL | 106 |
| 5.23.1.2 AMD_1 | 106 |
| 5.23.1.3 AMD_2 | 107 |
| 5.23.1.4 AMD_aat | 107 |
| 5.23.1.5 AMD_control | 107 |
| 5.23.1.6 AMD_debug | 107 |
| 5.23.1.7 AMD_DEBUG0 | 107 |
| 5.23.1.8 AMD_DEBUG1 | 107 |

| | |
|--|-----|
| 5.23.1.9 AMD_DEBUG2 | 108 |
| 5.23.1.10 AMD_DEBUG3 | 108 |
| 5.23.1.11 AMD_DEBUG4 | 108 |
| 5.23.1.12 AMD_debug_init | 108 |
| 5.23.1.13 AMD_defaults | 108 |
| 5.23.1.14 AMD_dump | 108 |
| 5.23.1.15 AMD_info | 109 |
| 5.23.1.16 AMD_order | 109 |
| 5.23.1.17 AMD_post_tree | 109 |
| 5.23.1.18 AMD_postorder | 109 |
| 5.23.1.19 AMD_preprocess | 109 |
| 5.23.1.20 AMD_valid | 109 |
| 5.23.1.21 ASSERT | 110 |
| 5.23.1.22 EMPTY [1/2] | 110 |
| 5.23.1.23 EMPTY [2/2] | 110 |
| 5.23.1.24 FALSE | 110 |
| 5.23.1.25 FLIP | 110 |
| 5.23.1.26 GLOBAL | 110 |
| 5.23.1.27 ID | 111 |
| 5.23.1.28 IMPLIES | 111 |
| 5.23.1.29 Int | 111 |
| 5.23.1.30 Int_MAX | 111 |
| 5.23.1.31 MAX | 111 |
| 5.23.1.32 MIN | 111 |
| 5.23.1.33 PRINTF | 112 |
| 5.23.1.34 PRIVATE | 112 |
| 5.23.1.35 SIZE_T_MAX | 112 |
| 5.23.1.36 TRUE | 112 |
| 5.23.1.37 UNFLIP | 112 |
| 5.23.2 Function Documentation | 112 |
| 5.23.2.1 AMD_1() | 113 |
| 5.23.2.2 AMD_aat() | 113 |
| 5.23.2.3 AMD_post_tree() | 113 |
| 5.23.2.4 AMD_postorder() | 113 |
| 5.23.2.5 AMD_preprocess() | 114 |
| 5.24 amd_internal.h | 114 |
| 5.25 external/amd/amd_order.c File Reference | 118 |
| 5.25.1 Function Documentation | 118 |
| 5.25.1.1 AMD_order() | 118 |
| 5.26 amd_order.c | 118 |
| 5.27 external/amd/amd_post_tree.c File Reference | 121 |
| 5.27.1 Function Documentation | 121 |

| | |
|---|-----|
| 5.27.1.1 AMD_post_tree() | 121 |
| 5.28 amd_post_tree.c | 122 |
| 5.29 external/amd/amd_postorder.c File Reference | 123 |
| 5.29.1 Function Documentation | 123 |
| 5.29.1.1 AMD_postorder() | 124 |
| 5.30 amd_postorder.c | 124 |
| 5.31 external/amd/amd_preprocess.c File Reference | 127 |
| 5.31.1 Function Documentation | 127 |
| 5.31.1.1 AMD_preprocess() | 127 |
| 5.32 amd_preprocess.c | 127 |
| 5.33 external/amd/amd_valid.c File Reference | 129 |
| 5.33.1 Function Documentation | 129 |
| 5.33.1.1 AMD_valid() | 129 |
| 5.34 amd_valid.c | 129 |
| 5.35 external/ldl/ldl.c File Reference | 131 |
| 5.35.1 Function Documentation | 131 |
| 5.35.1.1 LDL_dsolve() | 131 |
| 5.35.1.2 LDL_ksolve() | 131 |
| 5.35.1.3 LDL_itsolve() | 132 |
| 5.35.1.4 LDL_numeric() | 132 |
| 5.35.1.5 LDL_perm() | 132 |
| 5.35.1.6 LDL_permt() | 132 |
| 5.35.1.7 LDL_symbolic() | 133 |
| 5.35.1.8 LDL_valid_matrix() | 133 |
| 5.35.1.9 LDL_valid_perm() | 133 |
| 5.36 ldl.c | 133 |
| 5.37 external/ldl/ldl.h File Reference | 141 |
| 5.37.1 Macro Definition Documentation | 142 |
| 5.37.1.1 LDL_DATE | 142 |
| 5.37.1.2 LDL_dsolve | 142 |
| 5.37.1.3 LDL_ID | 142 |
| 5.37.1.4 LDL_int | 142 |
| 5.37.1.5 LDL_ksolve | 143 |
| 5.37.1.6 LDL_itsolve | 143 |
| 5.37.1.7 LDL_MAIN_VERSION | 143 |
| 5.37.1.8 LDL_numeric | 143 |
| 5.37.1.9 LDL_perm | 143 |
| 5.37.1.10 LDL_permt | 143 |
| 5.37.1.11 LDL_SUB_VERSION | 144 |
| 5.37.1.12 LDL_SUBSUB_VERSION | 144 |
| 5.37.1.13 LDL_symbolic | 144 |
| 5.37.1.14 LDL_valid_matrix | 144 |

| | |
|---|-----|
| 5.37.1.15 LDL_valid_perm | 144 |
| 5.37.1.16 LDL_VERSION | 144 |
| 5.37.1.17 LDL_VERSION_CODE | 145 |
| 5.37.2 Function Documentation | 145 |
| 5.37.2.1 ldl_dsolve() | 145 |
| 5.37.2.2 ldl_l_dsolve() | 145 |
| 5.37.2.3 ldl_l_lsolve() | 145 |
| 5.37.2.4 ldl_l_itsolve() | 145 |
| 5.37.2.5 ldl_l_numeric() | 146 |
| 5.37.2.6 ldl_l_perm() | 146 |
| 5.37.2.7 ldl_l_permt() | 146 |
| 5.37.2.8 ldl_l_symbolic() | 146 |
| 5.37.2.9 ldl_l_valid_matrix() | 147 |
| 5.37.2.10 ldl_l_valid_perm() | 147 |
| 5.37.2.11 ldl_lsolve() | 147 |
| 5.37.2.12 ldl_itsolve() | 147 |
| 5.37.2.13 ldl_numeric() | 148 |
| 5.37.2.14 ldl_perm() | 148 |
| 5.37.2.15 ldl_permt() | 148 |
| 5.37.2.16 ldl_symbolic() | 148 |
| 5.37.2.17 ldl_valid_matrix() | 149 |
| 5.37.2.18 ldl_valid_perm() | 149 |
| 5.38 ldl.h | 149 |
| 5.39 external/README.md File Reference | 150 |
| 5.40 external/SuiteSparse_config.c File Reference | 150 |
| 5.40.1 Function Documentation | 151 |
| 5.40.1.1 SuiteSparse_calloc() | 151 |
| 5.40.1.2 SuiteSparse_divcomplex() | 151 |
| 5.40.1.3 SuiteSparse_finish() | 151 |
| 5.40.1.4 SuiteSparse_free() | 152 |
| 5.40.1.5 SuiteSparse_hypot() | 152 |
| 5.40.1.6 SuiteSparse_malloc() | 152 |
| 5.40.1.7 SuiteSparse_realloc() | 152 |
| 5.40.1.8 SuiteSparse_start() | 152 |
| 5.40.1.9 SuiteSparse_tic() | 153 |
| 5.40.1.10 SuiteSparse_time() | 153 |
| 5.40.1.11 SuiteSparse_toc() | 153 |
| 5.40.1.12 SuiteSparse_version() | 153 |
| 5.40.2 Variable Documentation | 153 |
| 5.40.2.1 SuiteSparse_config | 153 |
| 5.41 SuiteSparse_config.c | 154 |
| 5.42 external/SuiteSparse_config.h File Reference | 160 |

| | |
|---|-----|
| 5.42.1 Macro Definition Documentation | 161 |
| 5.42.1.1 SUITESPARSE_DATE | 161 |
| 5.42.1.2 SUITESPARSE_HAS_VERSION_FUNCTION | 161 |
| 5.42.1.3 SuiteSparse_long | 161 |
| 5.42.1.4 SuiteSparse_long_id | 162 |
| 5.42.1.5 SuiteSparse_long_idd | 162 |
| 5.42.1.6 SuiteSparse_long_max | 162 |
| 5.42.1.7 SUITESPARSE_MAIN_VERSION | 162 |
| 5.42.1.8 SUITESPARSE_PRINTF | 162 |
| 5.42.1.9 SUITESPARSE_SUB_VERSION | 162 |
| 5.42.1.10 SUITESPARSE_SUBSUB_VERSION | 163 |
| 5.42.1.11 SUITESPARSE_VER_CODE | 163 |
| 5.42.1.12 SUITESPARSE_VERSION | 163 |
| 5.42.2 Function Documentation | 163 |
| 5.42.2.1 SuiteSparse_calloc() | 163 |
| 5.42.2.2 SuiteSparse_divcomplex() | 163 |
| 5.42.2.3 SuiteSparse_finish() | 164 |
| 5.42.2.4 SuiteSparse_free() | 164 |
| 5.42.2.5 SuiteSparse_hypot() | 164 |
| 5.42.2.6 SuiteSparse_malloc() | 164 |
| 5.42.2.7 SuiteSparse_realloc() | 164 |
| 5.42.2.8 SuiteSparse_start() | 165 |
| 5.42.2.9 SuiteSparse_tic() | 165 |
| 5.42.2.10 SuiteSparse_time() | 165 |
| 5.42.2.11 SuiteSparse_toc() | 165 |
| 5.42.2.12 SuiteSparse_version() | 165 |
| 5.42.3 Variable Documentation | 165 |
| 5.42.3.1 SuiteSparse_config | 166 |
| 5.43 SuiteSparse_config.h | 166 |
| 5.44 include/abip.h File Reference | 169 |
| 5.44.1 Typedef Documentation | 169 |
| 5.44.1.1 ABIPAdaptWork | 170 |
| 5.44.1.2 ABIPData | 170 |
| 5.44.1.3 ABIPInfo | 170 |
| 5.44.1.4 ABIPLinSysWork | 170 |
| 5.44.1.5 ABIPMatrix | 170 |
| 5.44.1.6 ABIPResiduals | 170 |
| 5.44.1.7 ABIPScaling | 171 |
| 5.44.1.8 ABIPSettings | 171 |
| 5.44.1.9 ABIPSolution | 171 |
| 5.44.1.10 ABIPWork | 171 |
| 5.44.2 Function Documentation | 171 |

| | |
|---|-----|
| 5.44.2.1 finish() | 171 |
| 5.44.2.2 init() | 172 |
| 5.44.2.3 main() | 172 |
| 5.44.2.4 solve() | 172 |
| 5.44.2.5 version() | 172 |
| 5.45 abip.h | 173 |
| 5.46 include/abip_blas.h File Reference | 175 |
| 5.47 abip_blas.h | 175 |
| 5.48 include/adaptive.h File Reference | 176 |
| 5.48.1 Function Documentation | 176 |
| 5.48.1.1 adaptive() | 176 |
| 5.48.1.2 free_adapt() | 176 |
| 5.48.1.3 get_adapt_summary() | 177 |
| 5.48.1.4 init_adapt() | 177 |
| 5.49 adaptive.h | 177 |
| 5.50 include/cs.h File Reference | 177 |
| 5.50.1 Function Documentation | 178 |
| 5.50.1.1 ABIP() | 178 |
| 5.50.1.2 cs_compress() | 178 |
| 5.50.1.3 cs_cumsum() | 178 |
| 5.50.1.4 cs_pinv() | 179 |
| 5.50.1.5 cs_spalloc() | 179 |
| 5.50.1.6 cs_spfree() | 179 |
| 5.50.1.7 cs_symperm() | 179 |
| 5.50.1.8 cs_transpose() | 179 |
| 5.50.2 Variable Documentation | 180 |
| 5.50.2.1 cs | 180 |
| 5.51 cs.h | 180 |
| 5.52 include/ctrlc.h File Reference | 181 |
| 5.52.1 Macro Definition Documentation | 181 |
| 5.52.1.1 abip_end_interrupt_listener | 181 |
| 5.52.1.2 abip_is_interrupted | 181 |
| 5.52.1.3 abip_start_interrupt_listener | 181 |
| 5.52.2 Typedef Documentation | 182 |
| 5.52.2.1 abip_make_iso_compilers_happy | 182 |
| 5.53 ctrlc.h | 182 |
| 5.54 include/glbopts.h File Reference | 182 |
| 5.54.1 Macro Definition Documentation | 184 |
| 5.54.1.1 _abip_calloc | 184 |
| 5.54.1.2 _abip_free | 184 |
| 5.54.1.3 _abip_malloc | 184 |
| 5.54.1.4 _abip_realloc | 184 |

| | |
|--|-----|
| 5.54.1.5 ABIP | 184 |
| 5.54.1.6 abip_calloc | 185 |
| 5.54.1.7 ABIP_FAILED | 185 |
| 5.54.1.8 abip_free | 185 |
| 5.54.1.9 ABIP_INDETERMINATE | 185 |
| 5.54.1.10 ABIP_INFEASIBLE | 185 |
| 5.54.1.11 ABIP_INFEASIBLE_INACCURATE | 186 |
| 5.54.1.12 abip_malloc | 186 |
| 5.54.1.13 ABIP_NULL | 186 |
| 5.54.1.14 abip_printf | 186 |
| 5.54.1.15 abip_realloc | 186 |
| 5.54.1.16 ABIP_SIGINT | 186 |
| 5.54.1.17 ABIP_SOLVED | 187 |
| 5.54.1.18 ABIP_SOLVED_INACCURATE | 187 |
| 5.54.1.19 ABIP_UNBOUNDED | 187 |
| 5.54.1.20 ABIP_UNBOUNDED_INACCURATE | 187 |
| 5.54.1.21 ABIP_UNFINISHED | 187 |
| 5.54.1.22 ABIP_VERSION | 187 |
| 5.54.1.23 ABS | 188 |
| 5.54.1.24 ADAPTIVE | 188 |
| 5.54.1.25 ADAPTIVE_LOOKBACK | 188 |
| 5.54.1.26 ALPHA | 188 |
| 5.54.1.27 CG_RATE | 188 |
| 5.54.1.28 CONVERGED_INTERVAL | 188 |
| 5.54.1.29 DEBUG_FUNC | 189 |
| 5.54.1.30 EPS | 189 |
| 5.54.1.31 EPS_COR | 189 |
| 5.54.1.32 EPS_PEN | 189 |
| 5.54.1.33 EPS_TOL | 189 |
| 5.54.1.34 INDETERMINATE_TOL | 189 |
| 5.54.1.35 INFINITY | 190 |
| 5.54.1.36 MAX | 190 |
| 5.54.1.37 MAX_ADMM_ITERS | 190 |
| 5.54.1.38 MAX_IPM_ITERS | 190 |
| 5.54.1.39 MIN | 190 |
| 5.54.1.40 NAN | 190 |
| 5.54.1.41 NORMALIZE | 191 |
| 5.54.1.42 POWF | 191 |
| 5.54.1.43 RETURN | 191 |
| 5.54.1.44 RHO_Y | 191 |
| 5.54.1.45 SAFEDIV_POS | 191 |
| 5.54.1.46 SCALE | 191 |

| | |
|--|-----|
| 5.54.1.47 SPARSITY_RATIO | 192 |
| 5.54.1.48 SQRTF | 192 |
| 5.54.1.49 VERBOSE | 192 |
| 5.54.1.50 WARM_START | 192 |
| 5.54.2 Typedef Documentation | 192 |
| 5.54.2.1 abip_float | 192 |
| 5.54.2.2 abip_int | 192 |
| 5.55 glbopts.h | 193 |
| 5.56 include/linalg.h File Reference | 195 |
| 5.56.1 Function Documentation | 195 |
| 5.56.1.1 add_array() | 196 |
| 5.56.1.2 add_scaled_array() | 196 |
| 5.56.1.3 dot() | 196 |
| 5.56.1.4 min_abs_sqrt() | 196 |
| 5.56.1.5 norm() | 197 |
| 5.56.1.6 norm_diff() | 197 |
| 5.56.1.7 norm_inf() | 197 |
| 5.56.1.8 norm_inf_diff() | 197 |
| 5.56.1.9 norm_inf_sqrt() | 198 |
| 5.56.1.10 norm_one() | 198 |
| 5.56.1.11 norm_one_sqrt() | 198 |
| 5.56.1.12 norm_sq() | 198 |
| 5.56.1.13 scale_array() | 199 |
| 5.56.1.14 set_as_scaled_array() | 199 |
| 5.56.1.15 set_as_sq() | 199 |
| 5.56.1.16 set_as_sqrt() | 199 |
| 5.57 linalg.h | 200 |
| 5.58 include/linsys.h File Reference | 201 |
| 5.58.1 Function Documentation | 202 |
| 5.58.1.1 accum_by_A() | 202 |
| 5.58.1.2 accum_by_Atrans() | 202 |
| 5.58.1.3 copy_A_matrix() | 202 |
| 5.58.1.4 free_A_matrix() | 202 |
| 5.58.1.5 free_lin_sys_work() | 203 |
| 5.58.1.6 free_lin_sys_work_pds() | 203 |
| 5.58.1.7 get_lin_sys_method() | 203 |
| 5.58.1.8 get_lin_sys_summary() | 203 |
| 5.58.1.9 init_lin_sys_work() | 203 |
| 5.58.1.10 normalize_A() | 204 |
| 5.58.1.11 solve_lin_sys() | 204 |
| 5.58.1.12 un_normalize_A() | 204 |
| 5.58.1.13 validate_lin_sys() | 204 |

| | |
|---|-----|
| 5.59 linsys.h | 205 |
| 5.60 include/normalize.h File Reference | 206 |
| 5.60.1 Function Documentation | 206 |
| 5.60.1.1 calc_scaled_resids() | 206 |
| 5.60.1.2 normalize_b_c() | 206 |
| 5.60.1.3 normalize_warm_start() | 207 |
| 5.60.1.4 un_normalize_sol() | 207 |
| 5.61 normalize.h | 207 |
| 5.62 include/util.h File Reference | 207 |
| 5.62.1 Function Documentation | 208 |
| 5.62.1.1 ABIP() | 208 |
| 5.62.1.2 free_data() | 208 |
| 5.62.1.3 free_sol() | 209 |
| 5.62.1.4 print_array() | 209 |
| 5.62.1.5 print_data() | 209 |
| 5.62.1.6 print_work() | 209 |
| 5.62.1.7 set_default_settings() | 210 |
| 5.62.1.8 str_toc() | 210 |
| 5.62.1.9 tic() | 210 |
| 5.62.1.10 toc() | 210 |
| 5.62.1.11 tocq() | 210 |
| 5.63 util.h | 211 |
| 5.64 interface/abip_direct.m File Reference | 211 |
| 5.64.1 Function Documentation | 212 |
| 5.64.1.1 R() | 212 |
| 5.64.2 Variable Documentation | 212 |
| 5.64.2.1 A | 212 |
| 5.64.2.2 Ax | 212 |
| 5.64.2.3 b | 212 |
| 5.64.2.4 c | 213 |
| 5.64.2.5 function | 213 |
| 5.64.2.6 x | 213 |
| 5.65 abip_direct.m | 213 |
| 5.66 interface/abip_indirect.m File Reference | 214 |
| 5.66.1 Function Documentation | 214 |
| 5.66.1.1 R() | 214 |
| 5.66.2 Variable Documentation | 214 |
| 5.66.2.1 A | 214 |
| 5.66.2.2 Ax | 214 |
| 5.66.2.3 b | 215 |
| 5.66.2.4 c | 215 |
| 5.66.2.5 function | 215 |

| | |
|---|-----|
| 5.66.2.6 x | 215 |
| 5.67 abip_indirect.m | 216 |
| 5.68 linsys/abip_pardiso.c File Reference | 216 |
| 5.68.1 Function Documentation | 216 |
| 5.68.1.1 pardisoFactorize() | 217 |
| 5.68.1.2 pardisoFree() | 217 |
| 5.68.1.3 pardisoSolve() | 217 |
| 5.69 abip_pardiso.c | 217 |
| 5.70 linsys/abip_pardiso.h File Reference | 218 |
| 5.70.1 Macro Definition Documentation | 219 |
| 5.70.1.1 FACTORIZE | 219 |
| 5.70.1.2 PARDISO_BACKWARD | 219 |
| 5.70.1.3 PARDISO_FAC | 219 |
| 5.70.1.4 PARDISO_FORWARD | 219 |
| 5.70.1.5 PARDISO_FREE | 220 |
| 5.70.1.6 PARDISO_OK | 220 |
| 5.70.1.7 PARDISO_SOLVE | 220 |
| 5.70.1.8 PARDISO_SYM | 220 |
| 5.70.1.9 PARDISO_SYM_FAC | 220 |
| 5.70.1.10 PARDISOINDEX | 220 |
| 5.70.1.11 PIVOTING | 221 |
| 5.70.1.12 SYMBOLIC | 221 |
| 5.70.2 Function Documentation | 221 |
| 5.70.2.1 pardiso() | 221 |
| 5.70.2.2 pardisoFactorize() | 221 |
| 5.70.2.3 pardisoFree() | 222 |
| 5.70.2.4 pardisoinit() | 222 |
| 5.70.2.5 pardisoSolve() | 222 |
| 5.71 abip_pardiso.h | 222 |
| 5.72 linsys/amatrix.h File Reference | 223 |
| 5.73 amatrix.h | 223 |
| 5.74 linsys/common.c File Reference | 224 |
| 5.74.1 Macro Definition Documentation | 224 |
| 5.74.1.1 MAX_SCALE | 225 |
| 5.74.1.2 MIN_SCALE | 225 |
| 5.74.2 Function Documentation | 225 |
| 5.74.2.1 _accum_by_A() | 225 |
| 5.74.2.2 _accum_by_Atrans() | 225 |
| 5.74.2.3 _normalize_A() | 226 |
| 5.74.2.4 _un_normalize_A() | 226 |
| 5.74.2.5 copy_A_matrix() | 226 |
| 5.74.2.6 cumsum() | 226 |

| | |
|---------------------------------------|-----|
| 5.74.2.7 free_A_matrix() | 227 |
| 5.74.2.8 validate_lin_sys() | 227 |
| 5.75 common.c | 227 |
| 5.76 linsys/common.h File Reference | 235 |
| 5.76.1 Function Documentation | 236 |
| 5.76.1.1 _accum_by_A() | 236 |
| 5.76.1.2 _accum_by_Atrans() | 236 |
| 5.76.1.3 _normalize_A() | 236 |
| 5.76.1.4 _un_normalize_A() | 237 |
| 5.76.1.5 cumsum() | 237 |
| 5.77 common.h | 237 |
| 5.78 linsys/direct.c File Reference | 238 |
| 5.78.1 Function Documentation | 238 |
| 5.78.1.1 _ldl_factor() | 238 |
| 5.78.1.2 _ldl_init() | 239 |
| 5.78.1.3 _ldl_solve() | 239 |
| 5.78.1.4 accum_by_A() | 239 |
| 5.78.1.5 accum_by_Atrans() | 239 |
| 5.78.1.6 factorize() | 240 |
| 5.78.1.7 form_kkt() | 240 |
| 5.78.1.8 free_lin_sys_work() | 240 |
| 5.78.1.9 get_lin_sys_method() | 240 |
| 5.78.1.10 get_lin_sys_summary() | 240 |
| 5.78.1.11 init_lin_sys_work() | 241 |
| 5.78.1.12 normalize_A() | 241 |
| 5.78.1.13 solve_lin_sys() | 241 |
| 5.78.1.14 un_normalize_A() | 241 |
| 5.79 direct.c | 242 |
| 5.80 linsys/direct.h File Reference | 245 |
| 5.81 direct.h | 246 |
| 5.82 linsys/indirect.c File Reference | 246 |
| 5.82.1 Macro Definition Documentation | 247 |
| 5.82.1.1 CG_BEST_TOL | 247 |
| 5.82.1.2 CG_MIN_TOL | 247 |
| 5.82.2 Function Documentation | 247 |
| 5.82.2.1 accum_by_A() | 247 |
| 5.82.2.2 accum_by_Atrans() | 248 |
| 5.82.2.3 free_lin_sys_work() | 248 |
| 5.82.2.4 get_lin_sys_method() | 248 |
| 5.82.2.5 get_lin_sys_summary() | 248 |
| 5.82.2.6 init_lin_sys_work() | 248 |
| 5.82.2.7 normalize_A() | 249 |

| | |
|--|-----|
| 5.82.2.8 solve_lin_sys() | 249 |
| 5.82.2.9 un_normalize_A() | 249 |
| 5.83 indirect.c | 249 |
| 5.84 linsys/indirect.h File Reference | 254 |
| 5.85 indirect.h | 255 |
| 5.86 make_abip.m File Reference | 255 |
| 5.86.1 Typedef Documentation | 256 |
| 5.86.1.1 int | 256 |
| 5.86.2 Function Documentation | 256 |
| 5.86.2.1 addpath() [1/3] | 256 |
| 5.86.2.2 addpath() [2/3] | 256 |
| 5.86.2.3 addpath() [3/3] | 257 |
| 5.86.2.4 compile_direct() | 257 |
| 5.86.2.5 compile_indirect() | 257 |
| 5.86.2.6 delete() | 257 |
| 5.86.2.7 elseif() | 257 |
| 5.86.2.8 if() [1/5] | 257 |
| 5.86.2.9 if() [2/5] | 257 |
| 5.86.2.10 if() [3/5] | 258 |
| 5.86.2.11 if() [4/5] | 258 |
| 5.86.2.12 if() [5/5] | 258 |
| 5.86.2.13 movefile() | 258 |
| 5.86.3 Variable Documentation | 258 |
| 5.86.3.1 abip_common_linsys | 258 |
| 5.86.3.2 abip_common_src | 258 |
| 5.86.3.3 abip_mexfile | 259 |
| 5.86.3.4 arr | 259 |
| 5.86.3.5 BLASLIB | 259 |
| 5.86.3.6 c | 259 |
| 5.86.3.7 common_abip | 259 |
| 5.86.3.8 EXTRA_VERBOSE | 259 |
| 5.86.3.9 float | 260 |
| 5.86.3.10 gpu | 260 |
| 5.86.3.11 INCS | 260 |
| 5.86.3.12 INT | 260 |
| 5.86.3.13 LCFLAG | 260 |
| 5.86.3.14 link | 260 |
| 5.86.3.15 LOCS | 261 |
| 5.86.3.16 WARNING | 261 |
| 5.87 make_abip.m | 261 |
| 5.88 mexfile/abip_mex.c File Reference | 262 |
| 5.88.1 Function Documentation | 262 |

| | |
|--|-----|
| 5.88.1.1 cast_to_abip_int_arr() | 262 |
| 5.88.1.2 free_mex() | 263 |
| 5.88.1.3 mexFunction() | 263 |
| 5.88.1.4 parse_warm_start() | 263 |
| 5.88.1.5 set_output_field() | 263 |
| 5.89 abip_mex.c | 264 |
| 5.90 mexfile/abip_version_mex.c File Reference | 269 |
| 5.90.1 Function Documentation | 269 |
| 5.90.1.1 mexFunction() | 269 |
| 5.91 abip_version_mex.c | 270 |
| 5.92 src/abip.c File Reference | 270 |
| 5.92.1 Function Documentation | 270 |
| 5.92.1.1 ABIP() | 270 |
| 5.92.1.2 finish() | 271 |
| 5.92.1.3 init() | 271 |
| 5.92.1.4 main() | 271 |
| 5.92.1.5 solve() | 271 |
| 5.93 abip.c | 272 |
| 5.94 src/abip_version.c File Reference | 298 |
| 5.94.1 Function Documentation | 298 |
| 5.94.1.1 version() | 299 |
| 5.95 abip_version.c | 299 |
| 5.96 src/adaptive.c File Reference | 299 |
| 5.96.1 Function Documentation | 299 |
| 5.96.1.1 adaptive() | 299 |
| 5.96.1.2 free_adapt() | 300 |
| 5.96.1.3 get_adapt_summary() | 300 |
| 5.96.1.4 init_adapt() | 300 |
| 5.97 adaptive.c | 300 |
| 5.98 src/cs.c File Reference | 305 |
| 5.98.1 Function Documentation | 305 |
| 5.98.1.1 cs_compress() | 306 |
| 5.98.1.2 cs_cumsum() | 306 |
| 5.98.1.3 cs_pinv() | 306 |
| 5.98.1.4 cs_spalloc() | 306 |
| 5.98.1.5 cs_spfree() | 306 |
| 5.98.1.6 cs_symperm() | 307 |
| 5.98.1.7 cs_transpose() | 307 |
| 5.99 cs.c | 307 |
| 5.100 src/ctrlc.c File Reference | 311 |
| 5.101 ctrlc.c | 311 |
| 5.102 src/linalg.c File Reference | 312 |

| | |
|--|-----|
| 5.102.1 Function Documentation | 313 |
| 5.102.1.1 add_array() | 313 |
| 5.102.1.2 add_scaled_array() | 313 |
| 5.102.1.3 dot() | 313 |
| 5.102.1.4 min_abs_sqrt() | 314 |
| 5.102.1.5 norm() | 314 |
| 5.102.1.6 norm_diff() | 314 |
| 5.102.1.7 norm_inf() | 314 |
| 5.102.1.8 norm_inf_diff() | 315 |
| 5.102.1.9 norm_inf_sqrt() | 315 |
| 5.102.1.10 norm_one() | 315 |
| 5.102.1.11 norm_one_sqrt() | 315 |
| 5.102.1.12 norm_sq() | 316 |
| 5.102.1.13 scale_array() | 316 |
| 5.102.1.14 set_as_scaled_array() | 316 |
| 5.102.1.15 set_as_sq() | 316 |
| 5.102.1.16 set_as_sqrt() | 317 |
| 5.103 linalg.c | 317 |
| 5.104 src/normalize.c File Reference | 320 |
| 5.104.1 Macro Definition Documentation | 320 |
| 5.104.1.1 MAX_SCALE | 320 |
| 5.104.1.2 MIN_SCALE | 320 |
| 5.104.2 Function Documentation | 321 |
| 5.104.2.1 calc_scaled_resids() | 321 |
| 5.104.2.2 normalize_b_c() | 321 |
| 5.104.2.3 normalize_warm_start() | 321 |
| 5.104.2.4 un_normalize_sol() | 321 |
| 5.105 normalize.c | 322 |
| 5.106 src/util.c File Reference | 323 |
| 5.106.1 Function Documentation | 324 |
| 5.106.1.1 free_data() | 324 |
| 5.106.1.2 free_sol() | 324 |
| 5.106.1.3 print_array() | 325 |
| 5.106.1.4 print_data() | 325 |
| 5.106.1.5 print_work() | 325 |
| 5.106.1.6 set_default_settings() | 325 |
| 5.106.1.7 str_toc() | 326 |
| 5.106.1.8 tic() | 326 |
| 5.106.1.9 toc() | 326 |
| 5.106.1.10 tocq() | 326 |
| 5.107 util.c | 327 |

Chapter 1

README

This AMD and LDL code are from the official SuiteSparse library: a suite of sparse matrix algorithms authored or co-authored by Tim Davis, Texas A&M University. You can visit this repo by: <https://github.com/DrTimothyAldenDavis/SuiteSparse/tree/dev>. We include each license in the subfolder, respectively.

Chapter 2

Data Structure Index

2.1 Data Structures

Here are the data structures with brief descriptions:

| | |
|---|----|
| ABIP_A_DATA_MATRIX | 7 |
| ABIP_ADAPTIVE_WORK | 8 |
| ABIP_INFO | 11 |
| ABIP_LIN_SYS_WORK | 14 |
| ABIP_PROBLEM_DATA | 17 |
| ABIP_RESIDUALS | 19 |
| ABIP_SCALING | 22 |
| ABIP_SETTINGS | 23 |
| ABIP_SOL_VARS | 29 |
| ABIP_WORK | 30 |
| SuiteSparse_config_struct | 37 |

Chapter 3

File Index

3.1 File List

Here is a list of all files with brief descriptions:

| | |
|---|-----|
| compile_direct.m | 41 |
| compile_indirect.m | 48 |
| make_abip.m | 255 |
| external/SuiteSparse_config.c | 150 |
| external/SuiteSparse_config.h | 160 |
| external/amd/amd.h | 49 |
| external/amd/amd_1.c | 64 |
| external/amd/amd_2.c | 67 |
| external/amd/amd_aat.c | 91 |
| external/amd/amd_control.c | 94 |
| external/amd/amd_defaults.c | 95 |
| external/amd/amd_dump.c | 96 |
| external/amd/amd_global.c | 100 |
| external/amd/amd_info.c | 103 |
| external/amd/amd_internal.h | 105 |
| external/amd/amd_order.c | 118 |
| external/amd/amd_post_tree.c | 121 |
| external/amd/amd_postorder.c | 123 |
| external/amd/amd_preprocess.c | 127 |
| external/amd/amd_valid.c | 129 |
| external/ldl/ldl.c | 131 |
| external/ldl/ldl.h | 141 |
| include/abip.h | 169 |
| include/abip_blas.h | 175 |
| include/adaptive.h | 176 |
| include/cs.h | 177 |
| include/ctrlc.h | 181 |
| include/glbopts.h | 182 |
| include/linalg.h | 195 |
| include/linsys.h | 201 |
| include/normalize.h | 206 |
| include/util.h | 207 |
| interface/abip_direct.m | 211 |
| interface/abip_indirect.m | 214 |
| linsys/abip_pardiso.c | 216 |

| | |
|--|-----|
| linsys/abip_pardiso.h | 218 |
| linsys/amatrix.h | 223 |
| linsys/common.c | 224 |
| linsys/common.h | 235 |
| linsys/direct.c | 238 |
| linsys/direct.h | 245 |
| linsys/indirect.c | 246 |
| linsys/indirect.h | 254 |
| mexfile/abip_mex.c | 262 |
| mexfile/abip_version_mex.c | 269 |
| src/abip.c | 270 |
| src/abip_version.c | 298 |
| src/adaptive.c | 299 |
| src/cs.c | 305 |
| src/ctrlc.c | 311 |
| src/linalg.c | 312 |
| src/normalize.c | 320 |
| src/util.c | 323 |

Chapter 4

Data Structure Documentation

4.1 ABIP_A_DATA_MATRIX Struct Reference

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

Data Fields

- [abip_float](#) * x
- [abip_int](#) * i
- [abip_int](#) * p
- [abip_int](#) m
- [abip_int](#) n

4.1.1 Detailed Description

Definition at line 10 of file [amatrix.h](#).

4.1.2 Field Documentation

4.1.2.1 i

```
abip\_int* i
```

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

4.1.2.2 m

`abip_int` m

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

4.1.2.3 n

`abip_int` n

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

4.1.2.4 p

`abip_int*` p

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

4.1.2.5 x

`abip_float*` x

Definition at line 12 of file [amatrix.h](#).

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

- [linsys/amatrix.h](#)

4.2 ABIP_ADAPTIVE_WORK Struct Reference

Data Fields

- `abip_float` * u_prev
- `abip_float` * v_prev
- `abip_float` * ut
- `abip_float` * u
- `abip_float` * v
- `abip_float` * ut_next
- `abip_float` * u_next
- `abip_float` * v_next
- `abip_float` * delta_ut
- `abip_float` * delta_u
- `abip_float` * delta_v
- `abip_int` l
- `abip_int` k
- `abip_float` total_adapt_time

4.2.1 Detailed Description

Definition at line 13 of file [adaptive.c](#).

4.2.2 Field Documentation

4.2.2.1 delta_u

`abip_float* delta_u`

Definition at line 25 of file [adaptive.c](#).

4.2.2.2 delta_ut

`abip_float* delta_ut`

Definition at line 24 of file [adaptive.c](#).

4.2.2.3 delta_v

`abip_float* delta_v`

Definition at line 26 of file [adaptive.c](#).

4.2.2.4 k

`abip_int k`

Definition at line 29 of file [adaptive.c](#).

4.2.2.5 l

`abip_int l`

Definition at line 28 of file [adaptive.c](#).

4.2.2.6 total_adapt_time

`abip_float total_adapt_time`

Definition at line 31 of file [adaptive.c](#).

4.2.2.7 u

`abip_float* u`

Definition at line 18 of file [adaptive.c](#).

4.2.2.8 u_next

`abip_float* u_next`

Definition at line 21 of file [adaptive.c](#).

4.2.2.9 u_prev

`abip_float* u_prev`

Definition at line 15 of file [adaptive.c](#).

4.2.2.10 ut

`abip_float* ut`

Definition at line 17 of file [adaptive.c](#).

4.2.2.11 ut_next

`abip_float* ut_next`

Definition at line 20 of file [adaptive.c](#).

4.2.2.12 v

`abip_float* v`

Definition at line 19 of file [adaptive.c](#).

4.2.2.13 v_next

`abip_float* v_next`

Definition at line 22 of file [adaptive.c](#).

4.2.2.14 v_prev

`abip_float* v_prev`

Definition at line 16 of file [adaptive.c](#).

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

- [src/adaptive.c](#)

4.3 ABIP_INFO Struct Reference

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

Data Fields

- `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`

4.3.1 Detailed Description

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

4.3.2 Field Documentation

4.3.2.1 admm_iter

`abip_int` `admm_iter`

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

4.3.2.2 dobj

`abip_float` `dobj`

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

4.3.2.3 ipm_iter

`abip_int` `ipm_iter`

Definition at line 92 of file [abip.h](#).

4.3.2.4 pobj

`abip_float` `pobj`

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

4.3.2.5 rel_gap

`abip_float` `rel_gap`

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

4.3.2.6 res_dual

`abip_float res_dual`

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

4.3.2.7 res_infeas

`abip_float res_infeas`

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

4.3.2.8 res_pri

`abip_float res_pri`

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

4.3.2.9 res_unbdd

`abip_float res_unbdd`

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

4.3.2.10 setup_time

`abip_float setup_time`

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

4.3.2.11 solve_time

`abip_float solve_time`

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

4.3.2.12 status

```
char status[32]
```

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

4.3.2.13 status_val

```
abip_int status_val
```

Definition at line 91 of file [abip.h](#).

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

- [include/abip.h](#)

4.4 ABIP_LIN_SYS_WORK Struct Reference

```
#include <direct.h>
```

Data Fields

- [cs](#) * [L](#)
- [abip_float](#) * [D](#)
- [abip_int](#) * [P](#)
- [abip_int](#) * [i](#)
- [abip_int](#) * [j](#)
- [abip_float](#) * [bp](#)
- void * [pardiso_work](#) [[PARDISOINDEX](#)]
- [abip_float](#) [total_solve_time](#)
- [abip_float](#) * [p](#)
- [abip_float](#) * [r](#)
- [abip_float](#) * [Gp](#)
- [abip_float](#) * [tmp](#)
- [ABIPMatrix](#) * [At](#)
- [abip_float](#) * [z](#)
- [abip_float](#) * [M](#)
- [abip_int](#) [tot_cg_its](#)

4.4.1 Detailed Description

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

4.4.2 Field Documentation

4.4.2.1 At

`ABIPMatrix*` At

Definition at line 20 of file [indirect.h](#).

4.4.2.2 bp

`abip_float*` bp

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

4.4.2.3 D

`abip_float*` D

Definition at line 19 of file [direct.h](#).

4.4.2.4 Gp

`abip_float*` Gp

Definition at line 18 of file [indirect.h](#).

4.4.2.5 i

`abip_int*` i

Definition at line 21 of file [direct.h](#).

4.4.2.6 j

`abip_int*` j

Definition at line 22 of file [direct.h](#).

4.4.2.7 L

`cs* L`

Definition at line 18 of file [direct.h](#).

4.4.2.8 M

`abip_float* M`

Definition at line 24 of file [indirect.h](#).

4.4.2.9 P

`abip_int* P`

Definition at line 20 of file [direct.h](#).

4.4.2.10 p

`abip_float* p`

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

4.4.2.11 pardiso_work

`void* pardiso_work[PARDISOINDEX]`

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

4.4.2.12 r

`abip_float* r`

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

4.4.2.13 tmp

```
abip_float* tmp
```

Definition at line 19 of file [indirect.h](#).

4.4.2.14 tot_cg_its

```
abip_int tot_cg_its
```

Definition at line 27 of file [indirect.h](#).

4.4.2.15 total_solve_time

```
abip_float total_solve_time
```

Definition at line 26 of file [direct.h](#).

4.4.2.16 z

```
abip_float* z
```

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

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

- [linsys/direct.h](#)
- [linsys/indirect.h](#)

4.5 ABIP_PROBLEM_DATA Struct Reference

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

Data Fields

- [abip_int m](#)
- [abip_int n](#)
- [ABIPMatrix * A](#)
- [abip_float * b](#)
- [abip_float * c](#)
- [abip_float sp](#)
- [ABIPSettings * stgs](#)

4.5.1 Detailed Description

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

4.5.2 Field Documentation

4.5.2.1 A

`ABIPMatrix*` A

Definition at line 27 of file [abip.h](#).

4.5.2.2 b

`abip_float*` b

Definition at line 29 of file [abip.h](#).

4.5.2.3 c

`abip_float*` c

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

4.5.2.4 m

`abip_int` m

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

4.5.2.5 n

`abip_int` n

Definition at line 26 of file [abip.h](#).

4.5.2.6 sp

`abip_float` sp

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

4.5.2.7 stgs

`ABIPSettings*` stgs

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

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

- [include/abip.h](#)

4.6 ABIP_RESIDUALS Struct Reference

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

Data Fields

- [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](#) tau
- [abip_float](#) kap

4.6.1 Detailed Description

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

4.6.2 Field Documentation

4.6.2.1 `bt_y_by_tau`

`abip_float` `bt_y_by_tau`

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

4.6.2.2 `ct_x_by_tau`

`abip_float` `ct_x_by_tau`

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

4.6.2.3 `kap`

`abip_float` `kap`

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

4.6.2.4 `last_admm_iter`

`abip_int` `last_admm_iter`

Definition at line 181 of file [abip.h](#).

4.6.2.5 `last_ipm_iter`

`abip_int` `last_ipm_iter`

Definition at line 180 of file [abip.h](#).

4.6.2.6 `last_mu`

`abip_float` `last_mu`

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

4.6.2.7 rel_gap

`abip_float` rel_gap

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

4.6.2.8 res_dual

`abip_float` res_dual

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

4.6.2.9 res_infeas

`abip_float` res_infeas

Definition at line 187 of file [abip.h](#).

4.6.2.10 res_pri

`abip_float` res_pri

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

4.6.2.11 res_unbdd

`abip_float` res_unbdd

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

4.6.2.12 tau

`abip_float` tau

Definition at line 193 of file [abip.h](#).

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

- [include/abip.h](#)

4.7 ABIP_SCALING Struct Reference

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

Data Fields

- [abip_float](#) * [D](#)
- [abip_float](#) * [E](#)
- [abip_float](#) [mean_norm_row_A](#)
- [abip_float](#) [mean_norm_col_A](#)

4.7.1 Detailed Description

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

4.7.2 Field Documentation

4.7.2.1 D

[abip_float](#)* [D](#)

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

4.7.2.2 E

[abip_float](#)* [E](#)

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

4.7.2.3 mean_norm_col_A

[abip_float](#) [mean_norm_col_A](#)

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

4.7.2.4 mean_norm_row_A

`abip_float mean_norm_row_A`

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

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

- [include/abip.h](#)

4.8 ABIP_SETTINGS Struct Reference

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

Data Fields

- [abip_int normalize](#)
- [abip_int pfeasopt](#)
- [abip_float scale](#)
- [abip_float rho_y](#)
- [abip_float sparsity_ratio](#)
- [abip_int max_ipm_iters](#)
- [abip_int max_admm_iters](#)
- [abip_float max_time](#)
- [abip_float eps](#)
- [abip_float alpha](#)
- [abip_float cg_rate](#)
- [abip_int adaptive](#)
- [abip_float eps_cor](#)
- [abip_float eps_pen](#)
- [abip_float dynamic_sigma](#)
- [abip_float dynamic_x](#)
- [abip_float dynamic_eta](#)
- [abip_int restart_fre](#)
- [abip_int restart_thresh](#)
- [abip_int verbose](#)
- [abip_int warm_start](#)
- [abip_int adaptive_lookback](#)
- [abip_int origin_rescale](#)
- [abip_int pc_ruiz_rescale](#)
- [abip_int qp_rescale](#)
- [abip_int ruiz_iter](#)
- [abip_int hybrid_mu](#)
- [abip_float hybrid_thresh](#)
- [abip_float dynamic_sigma_second](#)
- [abip_int half_update](#)
- [abip_int avg_criterion](#)

4.8.1 Detailed Description

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

4.8.2 Field Documentation

4.8.2.1 adaptive

`abip_int` adaptive

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

4.8.2.2 adaptive_lookback

`abip_int` adaptive_lookback

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

4.8.2.3 alpha

`abip_float` alpha

Definition at line 49 of file [abip.h](#).

4.8.2.4 avg_criterion

`abip_int` avg_criterion

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

4.8.2.5 cg_rate

`abip_float` cg_rate

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

4.8.2.6 dynamic_eta

`abip_float` dynamic_eta

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

4.8.2.7 dynamic_sigma

`abip_float` dynamic_sigma

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

4.8.2.8 dynamic_sigma_second

`abip_float` dynamic_sigma_second

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

4.8.2.9 dynamic_x

`abip_float` dynamic_x

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

4.8.2.10 eps

`abip_float` eps

Definition at line 48 of file [abip.h](#).

4.8.2.11 eps_cor

`abip_float` eps_cor

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

4.8.2.12 `eps_pen`

`abip_float` `eps_pen`

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

4.8.2.13 `half_update`

`abip_int` `half_update`

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

4.8.2.14 `hybrid_mu`

`abip_int` `hybrid_mu`

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

4.8.2.15 `hybrid_thresh`

`abip_float` `hybrid_thresh`

Definition at line 75 of file [abip.h](#).

4.8.2.16 `max_admm_iters`

`abip_int` `max_admm_iters`

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

4.8.2.17 `max_ipm_iters`

`abip_int` `max_ipm_iters`

Definition at line 44 of file [abip.h](#).

4.8.2.18 max_time

`abip_float` max_time

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

4.8.2.19 normalize

`abip_int` normalize

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

4.8.2.20 origin_rescale

`abip_int` origin_rescale

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

4.8.2.21 pc_ruiz_rescale

`abip_int` pc_ruiz_rescale

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

4.8.2.22 pfeasopt

`abip_int` pfeasopt

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

4.8.2.23 qp_rescale

`abip_int` qp_rescale

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

4.8.2.24 restart_fre

`abip_int restart_fre`

Definition at line 61 of file [abip.h](#).

4.8.2.25 restart_thresh

`abip_int restart_thresh`

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

4.8.2.26 rho_y

`abip_float rho_y`

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

4.8.2.27 ruiz_iter

`abip_int ruiz_iter`

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

4.8.2.28 scale

`abip_float scale`

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

4.8.2.29 sparsity_ratio

`abip_float sparsity_ratio`

Definition at line 42 of file [abip.h](#).

4.8.2.30 verbose

`abip_int` verbose

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

4.8.2.31 warm_start

`abip_int` warm_start

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

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

- [include/abip.h](#)

4.9 ABIP_SOL_VARS Struct Reference

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

Data Fields

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

4.9.1 Detailed Description

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

4.9.2 Field Documentation

4.9.2.1 s

`abip_float`* s

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

4.9.2.2 x

`abip_float* x`

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

4.9.2.3 y

`abip_float* y`

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

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

- [include/abip.h](#)

4.10 ABIP_WORK Struct Reference

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

Data Fields

- [abip_float sigma](#)
- [abip_float gamma](#)
- [abip_int final_check](#)
- [abip_int double_check](#)
- [abip_float mu](#)
- [abip_float beta](#)
- [abip_float * u](#)
- [abip_float * v](#)
- [abip_float * u_t](#)
- [abip_float * u_prev](#)
- [abip_float * v_prev](#)
- [abip_float * u_avg](#)
- [abip_float * v_avg](#)
- [abip_float * u_avgcon](#)
- [abip_float * v_avgcon](#)
- [abip_float * u_sumcon](#)
- [abip_float * v_sumcon](#)
- [abip_int fre_old](#)
- [abip_float * h](#)
- [abip_float * g](#)
- [abip_float * pr](#)
- [abip_float * dr](#)
- [abip_float g_th](#)
- [abip_float sc_b](#)
- [abip_float sc_c](#)
- [abip_float nm_b](#)

- [abip_float nm_c](#)
- [abip_float * b](#)
- [abip_float * c](#)
- [abip_int m](#)
- [abip_int n](#)
- [ABIPMatrix * A](#)
- [abip_float sp](#)
- [ABIPLinSysWork * p](#)
- [ABIPAdaptWork * adapt](#)
- [ABIPSettings * stgs](#)
- [ABIPScaling * scal](#)

4.10.1 Detailed Description

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

4.10.2 Field Documentation

4.10.2.1 A

[ABIPMatrix*](#) A

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

4.10.2.2 adapt

[ABIPAdaptWork*](#) adapt

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

4.10.2.3 b

[abip_float*](#) b

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

4.10.2.4 **beta**

`abip_float` beta

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

4.10.2.5 **c**

`abip_float*` c

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

4.10.2.6 **double_check**

`abip_int` double_check

Definition at line 131 of file [abip.h](#).

4.10.2.7 **dr**

`abip_float*` dr

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

4.10.2.8 **final_check**

`abip_int` final_check

Definition at line 130 of file [abip.h](#).

4.10.2.9 **fre_old**

`abip_int` fre_old

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

4.10.2.10 g

`abip_float* g`

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

4.10.2.11 g_th

`abip_float g_th`

Definition at line 159 of file [abip.h](#).

4.10.2.12 gamma

`abip_float gamma`

Definition at line 129 of file [abip.h](#).

4.10.2.13 h

`abip_float* h`

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

4.10.2.14 m

`abip_int m`

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

4.10.2.15 mu

`abip_float mu`

Definition at line 133 of file [abip.h](#).

4.10.2.16 n

`abip_int` n

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

4.10.2.17 nm_b

`abip_float` nm_b

Definition at line 162 of file [abip.h](#).

4.10.2.18 nm_c

`abip_float` nm_c

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

4.10.2.19 p

`ABIPLinSysWork*` p

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

4.10.2.20 pr

`abip_float*` pr

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

4.10.2.21 sc_b

`abip_float` sc_b

Definition at line 160 of file [abip.h](#).

4.10.2.22 **sc_c**

`abip_float` `sc_c`

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

4.10.2.23 **scal**

`ABIPScaling*` `scal`

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

4.10.2.24 **sigma**

`abip_float` `sigma`

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

4.10.2.25 **sp**

`abip_float` `sp`

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

4.10.2.26 **stgs**

`ABIPSettings*` `stgs`

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

4.10.2.27 **u**

`abip_float*` `u`

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

4.10.2.28 u_avg

`abip_float* u_avg`

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

4.10.2.29 u_avgcon

`abip_float* u_avgcon`

Definition at line 145 of file [abip.h](#).

4.10.2.30 u_prev

`abip_float* u_prev`

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

4.10.2.31 u_sumcon

`abip_float* u_sumcon`

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

4.10.2.32 u_t

`abip_float* u_t`

Definition at line 138 of file [abip.h](#).

4.10.2.33 v

`abip_float* v`

Definition at line 137 of file [abip.h](#).

4.10.2.34 v_avg

`abip_float* v_avg`

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

4.10.2.35 v_avgcon

`abip_float* v_avgcon`

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

4.10.2.36 v_prev

`abip_float* v_prev`

Definition at line 140 of file [abip.h](#).

4.10.2.37 v_sumcon

`abip_float* v_sumcon`

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

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

- [include/abip.h](#)

4.11 SuiteSparse_config_struct Struct Reference

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

Data Fields

- `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 *)`

4.11.1 Detailed Description

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

4.11.2 Field Documentation

4.11.2.1 calloc_func

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

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

4.11.2.2 divcomplex_func

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

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

4.11.2.3 free_func

```
void (* free_func) (void *)
```

Definition at line 92 of file [SuiteSparse_config.h](#).

4.11.2.4 hypot_func

```
abip_float (* hypot_func) (abip_float, abip_float)
```

Definition at line 94 of file [SuiteSparse_config.h](#).

4.11.2.5 malloc_func

```
void *(* malloc_func) (size_t)
```

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

4.11.2.6 printf_func

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

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

4.11.2.7 realloc_func

```
void *(* 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:

- [external/SuiteSparse_config.h](#)

Chapter 5

File Documentation

5.1 compile_direct.m File Reference

Functions

- `function compile_direct (flags, common_abip) abip_include`
- `if exist (lib_path) % % platform`
- `fprintf ("Unknown platform. Not using MKL \n")`
- `end End if if (flags.COMPILE_WITH_OPENMP) cmd = sprintf('-DEXTRA_VERBOSE %s', flags.LCFLAG)`
- `eval (cmd)`

Variables

- `platform = convertCharsToStrings(computer('arch'))`
- `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`
- 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 then you uncomment the `following`
- `end mkl_macro = "-DABIP_PARDISO"`
- `pardiso_src = fullfile("linsys", "abip_pardiso.c")`
- `flags link`
- `if mexext`
- `else cmd = sprintf ('mex -O %s %s %s %s %s %s %s %s', mkl_macro, flags.arr, lib_path, flags.LCFLAG, flags.INCS, abip_include, flags.INT)`
- `end ldl_path = fullfile("external", "ldl")`
- `amd_path = fullfile("external", "amd")`
- `ldl_files = ["ldl.c"]`
- `amd_files`
- `abip_ldl = fullfile(ldl_path, ldl_files)`
- `abip_amd = fullfile(amd_path, amd_files)`

5.1.1 Function Documentation

5.1.1.1 compile_direct()

```
function compile_direct (
    flags ,
    common_abip )
```

5.1.1.2 eval()

```
eval (
    cmd )
```

5.1.1.3 exist()

```
if exist (
    lib_path )
```

5.1.1.4 fprintf()

```
fprintf (
    "Unknown platform. Not using MKL \n" )
```

5.1.1.5 if()

```
end if (
    flags. COMPILER_WITH_OPENMP ) = sprintf('-DEXTRA_VERBOSE %s', flags.LCFLAG)
```

5.1.2 Variable Documentation

5.1.2.1 abip_amd

```
abip_amd = fullfile(amd_path, amd_files)
```

Definition at line 73 of file [compile_direct.m](#).

5.1.2.2 abip_ldl

```
abip_ldl = fullfile(ldl_path, ldl_files)
```

Definition at line 71 of file [compile_direct.m](#).

5.1.2.3 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 14 of file [compile_direct.m](#).

5.1.2.4 amd_files

```
amd_files
```

Initial value:

```
= ["amd_order", "amd_dump", "amd_postorder", "amd_post_tree", ...  
   "amd_aat", "amd_2", "amd_1", "amd_defaults", "amd_control", ...  
   "amd_info", "amd_valid", "amd_global", "amd_preprocess"]
```

Definition at line 66 of file [compile_direct.m](#).

5.1.2.5 amd_path

```
amd_path = fullfile("external", "amd")
```

Definition at line 63 of file [compile_direct.m](#).

5.1.2.6 cmd

```
cmd = sprintf ('mex -O %s %s %s %s %s %s %s %s', mkl_macro, flags.arr, lib_path, flags.LCFLAG, flags.INCS, abip_include, flags.INT)
```

Definition at line 59 of file [compile_direct.m](#).

5.1.2.7 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 15 of file [compile_direct.m](#).

5.1.2.8 following

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 then you uncomment the following

Definition at line 16 of file [compile_direct.m](#).

5.1.2.9 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 15 of file [compile_direct.m](#).

5.1.2.10 ldl_files

```
ldl_files = ["ldl.c"]
```

Definition at line 65 of file [compile_direct.m](#).

5.1.2.11 `ldl_path`

```
end ldl_path = fullfile("external", "ldl")
```

Definition at line 62 of file [compile_direct.m](#).

5.1.2.12 `lib_path`

```
else lib_path = ""
```

Definition at line 10 of file [compile_direct.m](#).

5.1.2.13 `link`

```
else flags link
```

Initial value:

```
= [flags.link, ' -lmkl_intel_ilp64',...  
    ' -lmkl_core', ' -lmkl_sequential ']
```

Definition at line 34 of file [compile_direct.m](#).

5.1.2.14 `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 15 of file [compile_direct.m](#).

5.1.2.15 `mexext`

```
if mexext
```

Initial value:

```
== "mexw64"  
    flags.link = '-lut -lmwblas -lmwlapack'
```

Definition at line 47 of file [compile_direct.m](#).

5.1.2.16 MKL

If use MKL

Definition at line 12 of file [compile_direct.m](#).

5.1.2.17 mkl_macro

```
mkl_macro = "-DABIP_PARDISO"
```

Definition at line 29 of file [compile_direct.m](#).

5.1.2.18 oneapi

If use we suggest you set the environmental variables by oneapi

Definition at line 12 of file [compile_direct.m](#).

5.1.2.19 pardiso_src

```
end End if pardiso_src = fullfile("linsys", "abip_pardiso.c")
```

Definition at line 30 of file [compile_direct.m](#).

5.1.2.20 platform

```
elseif platform = convertCharsToStrings(computer('arch'))
```

Definition at line 9 of file [compile_direct.m](#).

5.1.2.21 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 14 of file [compile_direct.m](#).

5.2 compile_direct.m

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

```

00001 function compile_direct(flags, common_abip)
00002
00003 abip_include = strjoin("-I" + [fullfile("include");
00004     fullfile("linsys");
00005     fullfile("external");
00006     fullfile("external", "amd");
00007     fullfile("external", "ldl")]);
00008
00009 platform = convertCharsToStrings(computer('arch'));
00010 lib_path = "";
00011 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
00012 % If use MKL, we suggest you set the environmental variables by oneapi,
00013 % it is typically placed at /opt/intel/oneapi/setvars.sh
00014 % alternatively, you can set the lib_path to your MKL path by your self,
00015 % For example, in linux, you may find it at /opt/intel/oneapi/mkl/2021.2.0/lib/intel64,
00016 % then you uncomment the following:
00017 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
00018 % lib_path = "/opt/intel/oneapi/mkl/2021.2.0/lib/";
00019 % if exist(lib_path) % #ok
00020 %     platform = computer('arch');
00021 %     lib_path = "C:\Program Files (x86)\Intel\oneAPI\mkl\2022.1.0\lib\intel64";
00022 %     lib_path = "-L" + lib_path;
00023 % else
00024 %     platform = "nomkl";
00025 % end
00026
00027
00028 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
00029 mkl_macro = "-DABIP_PARDISO";
00030 pardiso_src = fullfile("linsys", "abip_pardiso.c");
00031
00032 if platform == "win64"
00033     fprintf("Linking MKL in Windows \n");
00034     flags.link = [flags.link, ' -lmkl_intel_ilp64',...
00035         ' -lmkl_core', ' -lmkl_sequential'];
00036 elseif platform == "maci64"
00037     fprintf("Linking MKL in MacOS \n");
00038     flags.link = [flags.link, ' -lmkl_intel_ilp64',...
00039         ' -lmkl_core', ' -lmkl_sequential'];
00040 elseif platform == "glnxa64"
00041     fprintf("Linking MKL in Linux \n");
00042     flags.link = [flags.link, ' -lmkl_intel_ilp64',...
00043         ' -lmkl_core', ' -lmkl_sequential'];
00044 else
00045     lib_path = "";
00046     mkl_macro = "";
00047     if mexext == "mexw64"
00048         flags.link = '-lut -lmwblas -lmwlapack';
00049     else
00050         flags.link = '-lm -lut -lmwblas -lmwlapack';
00051     end % End if
00052     pardiso_src = "";
00053     fprintf("Unknown platform. Not using MKL \n");
00054 end % End if
00055
00056 if (flags.COMPILE_WITH_OPENMP)
00057     cmd = sprintf('mex -O %s %s %s %s COMPILE_FLAGS="/openmp \\\$COMPILE_FLAGS" CFLAGS="\\\$CFLAGS -fopenmp" %s
00058 %s', flags.arr, flags.LCFLAG, flags.INCS, abip_include, flags.INT);
00059 else
00059     cmd = sprintf('mex -O %s %s %s %s %s %s %s %s', mkl_macro, flags.arr, lib_path, flags.LCFLAG,
00060         flags.INCS, abip_include, flags.INT);
00061 end
00062
00062 ldl_path = fullfile("external", "ldl");
00063 amd_path = fullfile("external", "amd");
00064
00065 ldl_files = ["ldl.c"];
00066 amd_files = ["amd_order", "amd_dump", "amd_postorder", "amd_post_tree", ...
00067     "amd_aat", "amd_2", "amd_1", "amd_defaults", "amd_control", ...
00068     "amd_info", "amd_valid", "amd_global", "amd_preprocess"];
00069 amd_files = amd_files + ".c";
00070
00071 abip_ldl = fullfile(ldl_path, ldl_files);
00072 abip_ldl = strjoin(abip_ldl);
00073 abip_amd = fullfile(amd_path, amd_files);
00074 abip_amd = strjoin(abip_amd);
00075
00076 cmd = sprintf('%s %s %s %s %s', cmd, abip_amd, abip_ldl,...
00077     fullfile("linsys", "direct.c"), pardiso_src);
00078 cmd = sprintf(' %s %s %s %s %s -output abip_direct', cmd, common_abip, flags.link, flags.LOCS,
00079     flags.BLASLIB);
00079

```

```
00080 eval(cmd);  
00081
```

5.3 compile_indirect.m File Reference

Functions

- function `compile_indirect` (flags, `common_abip`) `abip_include`
- compile indirect `if` (flags.COMPILE_WITH_OPENMP) `cmd`
- end `eval` (`cmd`)

Variables

- else `cmd`

5.3.1 Function Documentation

5.3.1.1 compile_indirect()

```
function compile_indirect (  
    flags ,  
    common_abip )
```

5.3.1.2 eval()

```
end eval (  
    cmd )
```

5.3.1.3 if()

```
compile indirect if (  
    flags. COMPILE_WITH_OPENMP )
```

5.3.2 Variable Documentation

5.3.2.1 cmd

else cmd

Initial value:

```
= sprintf('mex -O %s %s %s %s %s %s %s %s %s %s %s -output abip_indirect',...
    flags.arr, flags.LCFLAG, common_abip, flags.INCS,...
    fullfile("linsys", "indirect.c"),...
    flags.link, abip_include, flags.LOCS, flags.BLASLIB, flags.INT)
```

Definition at line 13 of file `compile_indirect.m`.

5.4 compile_indirect.m

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

```
00001 function compile_indirect(flags, common_abip)
00002
00003 abip_include = strjoin("-I" + [fullfile("include");
00004     fullfile("linsys")]);
00005
00006 % compile indirect
00007 if (flags.COMPILE_WITH_OPENMP)
00008     cmd = sprintf('mex -O %s %s %s %s %s %s %s %s %s %s %s -output abip_indirect',...
00009         flags.arr, flags.LCFLAG, common_abip, flags.INCS,...
00010         fullfile("linsys", "direct", "private.c"),...
00011         flags.link, abip_include, flags.LOCS, flags.BLASLIB, flags.INT);
00012 else
00013     cmd = sprintf('mex -O %s %s %s %s %s %s %s %s %s %s %s -output abip_indirect',...
00014         flags.arr, flags.LCFLAG, common_abip, flags.INCS,...
00015         fullfile("linsys", "indirect.c"),...
00016         flags.link, abip_include, flags.LOCS, flags.BLASLIB, flags.INT);
00017 end
00018
00019 eval(cmd);
```

5.5 external/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 ABIInfo[])`
- `SuiteSparse_long amd_l_order (SuiteSparse_long n, const SuiteSparse_long Ap[], const SuiteSparse_long Ai[], SuiteSparse_long P[], abip_float Control[], abip_float ABIInfo[])`
- `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 ABIInfo[])`
- `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 ABIInfo[])`
- `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 ABIInfo[])`
- `void amd_l_info (abip_float ABIInfo[])`

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 *,...)`

5.5.1 Macro Definition Documentation

5.5.1.1 AMD_AGGRESSIVE

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

Definition at line 341 of file [amd.h](#).

5.5.1.2 AMD_CONTROL

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

Definition at line 336 of file [amd.h](#).

5.5.1.3 AMD_DATE

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

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

5.5.1.4 AMD_DEFAULT_AGGRESSIVE

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

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

5.5.1.5 AMD_DEFAULT_DENSE

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

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

5.5.1.6 AMD_DENSE

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

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

5.5.1.7 AMD_DMAX

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

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

5.5.1.8 AMD_INFO

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

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

5.5.1.9 AMD_INVALID

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

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

5.5.1.10 AMD_LNZ

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

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

5.5.1.11 AMD_MAIN_VERSION

```
#define AMD_MAIN_VERSION 2
```

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

5.5.1.12 AMD_MEMORY

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

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

5.5.1.13 AMD_N

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

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

5.5.1.14 AMD_NCMPA

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

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

5.5.1.15 AMD_NDENSE

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

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

5.5.1.16 AMD_NDIV

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

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

5.5.1.17 AMD_NMULTSUBS_LDL

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

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

5.5.1.18 AMD_NMULTSUBS_LU

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

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

5.5.1.19 AMD_NZ

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

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

5.5.1.20 AMD_NZ_A_PLUS_AT

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

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

5.5.1.21 AMD_NZDIAG

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

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

5.5.1.22 AMD_OK

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

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

5.5.1.23 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](#).

5.5.1.24 AMD_OUT_OF_MEMORY

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

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

5.5.1.25 AMD_STATUS

```
#define AMD_STATUS 0 /* return value of amd_order and amd_l_order */
```

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

5.5.1.26 AMD_SUB_VERSION

```
#define AMD_SUB_VERSION 4
```

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

5.5.1.27 AMD_SUBSUB_VERSION

```
#define AMD_SUBSUB_VERSION 6
```

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

5.5.1.28 AMD_SYMMETRY

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

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

5.5.1.29 AMD_VERSION

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

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

5.5.1.30 AMD_VERSION_CODE

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

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

5.5.1.31 EXTERN

```
#define EXTERN extern
```

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

5.5.2 Function Documentation

5.5.2.1 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[] )
```

5.5.2.2 amd_control()

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

5.5.2.3 amd_defaults()

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

5.5.2.4 amd_info()

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


5.5.2.5 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[] )
```

5.5.2.6 amd_l_control()

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

5.5.2.7 amd_l_defaults()

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

5.5.2.8 amd_l_info()

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

5.5.2.9 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[] )
```

5.5.2.10 amd_l_valid()

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

5.5.2.11 amd_order()

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

5.5.2.12 amd_valid()

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

5.5.3 Variable Documentation

5.5.3.1 amd_calloc

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

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

5.5.3.2 amd_free

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

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

5.5.3.3 amd_malloc

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

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

5.5.3.4 amd_printf

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

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

5.5.3.5 amd_realloc

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

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

5.6 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

```

5.7 external/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[])

5.7.1 Function Documentation

5.7.1.1 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](#).

5.8 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 }

```

5.9 external/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[])

5.9.1 Function Documentation

5.9.1.1 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](#).

5.10 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     ***** CAUTION: ARGUMENTS ARE NOT CHECKED FOR ERRORS ON INPUT. *****
00126     ***** If you want error checking, a more versatile input format, and a **
00127     ***** simpler user interface, use amd_order or amd_l_order instead. **
00128     ***** This routine is not meant to be user-callable. *****
00129     *****
00130     * -----
00131     * References:
00132     * -----
00133     *
00134     * [1] Timothy A. Davis and Iain Duff, "An unsymmetric-pattern multifrontal
00135     *     method for sparse LU factorization", SIAM J. Matrix Analysis and
00136     *     Applications, vol. 18, no. 1, pp. 140-158. Discusses UMFPACK / MA38,
00137     *     which first introduced the approximate minimum degree used by this
00138     *     routine.
00139     *
00140     *
00141     *
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 */
00769         /* =====
00770
00771         /* ----- */
00772         /* find next supervariable for elimination */
00773         /* ----- */
00774
00775         ASSERT (mindeg >= 0 && mindeg < n) ;
00776         for (deg = mindeg ; deg < n ; deg++)
00777         {
00778             me = Head [deg] ;
00779             if (me != EMPTY) break ;
00780         }
00781         mindeg = deg ;
00782         ASSERT (me >= 0 && me < n) ;
00783         AMD_DEBUG1 (("=====me: "ID"\n", me)) ;
00784
00785         /* ----- */
00786         /* remove chosen variable from link list */
00787         /* ----- */
00788
00789         inext = Next [me] ;
00790         ASSERT (inext >= EMPTY && inext < n) ;
00791         if (inext != EMPTY) Last [inext] = EMPTY ;
00792         Head [deg] = inext ;
00793
00794         /* ----- */
00795         /* me represents the elimination of pivots nel to nel+Nv[me]-1. */
00796         /* place me itself as the first in this set. */
00797         /* ----- */
00798
00799         elenme = Elen [me] ;
00800         nvpiv = Nv [me] ;
00801         ASSERT (nvpiv > 0) ;
00802         nel += nvpiv ;
00803
00804         /* =====
00805
00806         /* CONSTRUCT NEW ELEMENT */
00806         /* =====
00807
00808         /* -----
00809         * At this point, me is the pivotal supervariable. It will be
00810         * converted into the current element. Scan list of the pivotal
00811         * supervariable, me, setting tree pointers and constructing new list
00812         * of supervariables for the new element, me. p is a pointer to the
00813         * current position in the old list.
00814         * ----- */
00815
00816         /* flag the variable "me" as being in Lme by negating Nv [me] */
00817         Nv [me] = -nvpiv ;
00818         degme = 0 ;
00819         ASSERT (Pe [me] >= 0 && Pe [me] < iwlen) ;
00820
00821         if (elenme == 0)
00822         {
00823             /* -----
00824
00825             /* construct the new element in place */
00825             /* ----- */
00826
00827             pme1 = Pe [me] ;
00828             pme2 = pme1 - 1 ;
00829
00830             for (p = pme1 ; p <= pme1 + Len [me] - 1 ; p++)
00831             {
00832                 i = Iw [p] ;
00833                 ASSERT (i >= 0 && i < n && Nv [i] >= 0) ;

```

```

00834             nvi = Nv [i] ;
00835
00836             if (nvi > 0)
00837             {
00838
00839                 /*
00840                 /* i is a principal variable not yet placed in Lme. */
00841                 /* store i in new list */
00842                 /* ----- */
00843
00844                 /* flag i as being in Lme by negating Nv [i] */
00845                 degme += nvi ;
00846                 Nv [i] = -nvi ;
00847                 Iw [pme2] = i ;
00848                 /* ----- */
00849
00850                 /* remove variable i from degree list. */
00851                 /* ----- */
00852
00853                 ilast = Last [i] ;
00854                 inext = Next [i] ;
00855                 ASSERT (ilast >= EMPTY && ilast < n) ;
00856                 ASSERT (inext >= EMPTY && inext < n) ;
00857                 if (inext != EMPTY) Last [inext] = ilast ;
00858                 if (ilast != EMPTY)
00859                 {
00860                     Next [ilast] = inext ;
00861                 }
00862                 else
00863                 {
00864                     /* i is at the head of the degree list */
00865                     ASSERT (Degree [i] >= 0 && Degree [i] < n) ;
00866                     Head [Degree [i]] = inext ;
00867                 }
00868             }
00869         }
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 [psrc++] ;
00981
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                                     {
00990                                         Iw [pdst++] = Iw [psrc] ;
00991                                     }
00992                                     pme1 = pl ;
00993                                     pfree = pdst ;
00994                                     pj = Pe [e] ;
00995                                     p = Pe [me] ;
00996                                     }
00997                                     /* -----
00998 */
00999                                     /* i is a principal variable not yet placed in Lme */
01000                                     /* store i in new list */
01001                                     /* -----
01002 */
01003                                     /* flag i as being in Lme by negating Nv [i] */
01004                                     degme += nvi ;
01005                                     Nv [i] = -nvi ;
01006                                     Iw [pfree++] = i ;
01007                                     AMD_DEBUG2 (("      s: "ID"      nv "ID"\n", i, Nv
[i]));
01008                                     /* -----
01009 */
01010                                     /* remove variable i from degree link list */
01011                                     /* -----
01012 */
01013                                     ilast = Last [i] ;
01014                                     inext = Next [i] ;
01015                                     ASSERT (ilast >= EMPTY && ilast < n) ;
01016                                     ASSERT (inext >= EMPTY && inext < n) ;
01017                                     if (inext != EMPTY) Last [inext] = ilast ;
01018                                     if (ilast != EMPTY)
01019                                     {
01020                                         Next [ilast] = inext ;
01021                                     }
01022                                     else
01023                                     {
01024                                         /* i is at the head of the degree list */
01025                                         ASSERT (Degree [i] >= 0 && Degree [i] <
n) ;
01026                                         Head [Degree [i]] = inext ;
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                                     }
01059                                     }
01060                                     }
01061                                     }
01062                                     }

```



```

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             /* UMFPAK/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, degree) ;
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         for (p = Pe [i] + 1 ; p <= Pe [i] + ln - 1 ; p++)
01429         {
01430             ASSERT (Iw [p] >= 0 && Iw [p] < n) ;
01431             W [Iw [p]] = wflg ;
01432         }
01433
01434         /* -----
*/
01435         /* scan every other entry j following i in bucket */
01436         /* -----
*/
01437
01438         jlast = i ;
01439         j = Next [i] ;
01440         ASSERT (j >= EMPTY && j < n) ;
01441
01442         while (j != EMPTY)
01443         {
01444             /* -----
*/
01445             /* check if j and i have identical nonzero pattern */
01446             /* -----
*/
01447
01448             AMD_DEBUG3 (("compare i "ID" and j "ID"\n", i, j)) ;
01449
01450             /* check if i and j have the same Len and Elen */
01451             ASSERT (Len [j] >= 0 && Elen [j] >= 0) ;
01452             ASSERT (Pe [j] >= 0 && Pe [j] < iwlen) ;
01453             ok = (Len [j] == ln) && (Elen [j] == eln) ;
01454
01455             /* skip the first element
in the list (me) */
01456             for (p = Pe [j] + 1 ; ok && p <= Pe [j] + ln - 1 ;
p++)
01457             {
01458                 ASSERT (Iw [p] >= 0 && Iw [p] < n) ;
01459                 if (W [Iw [p]] != wflg) ok = 0 ;
01460             }
01461
01462             if (ok)
01463             {
01464                 /*
----- */
01465                 /* found it! j can be absorbed into i */
01466                 /*
----- */
01467
01468                 AMD_DEBUG1 (("found it! j "ID" => i
ID"\n", j, i));
01469                 Pe [j] = FLIP (i) ;
01470
01471                 /* 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 (("Tree:\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     }
01725     #endif

```



```

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 }

```

5.11 external/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[])

5.11.1 Function Documentation

5.11.1.1 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](#).

5.12 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 }

```

5.13 external/amd/amd_control.c File Reference

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

Functions

- GLOBAL void AMD_control (abip_float Control[])

5.13.1 Function Documentation

5.13.1.1 AMD_control()

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

Definition at line 18 of file [amd_control.c](#).

5.14 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 }

```

5.15 external/amd/amd_defaults.c File Reference

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

Functions

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

5.15.1 Function Documentation

5.15.1.1 AMD_defaults()

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

Definition at line 21 of file [amd_defaults.c](#).

5.16 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 }
```

5.17 external/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

5.17.1 Function Documentation

5.17.1.1 AMD_debug_init()

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

Definition at line 29 of file [amd_dump.c](#).

5.17.1.2 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](#).

5.17.2 Variable Documentation

5.17.2.1 AMD_debug

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

Definition at line 21 of file [amd_dump.c](#).

5.18 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

```

5.19 external/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](#)

5.19.1 Macro Definition Documentation

5.19.1.1 ABIP_NULL

```
#define ABIP_NULL 0
```

Definition at line 20 of file [amd_global.c](#).

5.19.2 Variable Documentation

5.19.2.1 amd_calloc

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

Definition at line 54 of file [amd_global.c](#).

5.19.2.2 amd_free

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

Definition at line 52 of file [amd_global.c](#).

5.19.2.3 amd_malloc

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

Definition at line 51 of file [amd_global.c](#).

5.19.2.4 amd_printf

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

Definition at line 75 of file [amd_global.c](#).

5.19.2.5 amd_realloc

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

Definition at line 53 of file [amd_global.c](#).

5.20 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 ;

```

5.21 external/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[])`

5.21.1 Macro Definition Documentation

5.21.1.1 PRI

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

Definition at line 17 of file [amd_info.c](#).

5.21.2 Function Documentation

5.21.2.1 AMD_info()

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

Definition at line 19 of file [amd_info.c](#).

5.22 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 }

```

5.23 external/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[])`

5.23.1 Macro Definition Documentation

5.23.1.1 ABIP_NULL

```
#define ABIP_NULL 0
```

Definition at line 138 of file [amd_internal.h](#).

5.23.1.2 AMD_1

```
#define AMD_1 amd_1
```

Definition at line 187 of file [amd_internal.h](#).

5.23.1.3 AMD_2

```
#define AMD_2 amd_2
```

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

5.23.1.4 AMD_aat

```
#define AMD_aat amd_aat
```

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

5.23.1.5 AMD_control

```
#define AMD_control amd_control
```

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

5.23.1.6 AMD_debug

```
#define AMD_debug amd_debug
```

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

5.23.1.7 AMD_DEBUG0

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

Definition at line 328 of file [amd_internal.h](#).

5.23.1.8 AMD_DEBUG1

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

Definition at line 329 of file [amd_internal.h](#).

5.23.1.9 AMD_DEBUG2

```
#define AMD_DEBUG2(  
    params )
```

Definition at line 330 of file [amd_internal.h](#).

5.23.1.10 AMD_DEBUG3

```
#define AMD_DEBUG3(  
    params )
```

Definition at line 331 of file [amd_internal.h](#).

5.23.1.11 AMD_DEBUG4

```
#define AMD_DEBUG4(  
    params )
```

Definition at line 332 of file [amd_internal.h](#).

5.23.1.12 AMD_debug_init

```
#define AMD_debug_init amd_debug_init
```

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

5.23.1.13 AMD_defaults

```
#define AMD_defaults amd\_defaults
```

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

5.23.1.14 AMD_dump

```
#define AMD_dump amd_dump
```

Definition at line 193 of file [amd_internal.h](#).

5.23.1.15 AMD_info

```
#define AMD_info amd_info
```

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

5.23.1.16 AMD_order

```
#define AMD_order amd_order
```

Definition at line 183 of file [amd_internal.h](#).

5.23.1.17 AMD_post_tree

```
#define AMD_post_tree amd_post_tree
```

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

5.23.1.18 AMD_postorder

```
#define AMD_postorder amd_postorder
```

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

5.23.1.19 AMD_preprocess

```
#define AMD_preprocess amd_preprocess
```

Definition at line 196 of file [amd_internal.h](#).

5.23.1.20 AMD_valid

```
#define AMD_valid amd_valid
```

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

5.23.1.21 ASSERT

```
#define ASSERT(  
    expression )
```

Definition at line 327 of file [amd_internal.h](#).

5.23.1.22 EMPTY [1/2]

```
#define EMPTY (-1)
```

Definition at line 129 of file [amd_internal.h](#).

5.23.1.23 EMPTY [2/2]

```
#define EMPTY (-1)
```

Definition at line 129 of file [amd_internal.h](#).

5.23.1.24 FALSE

```
#define FALSE (0)
```

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

5.23.1.25 FLIP

```
#define FLIP(  
    i )  -(i)-2)
```

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

5.23.1.26 GLOBAL

```
#define GLOBAL
```

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

5.23.1.27 ID

```
#define ID "%d"
```

Definition at line 180 of file [amd_internal.h](#).

5.23.1.28 IMPLIES

```
#define IMPLIES(  
    p,  
    q )  ( ! (p) || (q) )
```

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

5.23.1.29 Int

```
#define Int int
```

Definition at line 179 of file [amd_internal.h](#).

5.23.1.30 Int_MAX

```
#define Int_MAX INT_MAX
```

Definition at line 181 of file [amd_internal.h](#).

5.23.1.31 MAX

```
#define MAX(  
    a,  
    b )  (( (a) > (b) ) ? (a) : (b) )
```

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

5.23.1.32 MIN

```
#define MIN(  
    a,  
    b )  (( (a) < (b) ) ? (a) : (b) )
```

Definition at line 111 of file [amd_internal.h](#).

5.23.1.33 PRINTF

```
#define PRINTF(  
    params ) { if (amd_printf != ABIP_NULL) (void) amd_printf params ; }
```

Definition at line 205 of file [amd_internal.h](#).

5.23.1.34 PRIVATE

```
#define PRIVATE static
```

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

5.23.1.35 SIZE_T_MAX

```
#define SIZE_T_MAX ((size_t) (-1))
```

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

5.23.1.36 TRUE

```
#define TRUE (1)
```

Definition at line 125 of file [amd_internal.h](#).

5.23.1.37 UNFLIP

```
#define UNFLIP(  
    i ) ((i < EMPTY) ? FLIP (i) : (i))
```

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

5.23.2 Function Documentation

5.23.2.1 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](#).

5.23.2.2 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](#).

5.23.2.3 AMD_post_tree()

```
GLOBAL Int AMD_post_tree (
    Int root,
    Int k,
    Int Child[],
    const Int Sibling[],
    Int Order[],
    Int Stack[] )
```

5.23.2.4 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](#).

5.23.2.5 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](#).

5.24 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

```

5.25 external/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\[\]\)](#)

5.25.1 Function Documentation

5.25.1.1 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](#).

5.26 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         if (info) ABIPInfo [AMD_STATUS] = AMD_OUT_OF_MEMORY ;
00095         return (AMD_OUT_OF_MEMORY) ; /* problem too large */
00096     }
00097
00098     /* check the input matrix: AMD_OK, AMD_INVALID, or AMD_OK_BUT_JUMBLED */
00099

```

```

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     }

```

5.27 external/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)`

5.27.1 Function Documentation

5.27.1.1 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](#).

5.28 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 NDEBBUG
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 }

```

5.29 external/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[])

5.29.1 Function Documentation

5.29.1.1 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](#).

5.30 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 }

```

5.31 external/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[])

5.31.1 Function Documentation

5.31.1.1 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](#).

5.32 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 }

```

5.33 external/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[])

5.33.1 Function Documentation

5.33.1.1 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](#).

5.34 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     }

```

5.35 external/ldl/ldl.c File Reference

```
#include "ldl.h"
```

Functions

- void [LDL_symbolic](#) ([LDL_int](#) n, [LDL_int](#) Ap[], [LDL_int](#) Ai[], [LDL_int](#) Lp[], [LDL_int](#) Parent[], [LDL_int](#) Lnz[], [LDL_int](#) Flag[], [LDL_int](#) P[], [LDL_int](#) Pinv[])
- [LDL_int](#) [LDL_numeric](#) ([LDL_int](#) n, [LDL_int](#) Ap[], [LDL_int](#) Ai[], [abip_float](#) Ax[], [LDL_int](#) Lp[], [LDL_int](#) Parent[], [LDL_int](#) Lnz[], [LDL_int](#) Li[], [abip_float](#) Lx[], [abip_float](#) D[], [abip_float](#) Y[], [LDL_int](#) Pattern[], [LDL_int](#) Flag[], [LDL_int](#) P[], [LDL_int](#) Pinv[])
- void [LDL_ksolve](#) ([LDL_int](#) n, [abip_float](#) X[], [LDL_int](#) Lp[], [LDL_int](#) Li[], [abip_float](#) Lx[])
- void [LDL_dsolve](#) ([LDL_int](#) n, [abip_float](#) X[], [abip_float](#) D[])
- void [LDL_itsolve](#) ([LDL_int](#) n, [abip_float](#) X[], [LDL_int](#) Lp[], [LDL_int](#) Li[], [abip_float](#) Lx[])
- void [LDL_perm](#) ([LDL_int](#) n, [abip_float](#) X[], [abip_float](#) B[], [LDL_int](#) P[])
- void [LDL_permt](#) ([LDL_int](#) n, [abip_float](#) X[], [abip_float](#) B[], [LDL_int](#) P[])
- [LDL_int](#) [LDL_valid_perm](#) ([LDL_int](#) n, [LDL_int](#) P[], [LDL_int](#) Flag[])
- [LDL_int](#) [LDL_valid_matrix](#) ([LDL_int](#) n, [LDL_int](#) Ap[], [LDL_int](#) Ai[])

5.35.1 Function Documentation

5.35.1.1 LDL_dsolve()

```

void LDL_dsolve (
    LDL\_int n,
    abip\_float X[],
    abip\_float D[] )

```

Definition at line [385](#) of file [ldl.c](#).

5.35.1.2 LDL_ksolve()

```

void LDL_ksolve (
    LDL\_int n,
    abip\_float X[],
    LDL\_int Lp[],
    LDL\_int Li[],
    abip\_float Lx[] )

```

Definition at line [357](#) of file [ldl.c](#).

5.35.1.3 LDL_Itsolve()

```
void LDL_Itsolve (
    LDL_int n,
    abip_float X[],
    LDL_int Lp[],
    LDL_int Li[],
    abip_float Lx[] )
```

Definition at line 431 of file [ldl.c](#).

5.35.1.4 LDL_numeric()

```
LDL_int LDL_numeric (
    LDL_int n,
    LDL_int Ap[],
    LDL_int Ai[],
    abip_float Ax[],
    LDL_int Lp[],
    LDL_int Parent[],
    LDL_int Lnz[],
    LDL_int Li[],
    abip_float Lx[],
    abip_float D[],
    abip_float Y[],
    LDL_int Pattern[],
    LDL_int Flag[],
    LDL_int P[],
    LDL_int Pinv[] )
```

Definition at line 262 of file [ldl.c](#).

5.35.1.5 LDL_perm()

```
void LDL_perm (
    LDL_int n,
    abip_float X[],
    abip_float B[],
    LDL_int P[] )
```

Definition at line 459 of file [ldl.c](#).

5.35.1.6 LDL_permt()

```
void LDL_permt (
    LDL_int n,
    abip_float X[],
    abip_float B[],
    LDL_int P[] )
```

Definition at line 506 of file [ldl.c](#).

5.35.1.7 LDL_symbolic()

```
void LDL_symbolic (
    LDL_int n,
    LDL_int Ap[],
    LDL_int Ai[],
    LDL_int Lp[],
    LDL_int Parent[],
    LDL_int Lnz[],
    LDL_int Flag[],
    LDL_int P[],
    LDL_int Pinv[] )
```

Definition at line 187 of file [ldl.c](#).

5.35.1.8 LDL_valid_matrix()

```
LDL_int LDL_valid_matrix (
    LDL_int n,
    LDL_int Ap[],
    LDL_int Ai[] )
```

Definition at line 603 of file [ldl.c](#).

5.35.1.9 LDL_valid_perm()

```
LDL_int LDL_valid_perm (
    LDL_int n,
    LDL_int P[],
    LDL_int Flag[] )
```

Definition at line 553 of file [ldl.c](#).

5.36 ldl.c

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

```
00001 /* ===== */
00002 /* === ldl.c: sparse LDL' factorization and solve package ===== */
00003 /* ===== */
00004
00005 /* LDL: a simple set of routines for sparse LDL' factorization. These routines
00006 * are not terrifically fast (they do not use dense matrix kernels), but the
00007 * code is very short. The purpose is to illustrate the algorithms in a very
00008 * concise manner, primarily for educational purposes. Although the code is
00009 * very concise, this package is slightly faster than the built-in sparse
00010 * Cholesky factorization in MATLAB 7.0 (chol), when using the same input
00011 * permutation.
00012 *
00013 * The routines compute the LDL' factorization of a real sparse symmetric
00014 * matrix A (or PAP' if a permutation P is supplied), and solve upper
00015 * and lower triangular systems with the resulting L and D factors. If A is
00016 * positive definite then the factorization will be accurate. A can be
00017 * indefinite (with negative values on the diagonal D), but in this case no
00018 * guarantee of accuracy is provided, since no numeric pivoting is performed.
```

```

00019 *
00020 * The n-by-n sparse matrix A is in compressed-column form. The nonzero values
00021 * in column j are stored in Ax [Ap [j] ... Ap [j+1]-1], with corresponding row
00022 * indices in Ai [Ap [j] ... Ap [j+1]-1]. Ap [0] = 0 is required, and thus
00023 * nz = Ap [n] is the number of nonzeros in A. Ap is an int array of size n+1.
00024 * The int array Ai and the abip_float array Ax are of size nz. This data structure
00025 * is identical to the one used by MATLAB, except for the following
00026 * generalizations. The row indices in each column of A need not be in any
00027 * particular order, although they must be in the range 0 to n-1. Duplicate
00028 * entries can be present; any duplicates are summed. That is, if row index i
00029 * appears twice in a column j, then the value of A (i,j) is the sum of the two
00030 * entries. The data structure used here for the input matrix A is more
00031 * flexible than MATLAB's, which requires sorted columns with no duplicate
00032 * entries.
00033 *
00034 * Only the diagonal and upper triangular part of A (or PAP' if a permutation
00035 * P is provided) is accessed. The lower triangular parts of the matrix A or
00036 * PAP' can be present, but they are ignored.
00037 *
00038 * The optional input permutation is provided as an array P of length n. If
00039 * P [k] = j, the row and column j of A is the kth row and column of PAP'.
00040 * If P is present then the factorization is LDL' = PAP' or L*D*L' = A(P,P) in
00041 * 0-based MATLAB notation. If P is not present (a null pointer) then no
00042 * permutation is performed, and the factorization is LDL' = A.
00043 *
00044 * The lower triangular matrix L is stored in the same compressed-column
00045 * form (an int Lp array of size n+1, an int Li array of size Lp [n], and a
00046 * abip_float array Lx of the same size as Li). It has a unit diagonal, which is
00047 * not stored. The row indices in each column of L are always returned in
00048 * ascending order, with no duplicate entries. This format is compatible with
00049 * MATLAB, except that it would be more convenient for MATLAB to include the
00050 * unit diagonal of L. Doing so here would add additional complexity to the
00051 * code, and is thus omitted in the interest of keeping this code short and
00052 * readable.
00053 *
00054 * The elimination tree is held in the Parent [0..n-1] array. It is normally
00055 * not required by the user, but it is required by ldl_numeric. The diagonal
00056 * matrix D is held as an array D [0..n-1] of size n.
00057 *
00058 * -----
00059 * C-callable routines:
00060 * -----
00061 *
00062 * ldl_symbolic: Given the pattern of A, computes the Lp and Parent arrays
00063 * required by ldl_numeric. Takes time proportional to the number of
00064 * nonzeros in L. Computes the inverse Pinv of P if P is provided.
00065 * Also returns Lnz, the count of nonzeros in each column of L below
00066 * the diagonal (this is not required by ldl_numeric).
00067 * ldl_numeric: Given the pattern and numerical values of A, the Lp array,
00068 * the Parent array, and P and Pinv if applicable, computes the
00069 * pattern and numerical values of L and D.
00070 * ldl_solve: Solves Lx=b for a dense vector b.
00071 * ldl_dsolve: Solves Dx=b for a dense vector b.
00072 * ldl_ltsolve: Solves L'x=b for a dense vector b.
00073 * ldl_perm: Computes x=Pb for a dense vector b.
00074 * ldl_permt: Computes x=P'b for a dense vector b.
00075 * ldl_valid_perm: checks the validity of a permutation vector
00076 * ldl_valid_matrix: checks the validity of the sparse matrix A
00077 *
00078 * -----
00079 * Limitations of this package:
00080 * -----
00081 *
00082 * In the interest of keeping this code simple and readable, ldl_symbolic and
00083 * ldl_numeric assume their inputs are valid. You can check your own inputs
00084 * prior to calling these routines with the ldl_valid_perm and ldl_valid_matrix
00085 * routines. Except for the two ldl_valid_* routines, no routine checks to see
00086 * if the array arguments are present (non-NULL). Like all C routines, no
00087 * routine can determine if the arrays are long enough and don't overlap.
00088 *
00089 * The ldl_numeric does check the numerical factorization, however. It returns
00090 * n if the factorization is successful. If D (k,k) is zero, then k is
00091 * returned, and L is only partially computed.
00092 *
00093 * No pivoting to control fill-in is performed, which is often critical for
00094 * obtaining good performance. I recommend that you compute the permutation P
00095 * using AMD or SYMAMD (approximate minimum degree ordering routines), or an
00096 * appropriate graph-partitioning based ordering. See the ldlmain.c routine for
00097 * an example in MATLAB, and the ldlmain.c stand-alone C program for examples of
00098 * how to find P. Routines for manipulating compressed-column matrices are
00099 * available in UMFPACK. AMD, SYMAMD, UMFPACK, and this LDL package are all
00100 * available at http://www.suitesparse.com.
00101 *
00102 * -----
00103 * Possible simplifications:
00104 * -----
00105 *

```

```

00106 * These routines could be made even simpler with a few additional assumptions.
00107 * If no input permutation were performed, the caller would have to permute the
00108 * matrix first, but the computation of Pinv, and the use of P and Pinv could be
00109 * removed. If only the diagonal and upper triangular part of A or PAP' are
00110 * present, then the tests in the "if (i < k)" statement in ldl_symbolic and
00111 * "if (i <= k)" in ldl_numeric, are always true, and could be removed (i can
00112 * equal k in ldl_symbolic, but then the body of the if statement would
00113 * correctly do no work since Flag[k] == k). If we could assume that no
00114 * duplicate entries are present, then the statement Y[i] += Ax[p] could be
00115 * replaced with Y[i] = Ax[p] in ldl_numeric.
00116 *
00117 * -----
00118 * Description of the method:
00119 * -----
00120 *
00121 * LDL computes the symbolic factorization by finding the pattern of L one row
00122 * at a time. It does this based on the following theory. Consider a sparse
00123 * system Lx=b, where L, x, and b, are all sparse, and where L comes from a
00124 * Cholesky (or LDL') factorization. The elimination tree (etree) of L is
00125 * defined as follows. The parent of node j is the smallest k > j such that
00126 * L(k,j) is nonzero. Node j has no parent if column j of L is completely zero
00127 * below the diagonal (j is a root of the etree in this case). The nonzero
00128 * pattern of x is the union of the paths from each node i to the root, for
00129 * each nonzero b(i). To compute the numerical solution to Lx=b, we can
00130 * traverse the columns of L corresponding to nonzero values of x. This
00131 * traversal does not need to be done in the order 0 to n-1. It can be done in
00132 * any "topological" order, such that x(i) is computed before x(j) if i is a
00133 * descendant of j in the elimination tree.
00134 *
00135 * The row-form of the LDL' factorization is shown in the MATLAB function
00136 * ldlrow.m in this LDL package. Note that row k of L is found via a sparse
00137 * triangular solve of L(1:k-1, 1:k-1) \ A(1:k-1, k), to use 1-based MATLAB
00138 * notation. Thus, we can start with the nonzero pattern of the kth column of
00139 * A (above the diagonal), follow the paths up to the root of the etree of the
00140 * (k-1)-by-(k-1) leading submatrix of L, and obtain the pattern of the kth row
00141 * of L. Note that we only need the leading (k-1)-by-(k-1) submatrix of L to
00142 * do this. The elimination tree can be constructed as we go.
00143 *
00144 * The symbolic factorization does the same thing, except that it discards the
00145 * pattern of L as it is computed. It simply counts the number of nonzeros in
00146 * each column of L and then constructs the Lp index array when it's done. The
00147 * symbolic factorization does not need to do this in topological order.
00148 * Compare ldl_symbolic with the first part of ldl_numeric, and note that the
00149 * while (len > 0) loop is not present in ldl_symbolic.
00150 *
00151 * Copyright (c) 2006 by Timothy A Davis, http://www.suitesparse.com.
00152 * All Rights Reserved. Developed while on sabbatical
00153 * at Stanford University and Lawrence Berkeley National Laboratory. Refer to
00154 * the README file for the License.
00155 */
00156
00157 #include "ldl.h"
00158
00159 /* ===== */
00160 /* == ldl_symbolic ===== */
00161 /* ===== */
00162
00163 /* The input to this routine is a sparse matrix A, stored in column form, and
00164 * an optional permutation P. The output is the elimination tree
00165 * and the number of nonzeros in each column of L. Parent[i] = k if k is the
00166 * parent of i in the tree. The Parent array is required by ldl_numeric.
00167 * Lnz[k] gives the number of nonzeros in the kth column of L, excluding the
00168 * diagonal.
00169 *
00170 * One workspace vector (Flag) of size n is required.
00171 *
00172 * If P is NULL, then it is ignored. The factorization will be LDL' = A.
00173 * Pinv is not computed. In this case, neither P nor Pinv are required by
00174 * ldl_numeric.
00175 *
00176 * If P is not NULL, then it is assumed to be a valid permutation. If
00177 * row and column j of A is the kth pivot, the P[k] = j. The factorization
00178 * will be LDL' = PAP', or A(p,p) in MATLAB notation. The inverse permutation
00179 * Pinv is computed, where Pinv[j] = k if P[k] = j. In this case, both P
00180 * and Pinv are required as inputs to ldl_numeric.
00181 *
00182 * The floating-point operation count of the subsequent call to ldl_numeric
00183 * is not returned, but could be computed after ldl_symbolic is done. It is
00184 * the sum of (Lnz[k]) * (Lnz[k] + 2) for k = 0 to n-1.
00185 */
00186
00187 void LDL_symbolic
00188 (
00189     LDL_int n,          /* A and L are n-by-n, where n >= 0 */
00190     LDL_int Ap [ ],     /* input of size n+1, not modified */
00191     LDL_int Ai [ ],     /* input of size nz=Ap[n], not modified */
00192     LDL_int Lp [ ],     /* output of size n+1, not defined on input */

```

```

00193     LDL_int Parent [ ], /* output of size n, not defined on input */
00194     LDL_int Lnz [ ], /* output of size n, not defined on input */
00195     LDL_int Flag [ ], /* workspace of size n, not defn. on input or output */
00196     LDL_int P [ ], /* optional input of size n */
00197     LDL_int Pinv [ ] /* optional output of size n (used if P is not NULL) */
00198 )
00199 {
00200     LDL_int i;
00201     LDL_int k;
00202     LDL_int p;
00203     LDL_int kk;
00204     LDL_int p2 ;
00205
00206     if (P)
00207     {
00208         /* If P is present then compute Pinv, the inverse of P */
00209         for (k = 0 ; k < n ; k++)
00210         {
00211             Pinv [P [k]] = k ;
00212         }
00213     }
00214
00215     for (k = 0 ; k < n ; k++)
00216     {
00217         /* L(k,:) pattern: all nodes reachable in etree from nz in A(0:k-1,k) */
00218         Parent [k] = -1 ; /* parent of k is not yet known */
00219         Flag [k] = k ; /* mark node k as visited */
00220         Lnz [k] = 0 ; /* count of nonzeros in column k of L */
00221         kk = (P) ? (P [k]) : (k) ; /* kth original, or permuted, column */
00222         p2 = Ap [kk+1] ;
00223
00224         for (p = Ap [kk] ; p < p2 ; p++)
00225         {
00226             /* A (i,k) is nonzero (original or permuted A) */
00227             i = (Pinv) ? (Pinv [Ai [p]]) : (Ai [p]) ;
00228
00229             if (i < k)
00230             {
00231                 /* follow path from i to root of etree, stop at flagged node */
00232                 for ( ; Flag [i] != k ; i = Parent [i])
00233                 {
00234                     /* find parent of i if not yet determined */
00235                     if (Parent [i] == -1) Parent [i] = k ;
00236                     Lnz [i]++ ; /* L (k,i) is nonzero */
00237                     Flag [i] = k ; /* mark i as visited */
00238                 }
00239             }
00240         }
00241     }
00242
00243     /* construct Lp index array from Lnz column counts */
00244     Lp [0] = 0 ;
00245     for (k = 0 ; k < n ; k++)
00246     {
00247         Lp [k+1] = Lp [k] + Lnz [k] ;
00248     }
00249 }
00250
00251
00252 /* ===== */
00253 /* == ldl_numeric ===== */
00254 /* ===== */
00255
00256 /* Given a sparse matrix A (the arguments n, Ap, Ai, and Ax) and its symbolic
00257 * analysis (Lp and Parent, and optionally P and Pinv), compute the numeric LDL'
00258 * factorization of A or PAP'. The outputs of this routine are arguments Li,
00259 * Lx, and D. It also requires three size-n workspaces (Y, Pattern, and Flag).
00260 */
00261
00262 LDL_int LDL_numeric /* returns n if successful, k if D (k,k) is zero */
00263 (
00264     LDL_int n, /* A and L are n-by-n, where n >= 0 */
00265     LDL_int Ap [ ], /* input of size n+1, not modified */
00266     LDL_int Ai [ ], /* input of size nz=Ap[n], not modified */
00267     abip_float Ax [ ], /* input of size nz=Ap[n], not modified */
00268     LDL_int Lp [ ], /* input of size n+1, not modified */
00269     LDL_int Parent [ ], /* input of size n, not modified */
00270     LDL_int Lnz [ ], /* output of size n, not defn. on input */
00271     LDL_int Li [ ], /* output of size lnz=Lp[n], not defined on input */
00272     abip_float Lx [ ], /* output of size lnz=Lp[n], not defined on input */
00273     abip_float D [ ], /* output of size n, not defined on input */
00274     abip_float Y [ ], /* workspace of size n, not defn. on input or output */
00275     LDL_int Pattern [ ], /* workspace of size n, not defn. on input or output */
00276     LDL_int Flag [ ], /* workspace of size n, not defn. on input or output */
00277     LDL_int P [ ], /* optional input of size n */
00278     LDL_int Pinv [ ] /* optional input of size n */
00279 )

```



```

00280 {
00281     abip_float yi;
00282     abip_float l_ki ;
00283
00284     LDL_int i;
00285     LDL_int k;
00286     LDL_int p;
00287     LDL_int kk;
00288     LDL_int p2;
00289     LDL_int len;
00290     LDL_int top;
00291
00292     for (k = 0 ; k < n ; k++)
00293     {
00294
00295         if(abip_is_interrupted())
00296         {
00297             abip_printf("Interrupt detected in factorization! \n");
00298             return -1;
00299         }
00300
00301         /* compute nonzero Pattern of kth row of L, in topological order */
00302         Y [k] = 0.0 ;          /* Y(0:k) is now all zero */
00303         top = n ;              /* stack for pattern is empty */
00304         Flag [k] = k ;         /* mark node k as visited */
00305         Lnz [k] = 0 ;          /* count of nonzeros in column k of L */
00306         kk = (P) ? (P [k]) : (k) ; /* kth original, or permuted, column */
00307         p2 = Ap [kk+1] ;
00308
00309         for (p = Ap [kk] ; p < p2 ; p++)
00310         {
00311             i = (Pinv) ? (Pinv [Ai [p]]) : (Ai [p]) ; /* get A(i,k) */
00312
00313             if (i <= k)
00314             {
00315                 Y [i] += Ax [p] ; /* scatter A(i,k) into Y (sum duplicates) */
00316                 for (len = 0 ; Flag [i] != k ; i = Parent [i])
00317                 {
00318                     Pattern [len++] = i ; /* L(k,i) is nonzero */
00319                     Flag [i] = k ; /* mark i as visited */
00320                 }
00321                 while (len > 0) Pattern [--top] = Pattern [--len] ;
00322             }
00323         }
00324
00325         /* compute numerical values kth row of L (a sparse triangular solve) */
00326         D [k] = Y [k] ; /* get D(k,k) and clear Y(k) */
00327         Y [k] = 0.0 ;
00328         for ( ; top < n ; top++)
00329         {
00330             i = Pattern [top] ; /* Pattern [top:n-1] is pattern of L(:,k) */
00331             yi = Y [i] ; /* get and clear Y(i) */
00332             Y [i] = 0.0 ;
00333             p2 = Lp [i] + Lnz [i] ;
00334
00335             for (p = Lp [i] ; p < p2 ; p++)
00336             {
00337                 Y [Li [p]] -= Lx [p] * yi ;
00338             }
00339
00340             l_ki = yi / D [i] ; /* the nonzero entry L(k,i) */
00341             D [k] -= l_ki * yi ;
00342             Li [p] = k ; /* store L(k,i) in column form of L */
00343             Lx [p] = l_ki ;
00344             Lnz [i]++ ; /* increment count of nonzeros in col i */
00345         }
00346
00347         if (D [k] == 0.0) return (k) ; /* failure, D(k,k) is zero */
00348     }
00349     return (n) ; /* success, diagonal of D is all nonzero */
00350 }
00351
00352
00353 /* ===== */
00354 /* === ldl_solve: solve Lx=b ===== */
00355 /* ===== */
00356
00357 void LDL_solve
00358 (
00359     LDL_int n, /* L is n-by-n, where n >= 0 */
00360     abip_float X [ ], /* size n. right-hand-side on input, soln. on output */
00361     LDL_int Lp [ ], /* input of size n+1, not modified */
00362     LDL_int Li [ ], /* input of size lnz=Lp[n], not modified */
00363     abip_float Lx [ ] /* input of size lnz=Lp[n], not modified */
00364 )
00365 {
00366     LDL_int j;

```

```

00367         LDL_int p;
00368         LDL_int p2 ;
00369
00370         for (j = 0 ; j < n ; j++)
00371         {
00372             p2 = Lp [j+1] ;
00373             for (p = Lp [j] ; p < p2 ; p++)
00374             {
00375                 X [Li [p]] -= Lx [p] * X [j] ;
00376             }
00377         }
00378     }
00379
00380
00381 /* ===== */
00382 /* === ldl_dsolve: solve Dx=b ===== */
00383 /* ===== */
00384
00385 void LDL_dsolve
00386 (
00387     LDL_int n,          /* D is n-by-n, where n >= 0 */
00388     abip_float X [ ],   /* size n. right-hand-side on input, soln. on output */
00389     abip_float D [ ]    /* input of size n, not modified */
00390 )
00391 {
00392     LDL_int j ;
00393     for (j = 0 ; j < n - 7 ; ++j)
00394     {
00395         X [j] /= D [j] ; ++j;
00396         X [j] /= D [j] ; ++j;
00397         X [j] /= D [j] ; ++j;
00398         X [j] /= D [j] ; ++j;
00399         X [j] /= D [j] ; ++j;
00400         X [j] /= D [j] ; ++j;
00401         X [j] /= D [j] ; ++j;
00402         X [j] /= D [j] ;
00403     }
00404
00405     if (j < n - 3)
00406     {
00407         X [j] /= D [j] ; ++j;
00408         X [j] /= D [j] ; ++j;
00409         X [j] /= D [j] ; ++j;
00410         X [j] /= D [j] ; ++j;
00411     }
00412
00413     if (j < n - 1)
00414     {
00415         X [j] /= D [j] ; ++j;
00416         X [j] /= D [j] ; ++j;
00417     }
00418
00419     if (j < n)
00420     {
00421         X [j] /= D [j] ;
00422     }
00423
00424 }
00425
00426
00427 /* ===== */
00428 /* === ldl_ltsolve: solve L'x=b ===== */
00429 /* ===== */
00430
00431 void LDL_ltsolve
00432 (
00433     LDL_int n,          /* L is n-by-n, where n >= 0 */
00434     abip_float X [ ],   /* size n. right-hand-side on input, soln. on output */
00435     LDL_int Lp [ ],     /* input of size n+1, not modified */
00436     LDL_int Li [ ],     /* input of size lnz=Lp[n], not modified */
00437     abip_float Lx [ ]   /* input of size lnz=Lp[n], not modified */
00438 )
00439 {
00440     LDL_int j;
00441     LDL_int p;
00442     LDL_int p2;
00443
00444     for (j = n-1 ; j >= 0 ; j--)
00445     {
00446         p2 = Lp [j+1] ;
00447         for (p = Lp [j] ; p < p2 ; p++)
00448         {
00449             X [j] -= Lx [p] * X [Li [p]] ;
00450         }
00451     }
00452 }
00453

```

```

00454
00455 /* ===== */
00456 /* === ldl_perm: permute a vector, x=Pb ===== */
00457 /* ===== */
00458
00459 void LDL_perm
00460 (
00461     LDL_int n,          /* size of X, B, and P */
00462     abip_float X [ ],   /* output of size n. */
00463     abip_float B [ ],   /* input of size n. */
00464     LDL_int P [ ]       /* input permutation array of size n. */
00465 )
00466 {
00467     LDL_int j ;
00468     for (j = 0 ; j < n - 7; ++j)
00469     {
00470         X [j] = B [P [j]] ; ++j;
00471         X [j] = B [P [j]] ; ++j;
00472         X [j] = B [P [j]] ; ++j;
00473         X [j] = B [P [j]] ; ++j;
00474         X [j] = B [P [j]] ; ++j;
00475         X [j] = B [P [j]] ; ++j;
00476         X [j] = B [P [j]] ; ++j;
00477         X [j] = B [P [j]] ;
00478     }
00479
00480     if (j < n - 3)
00481     {
00482         X [j] = B [P [j]] ; ++j;
00483         X [j] = B [P [j]] ; ++j;
00484         X [j] = B [P [j]] ; ++j;
00485         X [j] = B [P [j]] ; ++j;
00486     }
00487
00488     if (j < n - 1)
00489     {
00490         X [j] = B [P [j]] ; ++j;
00491         X [j] = B [P [j]] ; ++j;
00492     }
00493
00494     if (j < n)
00495     {
00496         X [j] = B [P [j]] ;
00497     }
00498
00499 }
00500
00501
00502 /* ===== */
00503 /* === ldl_permt: permute a vector, x=P'b ===== */
00504 /* ===== */
00505
00506 void LDL_permt
00507 (
00508     LDL_int n,          /* size of X, B, and P */
00509     abip_float X [ ],   /* output of size n. */
00510     abip_float B [ ],   /* input of size n. */
00511     LDL_int P [ ]       /* input permutation array of size n. */
00512 )
00513 {
00514     LDL_int j;
00515     for (j = 0 ; j < n - 7; ++j)
00516     {
00517         X [P [j]] = B [j] ; ++j;
00518         X [P [j]] = B [j] ; ++j;
00519         X [P [j]] = B [j] ; ++j;
00520         X [P [j]] = B [j] ; ++j;
00521         X [P [j]] = B [j] ; ++j;
00522         X [P [j]] = B [j] ; ++j;
00523         X [P [j]] = B [j] ; ++j;
00524         X [P [j]] = B [j] ;
00525     }
00526
00527     if (j < n - 3)
00528     {
00529         X [P [j]] = B [j] ; ++j;
00530         X [P [j]] = B [j] ; ++j;
00531         X [P [j]] = B [j] ; ++j;
00532         X [P [j]] = B [j] ; ++j;
00533     }
00534
00535     if (j < n - 1)
00536     {
00537         X [P [j]] = B [j] ; ++j;
00538         X [P [j]] = B [j] ; ++j;
00539     }
00540

```

```

00541         if (j < n)
00542         {
00543             X [P [j]] = B [j] ;
00544         }
00545     }
00546
00547
00548     /* ===== */
00549     /* === ldl_valid_perm: check if a permutation vector is valid ===== */
00550     /* ===== */
00551
00552     /* returns 1 if valid, 0 otherwise */
00553     LDL_int LDL_valid_perm
00554     (
00555         LDL_int n,
00556         LDL_int P [ ],          /* input of size n, a permutation of 0:n-1 */
00557         LDL_int Flag [ ]        /* workspace of size n */
00558     )
00559     {
00560         LDL_int j;
00561         LDL_int k;
00562
00563         if (n < 0 || !Flag)
00564         {
00565             return (0) ;          /* n must be >= 0, and Flag must be present */
00566         }
00567
00568         if (!P)
00569         {
00570             return (1) ;          /* If ABIP_NULL, P is assumed to be the identity perm. */
00571         }
00572
00573         for (j = 0 ; j < n ; j++)
00574         {
00575             Flag [j] = 0 ;        /* clear the Flag array */
00576         }
00577
00578         for (k = 0 ; k < n ; k++)
00579         {
00580             j = P [k] ;
00581
00582             if (j < 0 || j >= n || Flag [j] != 0)
00583             {
00584                 return (0) ;      /* P is not valid */
00585             }
00586             Flag [j] = 1 ;
00587         }
00588         return (1) ;              /* P is valid */
00589     }
00590
00591
00592     /* ===== */
00593     /* === ldl_valid_matrix: check if a sparse matrix is valid ===== */
00594     /* ===== */
00595
00596     /* This routine checks to see if a sparse matrix A is valid for input to
00597     * ldl_symbolic and ldl_numeric. It returns 1 if the matrix is valid, 0
00598     * otherwise. A is in sparse column form. The numerical values in column j
00599     * are stored in Ax [Ap [j] ... Ap [j+1]-1], with row indices in
00600     * Ai [Ap [j] ... Ap [j+1]-1]. The Ax array is not checked.
00601     */
00602
00603     LDL_int LDL_valid_matrix
00604     (
00605         LDL_int n,
00606         LDL_int Ap [ ],
00607         LDL_int Ai [ ]
00608     )
00609     {
00610         LDL_int j;
00611         LDL_int p;
00612
00613         if (n < 0 || !Ap || !Ai || Ap [0] != 0)
00614         {
00615             return (0) ;          /* n must be >= 0, and Ap and Ai must be present */
00616         }
00617
00618         for (j = 0 ; j < n ; j++)
00619         {
00620             if (Ap [j] > Ap [j+1])
00621             {
00622                 return (0) ;      /* Ap must be monotonically nondecreasing */
00623             }
00624         }
00625
00626         for (p = 0 ; p < Ap [n] ; p++)
00627         {

```

```

00628         if (Ai [p] < 0 || Ai [p] >= n)
00629         {
00630             return (0) ;      /* row indices must be in the range 0 to n-1 */
00631         }
00632     }
00633
00634     return (1) ;      /* matrix is valid */
00635 }

```

5.37 external/ldl/ldl.h File Reference

```
#include "SuiteSparse_config.h"
```

Macros

- `#define LDL_int int`
- `#define LDL_ID "%d"`
- `#define LDL_symbolic ldl_symbolic`
- `#define LDL_numeric ldl_numeric`
- `#define LDL_ksolve ldl_ksolve`
- `#define LDL_dsolve ldl_dsolve`
- `#define LDL_itsolve ldl_itsolve`
- `#define LDL_perm ldl_perm`
- `#define LDL_permt ldl_permt`
- `#define LDL_valid_perm ldl_valid_perm`
- `#define LDL_valid_matrix ldl_valid_matrix`
- `#define LDL_DATE "May 4, 2016"`
- `#define LDL_VERSION_CODE(main, sub) ((main) * 1000 + (sub))`
- `#define LDL_MAIN_VERSION 2`
- `#define LDL_SUB_VERSION 2`
- `#define LDL_SUBSUB_VERSION 6`
- `#define LDL_VERSION LDL_VERSION_CODE(LDL_MAIN_VERSION, LDL_SUB_VERSION)`

Functions

- `void ldl_symbolic (int n, int Ap[], int Ai[], int Lp[], int Parent[], int Lnz[], int Flag[], int P[], int Pinv[])`
- `int ldl_numeric (int n, int Ap[], int Ai[], abip_float Ax[], int Lp[], int Parent[], int Lnz[], int Li[], abip_float Lx[], abip_float D[], abip_float Y[], int Pattern[], int Flag[], int P[], int Pinv[])`
- `void ldl_ksolve (int n, abip_float X[], int Lp[], int Li[], abip_float Lx[])`
- `void ldl_dsolve (int n, abip_float X[], abip_float D[])`
- `void ldl_itsolve (int n, abip_float X[], int Lp[], int Li[], abip_float Lx[])`
- `void ldl_perm (int n, abip_float X[], abip_float B[], int P[])`
- `void ldl_permt (int n, abip_float X[], abip_float B[], int P[])`
- `int ldl_valid_perm (int n, int P[], int Flag[])`
- `int ldl_valid_matrix (int n, int Ap[], int Ai[])`
- `void ldl_l_symbolic (SuiteSparse_long n, SuiteSparse_long Ap[], SuiteSparse_long Ai[], SuiteSparse_long Lp[], SuiteSparse_long Parent[], SuiteSparse_long Lnz[], SuiteSparse_long Flag[], SuiteSparse_long P[], SuiteSparse_long Pinv[])`
- `SuiteSparse_long ldl_l_numeric (SuiteSparse_long n, SuiteSparse_long Ap[], SuiteSparse_long Ai[], abip_float Ax[], SuiteSparse_long Lp[], SuiteSparse_long Parent[], SuiteSparse_long Lnz[], SuiteSparse_long Li[], abip_float Lx[], abip_float D[], abip_float Y[], SuiteSparse_long Pattern[], SuiteSparse_long Flag[], SuiteSparse_long P[], SuiteSparse_long Pinv[])`

- void [ldl_l_solve](#) ([SuiteSparse_long](#) n, [abip_float](#) X[], [SuiteSparse_long](#) Lp[], [SuiteSparse_long](#) Li[], [abip_float](#) Lx[])
- void [ldl_l_dsolve](#) ([SuiteSparse_long](#) n, [abip_float](#) X[], [abip_float](#) D[])
- void [ldl_l_itsolve](#) ([SuiteSparse_long](#) n, [abip_float](#) X[], [SuiteSparse_long](#) Lp[], [SuiteSparse_long](#) Li[], [abip_float](#) Lx[])
- void [ldl_l_perm](#) ([SuiteSparse_long](#) n, [abip_float](#) X[], [abip_float](#) B[], [SuiteSparse_long](#) P[])
- void [ldl_l_permt](#) ([SuiteSparse_long](#) n, [abip_float](#) X[], [abip_float](#) B[], [SuiteSparse_long](#) P[])
- [SuiteSparse_long](#) [ldl_l_valid_perm](#) ([SuiteSparse_long](#) n, [SuiteSparse_long](#) P[], [SuiteSparse_long](#) Flag[])
- [SuiteSparse_long](#) [ldl_l_valid_matrix](#) ([SuiteSparse_long](#) n, [SuiteSparse_long](#) Ap[], [SuiteSparse_long](#) Ai[])

5.37.1 Macro Definition Documentation

5.37.1.1 LDL_DATE

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

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

5.37.1.2 LDL_dsolve

```
#define LDL_dsolve ldl\_dsolve
```

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

5.37.1.3 LDL_ID

```
#define LDL_ID "%d"
```

Definition at line [27](#) of file [ldl.h](#).

5.37.1.4 LDL_int

```
#define LDL_int int
```

Definition at line [26](#) of file [ldl.h](#).

5.37.1.5 LDL_Isolve

```
#define LDL_Isolve ldl\_Isolve
```

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

5.37.1.6 LDL_ltsolve

```
#define LDL_ltsolve ldl\_ltsolve
```

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

5.37.1.7 LDL_MAIN_VERSION

```
#define LDL_MAIN_VERSION 2
```

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

5.37.1.8 LDL_numeric

```
#define LDL_numeric ldl\_numeric
```

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

5.37.1.9 LDL_perm

```
#define LDL_perm ldl\_perm
```

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

5.37.1.10 LDL_permt

```
#define LDL_permt ldl\_permt
```

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

5.37.1.11 LDL_SUB_VERSION

```
#define LDL_SUB_VERSION 2
```

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

5.37.1.12 LDL_SUBSUB_VERSION

```
#define LDL_SUBSUB_VERSION 6
```

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

5.37.1.13 LDL_symbolic

```
#define LDL_symbolic ldl_symbolic
```

Definition at line 29 of file [ldl.h](#).

5.37.1.14 LDL_valid_matrix

```
#define LDL_valid_matrix ldl_valid_matrix
```

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

5.37.1.15 LDL_valid_perm

```
#define LDL_valid_perm ldl_valid_perm
```

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

5.37.1.16 LDL_VERSION

```
#define LDL_VERSION LDL_VERSION_CODE(LDL_MAIN_VERSION, LDL_SUB_VERSION)
```

Definition at line 111 of file [ldl.h](#).

5.37.1.17 LDL_VERSION_CODE

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

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

5.37.2 Function Documentation

5.37.2.1 ldl_dsolve()

```
void ldl_dsolve (  
    int n,  
    abip_float X[],  
    abip_float D[] )
```

5.37.2.2 ldl_l_dsolve()

```
void ldl_l_dsolve (  
    SuiteSparse_long n,  
    abip_float X[],  
    abip_float D[] )
```

5.37.2.3 ldl_l_lsolve()

```
void ldl_l_lsolve (  
    SuiteSparse_long n,  
    abip_float X[],  
    SuiteSparse_long Lp[],  
    SuiteSparse_long Li[],  
    abip_float Lx[] )
```

5.37.2.4 ldl_l_ltsolve()

```
void ldl_l_ltsolve (  
    SuiteSparse_long n,  
    abip_float X[],  
    SuiteSparse_long Lp[],  
    SuiteSparse_long Li[],  
    abip_float Lx[] )
```

5.37.2.5 `ldl_l_numeric()`

```
SuiteSparse_long ldl_l_numeric (
    SuiteSparse_long n,
    SuiteSparse_long Ap[],
    SuiteSparse_long Ai[],
    abip_float Ax[],
    SuiteSparse_long Lp[],
    SuiteSparse_long Parent[],
    SuiteSparse_long Lnz[],
    SuiteSparse_long Li[],
    abip_float Lx[],
    abip_float D[],
    abip_float Y[],
    SuiteSparse_long Pattern[],
    SuiteSparse_long Flag[],
    SuiteSparse_long P[],
    SuiteSparse_long Pinv[] )
```

5.37.2.6 `ldl_l_perm()`

```
void ldl_l_perm (
    SuiteSparse_long n,
    abip_float X[],
    abip_float B[],
    SuiteSparse_long P[] )
```

5.37.2.7 `ldl_l_permt()`

```
void ldl_l_permt (
    SuiteSparse_long n,
    abip_float X[],
    abip_float B[],
    SuiteSparse_long P[] )
```

5.37.2.8 `ldl_l_symbolic()`

```
void ldl_l_symbolic (
    SuiteSparse_long n,
    SuiteSparse_long Ap[],
    SuiteSparse_long Ai[],
    SuiteSparse_long Lp[],
    SuiteSparse_long Parent[],
    SuiteSparse_long Lnz[],
    SuiteSparse_long Flag[],
    SuiteSparse_long P[],
    SuiteSparse_long Pinv[] )
```

5.37.2.9 ldl_l_valid_matrix()

```
SuiteSparse_long ldl_l_valid_matrix (
    SuiteSparse_long n,
    SuiteSparse_long Ap[],
    SuiteSparse_long Ai[] )
```

5.37.2.10 ldl_l_valid_perm()

```
SuiteSparse_long ldl_l_valid_perm (
    SuiteSparse_long n,
    SuiteSparse_long P[],
    SuiteSparse_long Flag[] )
```

5.37.2.11 ldl_solve()

```
void ldl_solve (
    int n,
    abip_float X[],
    int Lp[],
    int Li[],
    abip_float Lx[] )
```

5.37.2.12 ldl_ltsolve()

```
void ldl_ltsolve (
    int n,
    abip_float X[],
    int Lp[],
    int Li[],
    abip_float Lx[] )
```

5.37.2.13 ldl_numeric()

```
int ldl_numeric (
    int n,
    int Ap[],
    int Ai[],
    abip_float Ax[],
    int Lp[],
    int Parent[],
    int Lnz[],
    int Li[],
    abip_float Lx[],
    abip_float D[],
    abip_float Y[],
    int Pattern[],
    int Flag[],
    int P[],
    int Pinv[] )
```

5.37.2.14 ldl_perm()

```
void ldl_perm (
    int n,
    abip_float X[],
    abip_float B[],
    int P[] )
```

5.37.2.15 ldl_permt()

```
void ldl_permt (
    int n,
    abip_float X[],
    abip_float B[],
    int P[] )
```

5.37.2.16 ldl_symbolic()

```
void ldl_symbolic (
    int n,
    int Ap[],
    int Ai[],
    int Lp[],
    int Parent[],
    int Lnz[],
    int Flag[],
    int P[],
    int Pinv[] )
```

5.37.2.17 ldl_valid_matrix()

```
int ldl_valid_matrix (
    int n,
    int Ap[ ],
    int Ai[ ] )
```

5.37.2.18 ldl_valid_perm()

```
int ldl_valid_perm (
    int n,
    int P[ ],
    int Flag[ ] )
```

5.38 ldl.h

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

```
00001 /* ===== */
00002 /* == ldl.h: include file for the LDL package ===== */
00003 /* ===== */
00004
00005 /* Copyright (c) Timothy A Davis, http://www.suitesparse.com.
00006  * All Rights Reserved. See LDL/Doc/License.txt for the License.
00007  */
00008
00009 #include "SuiteSparse_config.h"
00010
00011 #ifndef DLONG
00012 #define LDL_int SuiteSparse_long
00013 #define LDL_ID SuiteSparse_long_id
00014
00015 #define LDL_symbolic ldl_l_symbolic
00016 #define LDL_numeric ldl_l_numeric
00017 #define LDL_lsolve ldl_l_lsolve
00018 #define LDL_dsolve ldl_l_dsolve
00019 #define LDL_ltsolve ldl_l_ltsolve
00020 #define LDL_perm ldl_l_perm
00021 #define LDL_permt ldl_l_permt
00022 #define LDL_valid_perm ldl_l_valid_perm
00023 #define LDL_valid_matrix ldl_l_valid_matrix
00024
00025 #else
00026 #define LDL_int int
00027 #define LDL_ID "%d"
00028
00029 #define LDL_symbolic ldl_symbolic
00030 #define LDL_numeric ldl_numeric
00031 #define LDL_lsolve ldl_lsolve
00032 #define LDL_dsolve ldl_dsolve
00033 #define LDL_ltsolve ldl_ltsolve
00034 #define LDL_perm ldl_perm
00035 #define LDL_permt ldl_permt
00036 #define LDL_valid_perm ldl_valid_perm
00037 #define LDL_valid_matrix ldl_valid_matrix
00038
00039 #endif
00040
00041 /* ===== */
00042 /* == int version ===== */
00043 /* ===== */
00044
00045 void ldl_symbolic (int n, int Ap[ ], int Ai[ ], int Lp[ ],
00046     int Parent[ ], int Lnz[ ], int Flag[ ], int P[ ], int Pinv[ ]) ;
00047
00048 int ldl_numeric (int n, int Ap[ ], int Ai[ ], abip_float Ax[ ],
00049     int Lp[ ], int Parent[ ], int Lnz[ ], int Li[ ], abip_float Lx[ ],
00050     abip_float D[ ], abip_float Y[ ], int Pattern[ ], int Flag[ ],
00051     int P[ ], int Pinv[ ]) ;
00052
```

```

00053 void ldl_lsolve (int n, abip_float X [ ], int Lp [ ], int Li [ ],
00054     abip_float Lx [ ]) ;
00055
00056 void ldl_dsolve (int n, abip_float X [ ], abip_float D [ ]) ;
00057
00058 void ldl_ltsolve (int n, abip_float X [ ], int Lp [ ], int Li [ ],
00059     abip_float Lx [ ]) ;
00060
00061 void ldl_perm (int n, abip_float X [ ], abip_float B [ ], int P [ ]) ;
00062 void ldl_permt (int n, abip_float X [ ], abip_float B [ ], int P [ ]) ;
00063
00064 int ldl_valid_perm (int n, int P [ ], int Flag [ ]) ;
00065 int ldl_valid_matrix (int n, int Ap [ ], int Ai [ ]) ;
00066
00067 /* ===== */
00068 /* === long version ===== */
00069 /* ===== */
00070
00071 void ldl_l_symbolic (SuiteSparse_long n, SuiteSparse_long Ap [ ],
00072     SuiteSparse_long Ai [ ], SuiteSparse_long Lp [ ],
00073     SuiteSparse_long Parent [ ], SuiteSparse_long Lnz [ ],
00074     SuiteSparse_long Flag [ ], SuiteSparse_long P [ ],
00075     SuiteSparse_long Pinv [ ]) ;
00076
00077 SuiteSparse_long ldl_l_numeric (SuiteSparse_long n, SuiteSparse_long Ap [ ],
00078     SuiteSparse_long Ai [ ], abip_float Ax [ ], SuiteSparse_long Lp [ ],
00079     SuiteSparse_long Parent [ ], SuiteSparse_long Lnz [ ],
00080     SuiteSparse_long Li [ ], abip_float Lx [ ], abip_float D [ ], abip_float Y [ ],
00081     SuiteSparse_long Pattern [ ], SuiteSparse_long Flag [ ],
00082     SuiteSparse_long P [ ], SuiteSparse_long Pinv [ ]) ;
00083
00084 void ldl_l_lsolve (SuiteSparse_long n, abip_float X [ ], SuiteSparse_long Lp [ ],
00085     SuiteSparse_long Li [ ], abip_float Lx [ ]) ;
00086
00087 void ldl_l_dsolve (SuiteSparse_long n, abip_float X [ ], abip_float D [ ]) ;
00088
00089 void ldl_l_ltsolve (SuiteSparse_long n, abip_float X [ ], SuiteSparse_long Lp [ ],
00090     SuiteSparse_long Li [ ], abip_float Lx [ ]) ;
00091
00092 void ldl_l_perm (SuiteSparse_long n, abip_float X [ ], abip_float B [ ],
00093     SuiteSparse_long P [ ]) ;
00094 void ldl_l_permt (SuiteSparse_long n, abip_float X [ ], abip_float B [ ],
00095     SuiteSparse_long P [ ]) ;
00096
00097 SuiteSparse_long ldl_l_valid_perm (SuiteSparse_long n, SuiteSparse_long P [ ],
00098     SuiteSparse_long Flag [ ]) ;
00099 SuiteSparse_long ldl_l_valid_matrix (SuiteSparse_long n,
00100     SuiteSparse_long Ap [ ], SuiteSparse_long Ai [ ]) ;
00101
00102 /* ===== */
00103 /* === LDL version ===== */
00104 /* ===== */
00105
00106 #define LDL_DATE "May 4, 2016"
00107 #define LDL_VERSION_CODE(main,sub) ((main) * 1000 + (sub))
00108 #define LDL_MAIN_VERSION 2
00109 #define LDL_SUB_VERSION 2
00110 #define LDL_SUBSUB_VERSION 6
00111 #define LDL_VERSION LDL_VERSION_CODE(LDL_MAIN_VERSION, LDL_SUB_VERSION)
00112

```

5.39 external/README.md File Reference

5.40 external/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](#)

5.40.1 Function Documentation

5.40.1.1 SuiteSparse_calloc()

```
void * SuiteSparse_calloc (
    size_t nitems,
    size_t size_of_item )
```

Definition at line 211 of file [SuiteSparse_config.c](#).

5.40.1.2 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](#).

5.40.1.3 SuiteSparse_finish()

```
void SuiteSparse_finish (
    void )
```

Definition at line 173 of file [SuiteSparse_config.c](#).

5.40.1.4 SuiteSparse_free()

```
void * SuiteSparse_free (
    void * p )
```

Definition at line 311 of file [SuiteSparse_config.c](#).

5.40.1.5 SuiteSparse_hypot()

```
abip_float SuiteSparse_hypot (
    abip_float x,
    abip_float y )
```

Definition at line 464 of file [SuiteSparse_config.c](#).

5.40.1.6 SuiteSparse_malloc()

```
void * SuiteSparse_malloc (
    size_t nitems,
    size_t size_of_item )
```

Definition at line 182 of file [SuiteSparse_config.c](#).

5.40.1.7 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](#).

5.40.1.8 SuiteSparse_start()

```
void SuiteSparse_start (
    void )
```

Definition at line 109 of file [SuiteSparse_config.c](#).

5.40.1.9 SuiteSparse_tic()

```
void SuiteSparse_tic (
    abip_float tic[2] )
```

Definition at line 373 of file [SuiteSparse_config.c](#).

5.40.1.10 SuiteSparse_time()

```
abip_float SuiteSparse_time (
    void )
```

Definition at line 414 of file [SuiteSparse_config.c](#).

5.40.1.11 SuiteSparse_toc()

```
abip_float SuiteSparse_toc (
    abip_float tic[2] )
```

Definition at line 397 of file [SuiteSparse_config.c](#).

5.40.1.12 SuiteSparse_version()

```
int SuiteSparse_version (
    int version[3] )
```

Definition at line 429 of file [SuiteSparse_config.c](#).

5.40.2 Variable Documentation

5.40.2.1 SuiteSparse_config

```
struct SuiteSparse_config_struct SuiteSparse_config
```

Definition at line 53 of file [SuiteSparse_config.c](#).

5.41 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         #ifdef MATLAB_MEX_FILE
00058             /* MATLAB mexFunction: */
00059             mxMalloc, mxCalloc, mxRealloc, mxFree,
00060         #else
00061             /* standard ANSI C: */
00062             malloc, calloc, realloc, free,
00063         #endif
00064     #endif
00065     #else
00066         /* no memory manager defined; you must define one at run-time: */
00067         ABIP_NULL, ABIP_NULL, ABIP_NULL, ABIP_NULL,
00068     #endif
00069
00070     /* printf function */
00071     #ifndef NPRINT
00072         #ifdef MATLAB_MEX_FILE
00073             /* MATLAB mexFunction: */
00074             mexPrintf,
00075         #else
00076             /* standard ANSI C: */
00077             printf,
00078         #endif
00079     #endif
00080 }
00081
00082

```

```

00083         #else
00084             /* printf is disabled */
00085             ABIP_NULL,
00086         #endif
00087
00088         SuiteSparse_hypot,
00089         SuiteSparse_divcomplex
00090     };
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     #ifdef MATLAB_MEX_FILE
00114         /* MATLAB mexFunction: */
00115         SuiteSparse_config.malloc_func = mxMalloc ;
00116         SuiteSparse_config.calloc_func = mxCalloc ;
00117         SuiteSparse_config.realloc_func = mxRealloc ;
00118         SuiteSparse_config.free_func = mxFree ;
00119     #else
00120         /* standard ANSI C: */
00121         SuiteSparse_config.malloc_func = malloc ;
00122         SuiteSparse_config.calloc_func = calloc ;
00123         SuiteSparse_config.realloc_func = realloc ;
00124         SuiteSparse_config.free_func = free ;
00125     #endif
00126     #else
00127         /* no memory manager defined; you must define one after calling
00128            SuiteSparse_start */
00129         SuiteSparse_config.malloc_func = ABIP_NULL ;
00130         SuiteSparse_config.calloc_func = ABIP_NULL ;
00131         SuiteSparse_config.realloc_func = ABIP_NULL ;
00132         SuiteSparse_config.free_func = ABIP_NULL ;
00133     #endif
00134
00135     /* printf function */
00136     #ifndef NPRINT
00137     #ifdef MATLAB_MEX_FILE
00138         /* MATLAB mexFunction: */
00139         SuiteSparse_config.printf_func = mexPrintf ;
00140     #else
00141         /* standard ANSI C: */
00142         SuiteSparse_config.printf_func = printf ;
00143     #endif
00144     #else
00145         /* printf is disabled */
00146         SuiteSparse_config.printf_func = ABIP_NULL ;
00147     #endif
00148
00149     /* math functions */
00150     SuiteSparse_config.hypot_func = SuiteSparse_hypot ;
00151     SuiteSparse_config.divcomplex_func = SuiteSparse_divcomplex ;
00152 }
00153
00154 /* ----- */
00155 /* SuiteSparse_finish */
00156 /* ----- */
00157
00158 /* This currently does nothing, but in the future, applications should call
00159    SuiteSparse_start before calling any SuiteSparse function, and then
00160    SuiteSparse_finish after calling the last SuiteSparse function, just before
00161    exiting. In a multithreaded application, only a single thread should call
00162    this function.
00163 */

```

```

00169     Future releases of SuiteSparse might use this function for any
00170     SuiteSparse-wide cleanup operations or finalization of statistics.
00171 */
00172
00173 void SuiteSparse_finish ( void )
00174 {
00175     /* do nothing */ ;
00176 }
00177
00178 /* ----- */
00179 /* SuiteSparse_malloc: malloc wrapper */
00180 /* ----- */
00181
00182 void *SuiteSparse_malloc      /* pointer to allocated block of memory */
00183 (
00184     size_t nitems,           /* number of items to malloc */
00185     size_t size_of_item     /* sizeof each item */
00186 )
00187 {
00188     void *p ;
00189     size_t size ;
00190     if (nitems < 1) nitems = 1 ;
00191     if (size_of_item < 1) size_of_item = 1 ;
00192     size = nitems * size_of_item ;
00193
00194     if (size != ((abip_float) nitems) * size_of_item)
00195     {
00196         /* size_t overflow */
00197         p = ABIP_NULL ;
00198     }
00199     else
00200     {
00201         p = (void *) (SuiteSparse_config.malloc_func) (size) ;
00202     }
00203     return (p) ;
00204 }
00205
00206 /* ----- */
00207 /* SuiteSparse_calloc: calloc wrapper */
00208 /* ----- */
00209
00210 void *SuiteSparse_calloc      /* pointer to allocated block of memory */
00211 (
00212     size_t nitems,           /* number of items to calloc */
00213     size_t size_of_item     /* sizeof each item */
00214 )
00215 {
00216     void *p ;
00217     size_t size ;
00218     if (nitems < 1) nitems = 1 ;
00219     if (size_of_item < 1) size_of_item = 1 ;
00220     size = nitems * size_of_item ;
00221
00222     if (size != ((abip_float) nitems) * size_of_item)
00223     {
00224         /* size_t overflow */
00225         p = ABIP_NULL ;
00226     }
00227     else
00228     {
00229         p = (void *) (SuiteSparse_config.calloc_func) (nitems, size_of_item) ;
00230     }
00231     return (p) ;
00232 }
00233
00234 /* ----- */
00235 /* SuiteSparse_realloc: realloc wrapper */
00236 /* ----- */
00237
00238 /* If p is non-NULL on input, it points to a previously allocated object of
00239 size nitems_old * size_of_item. The object is reallocated to be of size
00240 nitems_new * size_of_item. If p is NULL on input, then a new object of that
00241 size is allocated. On success, a pointer to the new object is returned,
00242 and ok is returned as 1. If the allocation fails, ok is set to 0 and a
00243 pointer to the old (unmodified) object is returned.
00244 */
00245
00246 void *SuiteSparse_realloc      /* pointer to reallocated block of memory, or to original block if the
realloc failed. */
00247 (
00248     size_t nitems_new,       /* new number of items in the object */
00249     size_t nitems_old,       /* old number of items in the object */
00250     size_t size_of_item,     /* sizeof each item */
00251     void *p,                 /* old object to reallocate */
00252     int *ok                  /* 1 if successful, 0 otherwise */
00253 )
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 }
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 /* SuiteSparse_tic: return current wall clock time */
00325 /* ----- */
00326
00327 /* Returns the number of seconds (tic [0]) and nanoseconds (tic [1]) since some
00328 * unspecified but fixed time in the past. If no timer is installed, zero is
00329 * returned. A scalar abip_float precision value for 'tic' could be used, but this
00330 * might cause loss of precision because clock_gettime returns the time from
00331 * some distant time in the past. Thus, an array of size 2 is used.
00332 *
00333 * The timer is enabled by default. To disable the timer, compile with
00334 * -DNTIMER. If enabled on a POSIX C 1993 system, the timer requires linking
00335 * with the -lrt library.
00336 *
00337 * example:
00338 *
00339 *     abip_float tic [2], r, s, t ;
00340

```

```

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 #include <time.h>
00357
00358 void SuiteSparse_tic
00359 (
00360     abip_float tic [2]      /* output, contents undefined on input */
00361 )
00362 {
00363     /* POSIX C 1993 timer, requires -librt */
00364     struct timespec t ;
00365     clock_gettime (CLOCK_MONOTONIC, &t) ;
00366     tic [0] = (abip_float) (t.tv_sec) ;
00367     tic [1] = (abip_float) (t.tv_nsec) ;
00368 }
00369
00370 #else
00371 void SuiteSparse_tic
00372 (
00373     abip_float tic [2]      /* output, contents undefined on input */
00374 )
00375 {
00376     /* no timer installed */
00377     tic [0] = 0 ;
00378     tic [1] = 0 ;
00379 }
00380 #endif
00381
00382 /* ----- */
00383 /* SuiteSparse_toc: return time since last tic */
00384 /* ----- */
00385
00386 /* Assuming SuiteSparse_tic is accurate to the nanosecond, this function is
00387 * accurate down to the nanosecond for 2^53 nanoseconds since the last call to
00388 * SuiteSparse_tic, which is sufficient for SuiteSparse (about 104 days). If
00389 * additional accuracy is required, the caller can use two calls to
00390 * SuiteSparse_tic and do the calculations differently.
00391 */
00392
00393 abip_float SuiteSparse_toc /* returns time in seconds since last tic */
00394 (
00395     abip_float tic [2] /* input, not modified from last call to SuiteSparse_tic */
00396 )
00397 {
00398     abip_float toc [2] ;
00399     SuiteSparse_tic (toc) ;
00400     return ((toc [0] - tic [0]) + 1e-9 * (toc [1] - tic [1])) ;
00401 }
00402
00403 /* ----- */
00404 /* SuiteSparse_time: return current wallclock time in seconds */
00405 /* ----- */
00406
00407 /* This function might not be accurate down to the nanosecond. */
00408
00409 abip_float SuiteSparse_time /* returns current wall clock time in seconds */
00410 (
00411     void
00412 )
00413 {
00414     abip_float toc [2] ;
00415     SuiteSparse_tic (toc) ;
00416     return (toc [0] + 1e-9 * toc [1]) ;
00417 }
00418
00419 /* ----- */
00420 /* SuiteSparse_version: return the current version of SuiteSparse */
00421 /* ----- */

```

```

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 *     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 }

```

5.42 external/SuiteSparse_config.h File Reference

```

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

```

Data Structures

- 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](#)

5.42.1 Macro Definition Documentation

5.42.1.1 SUITESPARSE_DATE

```
#define SUITESPARSE_DATE "Dec 28, 2017"
```

Definition at line 237 of file [SuiteSparse_config.h](#).

5.42.1.2 SUITESPARSE_HAS_VERSION_FUNCTION

```
#define SUITESPARSE_HAS_VERSION_FUNCTION
```

Definition at line 235 of file [SuiteSparse_config.h](#).

5.42.1.3 SuiteSparse_long

```
#define SuiteSparse_long long
```

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

5.42.1.4 SuiteSparse_long_id

```
#define SuiteSparse_long_id "%" SuiteSparse_long_idd
```

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

5.42.1.5 SuiteSparse_long_idd

```
#define SuiteSparse_long_idd "ld"
```

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

5.42.1.6 SuiteSparse_long_max

```
#define SuiteSparse_long_max LONG_MAX
```

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

5.42.1.7 SUITESPARSE_MAIN_VERSION

```
#define SUITESPARSE_MAIN_VERSION 5
```

Definition at line 239 of file [SuiteSparse_config.h](#).

5.42.1.8 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](#).

5.42.1.9 SUITESPARSE_SUB_VERSION

```
#define SUITESPARSE_SUB_VERSION 1
```

Definition at line 240 of file [SuiteSparse_config.h](#).

5.42.1.10 SUITESPARSE_SUBSUB_VERSION

```
#define SUITESPARSE_SUBSUB_VERSION 2
```

Definition at line 241 of file [SuiteSparse_config.h](#).

5.42.1.11 SUITESPARSE_VER_CODE

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

Definition at line 238 of file [SuiteSparse_config.h](#).

5.42.1.12 SUITESPARSE_VERSION

```
#define SUITESPARSE_VERSION SUITESPARSE_VER_CODE(SUITESPARSE_MAIN_VERSION, SUITESPARSE_SUB_VERSION)
```

Definition at line 242 of file [SuiteSparse_config.h](#).

5.42.2 Function Documentation

5.42.2.1 SuiteSparse_calloc()

```
void * SuiteSparse_calloc (  
    size_t nitems,  
    size_t size_of_item )
```

Definition at line 211 of file [SuiteSparse_config.c](#).

5.42.2.2 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](#).

5.42.2.3 SuiteSparse_finish()

```
void SuiteSparse_finish (
    void )
```

Definition at line 173 of file [SuiteSparse_config.c](#).

5.42.2.4 SuiteSparse_free()

```
void * SuiteSparse_free (
    void * p )
```

Definition at line 311 of file [SuiteSparse_config.c](#).

5.42.2.5 SuiteSparse_hypot()

```
abip_float SuiteSparse_hypot (
    abip_float x,
    abip_float y )
```

Definition at line 464 of file [SuiteSparse_config.c](#).

5.42.2.6 SuiteSparse_malloc()

```
void * SuiteSparse_malloc (
    size_t nitems,
    size_t size_of_item )
```

Definition at line 182 of file [SuiteSparse_config.c](#).

5.42.2.7 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](#).

5.42.2.8 SuiteSparse_start()

```
void SuiteSparse_start (
    void )
```

Definition at line 109 of file [SuiteSparse_config.c](#).

5.42.2.9 SuiteSparse_tic()

```
void SuiteSparse_tic (
    abip_float tic[2] )
```

Definition at line 373 of file [SuiteSparse_config.c](#).

5.42.2.10 SuiteSparse_time()

```
abip_float SuiteSparse_time (
    void )
```

Definition at line 414 of file [SuiteSparse_config.c](#).

5.42.2.11 SuiteSparse_toc()

```
abip_float SuiteSparse_toc (
    abip_float tic[2] )
```

Definition at line 397 of file [SuiteSparse_config.c](#).

5.42.2.12 SuiteSparse_version()

```
int SuiteSparse_version (
    int version[3] )
```

Definition at line 429 of file [SuiteSparse_config.c](#).

5.42.3 Variable Documentation

5.42.3.1 SuiteSparse_config

struct `SuiteSparse_config_struct` SuiteSparse_config [extern]

Definition at line 53 of file `SuiteSparse_config.c`.

5.43 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 extern struct SuiteSparse_config_struct SuiteSparse_config ;
00099
00100 void SuiteSparse_start ( void ) ; /* called to start SuiteSparse */
00101
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 CCOLAMD, 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 * CCOLAMD 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_GPURuntime 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

```

5.44 include/abip.h File Reference

```

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

```

Data Structures

- struct [ABIP_PROBLEM_DATA](#)
- struct [ABIP_SETTINGS](#)
- struct [ABIP_SOL_VARS](#)
- struct [ABIP_INFO](#)
- struct [ABIP_SCALING](#)
- 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_SCALING](#) [ABIPScaling](#)
- typedef struct [ABIP_WORK](#) [ABIPWork](#)
- typedef struct [ABIP_ADAPTIVE_WORK](#) [ABIPAdaptWork](#)
- typedef struct [ABIP_RESIDUALS](#) [ABIPResiduals](#)

Functions

- [ABIPWork](#) *[ABIP](#)() [init](#) (const [ABIPData](#) *d, [ABIPInfo](#) *info)
- [abip_int](#) [ABIP](#)() [solve](#) ([ABIPWork](#) *w, const [ABIPData](#) *d, [ABIPSolution](#) *sol, [ABIPInfo](#) *info)
detailed update rule of ABIP
- void [ABIP](#)() [finish](#) ([ABIPWork](#) *w)
- [abip_int](#) [ABIP](#)() [main](#) (const [ABIPData](#) *d, [ABIPSolution](#) *sol, [ABIPInfo](#) *info)
the main function
- const char *[ABIP](#)() [version](#) (void)
return the abip version

5.44.1 Typedef Documentation

5.44.1.1 ABIPAdaptWork

```
typedef struct ABIP_ADAPTIVE_WORK ABIPAdaptWork
```

Definition at line 20 of file [abip.h](#).

5.44.1.2 ABIPData

```
typedef struct ABIP_PROBLEM_DATA ABIPData
```

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

5.44.1.3 ABIPInfo

```
typedef struct ABIP_INFO ABIPInfo
```

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

5.44.1.4 ABIPLinSysWork

```
typedef struct ABIP_LIN_SYS_WORK ABIPLinSysWork
```

Definition at line 12 of file [abip.h](#).

5.44.1.5 ABIPMatrix

```
typedef struct ABIP_A_DATA_MATRIX ABIPMatrix
```

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

5.44.1.6 ABIPResiduals

```
typedef struct ABIP_RESIDUALS ABIPResiduals
```

Definition at line 21 of file [abip.h](#).

5.44.1.7 ABIPScaling

```
typedef struct ABIP_SCALING ABIPScaling
```

Definition at line 18 of file [abip.h](#).

5.44.1.8 ABIPSettings

```
typedef struct ABIP_SETTINGS ABIPSettings
```

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

5.44.1.9 ABIPSolution

```
typedef struct ABIP_SOL_VARS ABIPSolution
```

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

5.44.1.10 ABIPWork

```
typedef struct ABIP_WORK ABIPWork
```

Definition at line 19 of file [abip.h](#).

5.44.2 Function Documentation

5.44.2.1 finish()

```
void ABIP() finish (  
    ABIPWork * w )
```

Definition at line 2301 of file [abip.c](#).

5.44.2.2 init()

```
ABIPWork *ABIP() init (
    const ABIPData * d,
    ABIPInfo * info )
```

Definition at line 2341 of file [abip.c](#).

5.44.2.3 main()

```
abip_int ABIP() main (
    const ABIPData * d,
    ABIPSolution * sol,
    ABIPInfo * info )
```

the main function

Definition at line 2393 of file [abip.c](#).

5.44.2.4 solve()

```
abip_int ABIP() solve (
    ABIPWork * w,
    const ABIPData * d,
    ABIPSolution * sol,
    ABIPInfo * info )
```

detailed update rule of ABIP

Definition at line 2056 of file [abip.c](#).

5.44.2.5 version()

```
const char *ABIP() version (
    void )
```

return the abip version

Definition at line 5 of file [abip_version.c](#).

5.45 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
00011 typedef struct ABIP_A_DATA_MATRIX ABIPMatrix;
00012 typedef struct ABIP_LIN_SYS_WORK ABIPLinSysWork;
00013
00014 typedef struct ABIP_PROBLEM_DATA ABIPData;
00015 typedef struct ABIP_SETTINGS ABIPSettings;
00016 typedef struct ABIP_SOL_VARS ABIPSolution;
00017 typedef struct ABIP_INFO ABIPIInfo;
00018 typedef struct ABIP_SCALING ABIPScaling;
00019 typedef struct ABIP_WORK ABIPWork;
00020 typedef struct ABIP_ADAPTIVE_WORK ABIPAdaptWork;
00021 typedef struct ABIP_RESIDUALS ABIPResiduals;
00022
00023 struct ABIP_PROBLEM_DATA
00024 {
00025     abip_int m;
00026     abip_int n;
00027     ABIPMatrix *A;
00028
00029     abip_float *b;
00030     abip_float *c;
00031     abip_float sp;
00032
00033     ABIPSettings *stgs;
00034 };
00035
00036 struct ABIP_SETTINGS
00037 {
00038     abip_int normalize;
00039     abip_int pfeasopt;
00040     abip_float scale;
00041     abip_float rho_y;
00042     abip_float sparsity_ratio;
00043
00044     abip_int max_ipm_iters;
00045     abip_int max_admm_iters;
00046     abip_float max_time;
00047
00048     abip_float eps;
00049     abip_float alpha;
00050     abip_float cg_rate; /* for indirect, tolerance goes down like (1/iter)^cg_rate: 2 */
00051
00052     abip_int adaptive;
00053     abip_float eps_cor;
00054     abip_float eps_pen;
00055
00056     abip_float dynamic_sigma; /* Dynamic strategy A to update the barrier parameter */
00057
00058     abip_float dynamic_x; /* Dynamic strategy B to update the barrier parameter */
00059     abip_float dynamic_eta;
00060
00061     abip_int restart_fre; /* Frequency of performing restart*/
00062     abip_int restart_thresh; /* Threshold of restart */
00063
00064     abip_int verbose; /* boolean, write out progress: 1 */
00065     abip_int warm_start; /* boolean, warm start (put initial guess in ABIPSolution struct): 0 */
00066
00067     abip_int adaptive_lookback;
00068
00069
00070     abip_int origin_rescale; /* boolean, use the origin_rescale or not */
00071     abip_int pc_ruiz_rescale; /* boolean, use the pc_rescale or not */
00072     abip_int qp_rescale; /* boolean, use the qp rescale or not*/
00073     abip_int ruiz_iter; /* int, number of ruiz rescaling */
00074     abip_int hybrid_mu; /* boolean, use the hybrid_mu strategy or not */
00075     abip_float hybrid_thresh; /* float, control when to use the LOQO mu strategy, only
used when hybrid_mu = 1 */
00076     abip_float dynamic_sigma_second; /* float, control the dynamic sigma when the mu strategy
becomes LOQO */
00077     abip_int half_update; /* boolean, use the half update technique or not*/
00078     abip_int avg_criterion; /* boolean, use the avg criterion technique or not*/

```

```

00079 };
00080
00081 struct ABIP_SOL_VARS
00082 {
00083     abip_float *x;
00084     abip_float *y;
00085     abip_float *s;
00086 };
00087
00088 struct ABIP_INFO
00089 {
00090     char status[32];
00091     abip_int status_val;
00092     abip_int ipm_iter;
00093     abip_int admm_iter;
00094
00095     abip_float pobj;
00096     abip_float dobj;
00097     abip_float res_pri;
00098     abip_float res_dual;
00099     abip_float rel_gap;
00100     abip_float res_infeas;
00101     abip_float res_unbdd;
00102
00103     abip_float setup_time;
00104     abip_float solve_time;
00105 };
00106
00107 struct ABIP_SCALING
00108 {
00109     abip_float *D;
00110     abip_float *E;
00111
00112     abip_float mean_norm_row_A;
00113     abip_float mean_norm_col_A;
00114 };
00115
00116 /* main functions: ABIP(init): allocates memory of matrix, e.g., [I A; A^T -I].
00117                    ABIP(solve): can be called many times with different b,c data per init
00118                    call.
00119                    ABIP(finish): cleans up the memory (one per init call) */
00119 ABIPWork *ABIP(init) (const ABIPData *d, ABIPInfo *info);
00120 abip_int ABIP(solve) (ABIPWork *w, const ABIPData *d, ABIPSolution *sol, ABIPInfo *info);
00121 void ABIP(finish) (ABIPWork *w);
00122
00123 abip_int ABIP(main) (const ABIPData *d, ABIPSolution *sol, ABIPInfo *info);
00124 const char *ABIP(version) (void);
00125
00126 struct ABIP_WORK
00127 {
00128     abip_float sigma;
00129     abip_float gamma;
00130     abip_int final_check;
00131     abip_int double_check;
00132
00133     abip_float mu;
00134     abip_float beta;
00135
00136     abip_float *u;
00137     abip_float *v;
00138     abip_float *u_t;
00139     abip_float *u_prev;
00140     abip_float *v_prev;
00141
00142     abip_float* u_avg;
00143     abip_float* v_avg;
00144
00145     abip_float* u_avgcon;
00146     abip_float* v_avgcon;
00147
00148     abip_float* u_sumcon;
00149     abip_float* v_sumcon;
00150
00151
00152     abip_int    fre_old;
00153
00154     abip_float *h;
00155     abip_float *g;
00156     abip_float *pr;
00157     abip_float *dr;
00158
00159     abip_float g_th;
00160     abip_float sc_b;
00161     abip_float sc_c;
00162     abip_float nm_b;
00163     abip_float nm_c;
00164

```

```

00165     abip_float *b;
00166     abip_float *c;
00167     abip_int m;
00168     abip_int n;
00169     ABIPMatrix *A;
00170     abip_float sp;
00171
00172     ABIPLinSysWork *p;
00173     ABIPAdaptWork *adapt;
00174     ABIPSettings *stgs;
00175     ABIPScaling *scal;
00176 };
00177
00178 struct ABIP_RESIDUALS
00179 {
00180     abip_int last_ipm_iter;
00181     abip_int last_admm_iter;
00182     abip_float last_mu;
00183
00184     abip_float res_pri;
00185     abip_float res_dual;
00186     abip_float rel_gap;
00187     abip_float res_infeas;
00188     abip_float res_unbdd;
00189
00190     abip_float ct_x_by_tau;
00191     abip_float bt_y_by_tau;
00192
00193     abip_float tau;
00194     abip_float kap;
00195 };
00196
00197 #ifdef __cplusplus
00198 }
00199 #endif
00200 #endif

```

5.46 include/abip_blas.h File Reference

5.47 abip_blas.h

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

```

00001 #ifndef ABIP_BLAS_H_GUARD
00002 #define ABIP_BLAS_H_GUARD
00003
00004 #ifdef USE_LAPACK
00005
00006 #ifdef __cplusplus
00007 extern "C" {
00008 #endif
00009
00010 #ifndef BLASSUFFIX
00011 #define BLASSUFFIX _
00012 #endif
00013
00014 #if defined(NOBLASSUFFIX) && NOBLASSUFFIX > 0
00015 #ifndef SFLOAT
00016 #define BLAS(x) d##x
00017 #else
00018 #define BLAS(x) s##x
00019 #endif
00020 #else
00021 #define stitch_(pre, x, post) pre##x##post
00022 #define stitch__(pre, x, post) stitch_(pre, x, post)
00023 #ifndef SFLOAT
00024 #define BLAS(x) stitch__(d, x, BLASSUFFIX)
00025 #else
00026 #define BLAS(x) stitch__(s, x, BLASSUFFIX)
00027 #endif
00028 #endif
00029
00030 #ifdef MATLAB_MEX_FILE
00031 typedef ptrdiff_t blas_int;
00032 #elif defined(BLAS64)
00033 #include <stdint.h>
00034 typedef int64_t blas_int;
00035 #else
00036 typedef int blas_int;

```

```
00037 #endif
00038
00039 #ifdef __cplusplus
00040 }
00041 #endif
00042
00043 #endif
00044
00045 #endif
```

5.48 include/adaptive.h File Reference

```
#include "abip.h"
#include "glbopts.h"
#include <math.h>
```

Functions

- [ABIPAdaptWork](#) *[ABIP](#)() [init_adapt](#) ([ABIPWork](#) *w)
- void [ABIP](#)() [free_adapt](#) ([ABIPAdaptWork](#) *a)
- [abip_int](#) [ABIP](#)() [adaptive](#) ([ABIPWork](#) *w, [abip_int](#) iter)
- char *[ABIP](#)() [get_adapt_summary](#) (const [ABIPInfo](#) *info, [ABIPAdaptWork](#) *a)

5.48.1 Function Documentation

5.48.1.1 [adaptive\(\)](#)

```
abip\_int ABIP() adaptive (
    ABIPWork * w,
    abip\_int iter )
```

Definition at line 305 of file [adaptive.c](#).

5.48.1.2 [free_adapt\(\)](#)

```
void ABIP() free\_adapt (
    ABIPAdaptWork * a )
```

Definition at line 336 of file [adaptive.c](#).

5.48.1.3 get_adapt_summary()

```
char *ABIP() get_adapt_summary (
    const ABIPInfo * info,
    ABIPAdaptWork * a )
```

Definition at line 406 of file [adaptive.c](#).

5.48.1.4 init_adapt()

```
ABIPAdaptWork *ABIP() init_adapt (
    ABIPWork * w )
```

Definition at line 258 of file [adaptive.c](#).

5.49 adaptive.h

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

```
00001 #ifndef ADAPT_H_GUARD
00002 #define ADAPT_H_GUARD
00003
00004 #ifdef __cplusplus
00005 extern "C" {
00006 #endif
00007
00008 #include "abip.h"
00009 #include "glbopts.h"
00010 #include <math.h>
00011
00012 ABIPAdaptWork *ABIP(init_adapt)
00013 (
00014     ABIPWork *w
00015 );
00016
00017 void ABIP(free_adapt)
00018 (
00019     ABIPAdaptWork *a
00020 );
00021
00022 abip_int ABIP(adaptive)
00023 (
00024     ABIPWork *w,
00025     abip_int iter
00026 );
00027
00028 char *ABIP(get_adapt_summary)
00029 (
00030     const ABIPInfo *info,
00031     ABIPAdaptWork *a
00032 );
00033
00034 #ifdef __cplusplus
00035 }
00036 #endif
00037 #endif
```

5.50 include/cs.h File Reference

```
#include "glbopts.h"
```

Functions

- struct [ABIP](#) ([cs_sparse](#))
- [cs *ABIP\(\)](#) [cs_compress](#) (const [cs](#) *[T](#))
- [cs *ABIP\(\)](#) [cs_spalloc](#) ([abip_int](#) [m](#), [abip_int](#) [n](#), [abip_int](#) [nnzmax](#), [abip_int](#) [values](#), [abip_int](#) [triplet](#))
- [cs *ABIP\(\)](#) [cs_sprefree](#) ([cs](#) *[A](#))
- [abip_float](#) [ABIP\(\)](#) [cs_cumsum](#) ([abip_int](#) *[p](#), [abip_int](#) *[c](#), [abip_int](#) [n](#))
- [cs *ABIP\(\)](#) [cs_transpose](#) (const [cs](#) *[A](#), [abip_int](#) [values](#))
- [abip_int](#) *[ABIP\(\)](#) [cs_pinv](#) ([abip_int](#) const *[p](#), [abip_int](#) [n](#))
- [cs *ABIP\(\)](#) [cs_symperm](#) (const [cs](#) *[A](#), const [abip_int](#) *[pinv](#), [abip_int](#) [values](#))

Variables

- [cs](#)

5.50.1 Function Documentation

5.50.1.1 [ABIP\(\)](#)

```
struct ABIP (
    cs_sparse )
```

Definition at line 1 of file [cs.h](#).

5.50.1.2 [cs_compress\(\)](#)

```
cs *ABIP() cs_compress (
    const cs * T )
```

Definition at line 57 of file [cs.c](#).

5.50.1.3 [cs_cumsum\(\)](#)

```
abip_float ABIP() cs_cumsum (
    abip_int * p,
    abip_int * c,
    abip_int n )
```

Definition at line 162 of file [cs.c](#).

5.50.1.4 cs_pinv()

```
abip_int *ABIP() cs_pinv (
    abip_int const * p,
    abip_int n )
```

Definition at line 190 of file [cs.c](#).

5.50.1.5 cs_spalloc()

```
cs *ABIP() cs_spalloc (
    abip_int m,
    abip_int n,
    abip_int nnzmax,
    abip_int values,
    abip_int triplet )
```

Definition at line 117 of file [cs.c](#).

5.50.1.6 cs_spfree()

```
cs *ABIP() cs_spfree (
    cs * A )
```

Definition at line 145 of file [cs.c](#).

5.50.1.7 cs_symperm()

```
cs *ABIP() cs_symperm (
    const cs * A,
    const abip_int * pinv,
    abip_int values )
```

Definition at line 218 of file [cs.c](#).

5.50.1.8 cs_transpose()

```
cs *ABIP() cs_transpose (
    const cs * A,
    abip_int values )
```

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

5.50.2 Variable Documentation

5.50.2.1 cs

CS

Definition at line 19 of file [cs.h](#).

5.51 cs.h

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

```

00001 #ifndef CS_H_GUARD
00002 #define CS_H_GUARD
00003
00004 #ifdef __cplusplus
00005 extern "C" {
00006 #endif
00007
00008 #include "glbopts.h"
00009
00010 typedef struct ABIP(cs_sparse)
00011 {
00012     abip_int nnzmax;
00013     abip_int m;
00014     abip_int n;
00015     abip_int *p;
00016     abip_int *i;
00017     abip_float *x;
00018     abip_int nnz;
00019 } cs;
00020
00021 cs *ABIP(cs_compress)
00022 (
00023     const cs *T
00024 );
00025
00026 cs *ABIP(cs_spalloc)
00027 (
00028     abip_int m,
00029     abip_int n,
00030     abip_int nnzmax,
00031     abip_int values,
00032     abip_int triplet
00033 );
00034
00035 cs *ABIP(cs_spfree)
00036 (
00037     cs *A
00038 );
00039
00040 abip_float ABIP(cs_cumsum)
00041 (
00042     abip_int *p,
00043     abip_int *c,
00044     abip_int n
00045 );
00046
00047 cs *ABIP(cs_transpose)
00048 (
00049     const cs *A,
00050     abip_int values
00051 );
00052
00053 abip_int *ABIP(cs_pinv)
00054 (
00055     abip_int const *p,
00056     abip_int n
00057 );
00058
00059 cs *ABIP(cs_symperm)
00060 (
00061     const cs *A,
```

```
00062     const abip_int *pinv,
00063     abip_int values
00064 );
00065
00066 #ifdef __cplusplus
00067 }
00068 #endif
00069 #endif
```

5.52 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`

5.52.1 Macro Definition Documentation

5.52.1.1 abip_end_interrupt_listener

```
#define abip_end_interrupt_listener( )
```

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

5.52.1.2 abip_is_interrupted

```
#define abip_is_interrupted( ) 0
```

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

5.52.1.3 abip_start_interrupt_listener

```
#define abip_start_interrupt_listener( )
```

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

5.52.2 Typedef Documentation

5.52.2.1 abip_make_iso_compilers_happy

typedef [int](#) [abip_make_iso_compilers_happy](#)

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

5.53 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
```

5.54 include/glibopts.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 [MAX_IPM_ITERS](#) (500)
- #define [MAX_ADMM_ITERS](#) (1000000)
- #define [EPS](#) (1E-3)
- #define [ALPHA](#) (1.8)
- #define [CG_RATE](#) (2.0)
- #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 [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](#) `((scs_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](#)

5.54.1 Macro Definition Documentation

5.54.1.1 [_abip_calloc](#)

```
#define _abip_calloc calloc
```

Definition at line [75](#) of file [glbopts.h](#).

5.54.1.2 [_abip_free](#)

```
#define _abip_free free
```

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

5.54.1.3 [_abip_malloc](#)

```
#define _abip_malloc malloc
```

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

5.54.1.4 [_abip_realloc](#)

```
#define _abip_realloc realloc
```

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

5.54.1.5 [ABIP](#)

```
#define ABIP(  
    x ) abip_##x
```

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

5.54.1.6 abip_calloc

```
#define abip_calloc(  
    x,  
    y ) _abip_calloc(x, y)
```

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

5.54.1.7 ABIP_FAILED

```
#define ABIP_FAILED (-4)
```

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

5.54.1.8 abip_free

```
#define abip_free(  
    x )
```

Value:

```
_abip_free(x);  
x = ABIP_NULL
```

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

5.54.1.9 ABIP_INDETERMINATE

```
#define ABIP_INDETERMINATE (-3)
```

Definition at line 26 of file [glbopts.h](#).

5.54.1.10 ABIP_INFEASIBLE

```
#define ABIP_INFEASIBLE (-2)
```

Definition at line 27 of file [glbopts.h](#).

5.54.1.11 ABIP_INFEASIBLE_INACCURATE

```
#define ABIP_INFEASIBLE_INACCURATE (-7)
```

Definition at line 22 of file [glbopts.h](#).

5.54.1.12 abip_malloc

```
#define abip_malloc(  
    x ) _abip_malloc(x)
```

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

5.54.1.13 ABIP_NULL

```
#define ABIP_NULL 0
```

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

5.54.1.14 abip_printf

```
#define abip_printf printf
```

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

5.54.1.15 abip_realloc

```
#define abip_realloc(  
    x,  
    y ) _abip_realloc(x, y)
```

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

5.54.1.16 ABIP_SIGINT

```
#define ABIP_SIGINT (-5)
```

Definition at line 24 of file [glbopts.h](#).

5.54.1.17 ABIP_SOLVED

```
#define ABIP_SOLVED (1)
```

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

5.54.1.18 ABIP_SOLVED_INACCURATE

```
#define ABIP_SOLVED_INACCURATE (2)
```

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

5.54.1.19 ABIP_UNBOUNDED

```
#define ABIP_UNBOUNDED (-1)
```

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

5.54.1.20 ABIP_UNBOUNDED_INACCURATE

```
#define ABIP_UNBOUNDED_INACCURATE (-6)
```

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

5.54.1.21 ABIP_UNFINISHED

```
#define ABIP_UNFINISHED (0)
```

Definition at line 29 of file [glbopts.h](#).

5.54.1.22 ABIP_VERSION

```
#define ABIP_VERSION ("2.0.0") /* string literals automatically null-terminated */
```

Definition at line 19 of file [glbopts.h](#).

5.54.1.23 ABS

```
#define ABS(  
    x ) ((x) < 0) ? -(x) : (x)
```

Definition at line 125 of file [glbopts.h](#).

5.54.1.24 ADAPTIVE

```
#define ADAPTIVE (1)
```

Definition at line 42 of file [glbopts.h](#).

5.54.1.25 ADAPTIVE_LOOKBACK

```
#define ADAPTIVE_LOOKBACK (20)
```

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

5.54.1.26 ALPHA

```
#define ALPHA (1.8)
```

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

5.54.1.27 CG_RATE

```
#define CG_RATE (2.0)
```

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

5.54.1.28 CONVERGED_INTERVAL

```
#define CONVERGED_INTERVAL (1)
```

Definition at line 160 of file [glbopts.h](#).

5.54.1.29 DEBUG_FUNC

```
#define DEBUG_FUNC
```

Definition at line 153 of file [glbopts.h](#).

5.54.1.30 EPS

```
#define EPS (1E-3)
```

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

5.54.1.31 EPS_COR

```
#define EPS_COR (0.2)
```

Definition at line 43 of file [glbopts.h](#).

5.54.1.32 EPS_PEN

```
#define EPS_PEN (0.1)
```

Definition at line 44 of file [glbopts.h](#).

5.54.1.33 EPS_TOL

```
#define EPS_TOL (1E-18)
```

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

5.54.1.34 INDETERMINATE_TOL

```
#define INDETERMINATE_TOL (1e-9)
```

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

5.54.1.35 INFINITY

```
#define INFINITY NAN
```

Definition at line 102 of file [glbopts.h](#).

5.54.1.36 MAX

```
#define MAX(  
    a,  
    b ) ((a) > (b)) ? (a) : (b)
```

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

5.54.1.37 MAX_ADMM_ITERS

```
#define MAX_ADMM_ITERS (1000000)
```

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

5.54.1.38 MAX_IPM_ITERS

```
#define MAX_IPM_ITERS (500)
```

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

5.54.1.39 MIN

```
#define MIN(  
    a,  
    b ) ((a) < (b)) ? (a) : (b)
```

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

5.54.1.40 NAN

```
#define NAN ((scs_float)0x7ff8000000000000)
```

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

5.54.1.41 NORMALIZE

```
#define NORMALIZE (1)
```

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

5.54.1.42 POWF

```
#define POWF pow
```

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

5.54.1.43 RETURN

```
#define RETURN return
```

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

5.54.1.44 RHO_Y

```
#define RHO_Y (1E-3)
```

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

5.54.1.45 SAFEDIV_POS

```
#define SAFEDIV_POS(  
    X,  
    Y ) ((Y) < EPS_TOL ? ((X) / EPS_TOL) : (X) / (Y))
```

Definition at line 158 of file [glbopts.h](#).

5.54.1.46 SCALE

```
#define SCALE (1.0)
```

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

5.54.1.47 SPARSITY_RATIO

```
#define SPARSITY_RATIO (0.01)
```

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

5.54.1.48 SQRTF

```
#define SQRTF sqrt
```

Definition at line 140 of file [glbopts.h](#).

5.54.1.49 VERBOSE

```
#define VERBOSE (1)
```

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

5.54.1.50 WARM_START

```
#define WARM_START (0)
```

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

5.54.2 Typedef Documentation

5.54.2.1 abip_float

```
typedef double abip_float
```

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

5.54.2.2 abip_int

```
typedef int abip_int
```

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

5.55 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 #ifndef ABIP
00011 #define ABIP(x) abip_##x
00012 #endif
00013
00014 // #ifndef ABIP_PARDISO
00015 // #define ABIP_PARDISO
00016 // #endif
00017
00018 /* ABIP VERSION NUMBER ----- */
00019 #define ABIP_VERSION
00020 ("2.0.0") /* string literals automatically null-terminated */
00021
00022 #define ABIP_INFEASIBLE_INACCURATE (-7)
00023 #define ABIP_UNBOUNDED_INACCURATE (-6)
00024 #define ABIP_SIGINT (-5)
00025 #define ABIP_FAILED (-4)
00026 #define ABIP_INDETERMINATE (-3)
00027 #define ABIP_INFEASIBLE (-2)
00028 #define ABIP_UNBOUNDED (-1)
00029 #define ABIP_UNFINISHED (0)
00030 #define ABIP_SOLVED (1)
00031 #define ABIP_SOLVED_INACCURATE (2)
00032
00033 #define MAX_IPM_ITERS (500)
00034 #define MAX_ADMM_ITERS (1000000)
00035 #define EPS (1E-3)
00036 #define ALPHA (1.8)
00037 #define CG_RATE (2.0)
00038 #define NORMALIZE (1)
00039 #define SCALE (1.0)
00040 #define SPARSITY_RATIO (0.01)
00041 #define RHO_Y (1E-3)
00042 #define ADAPTIVE (1)
00043 #define EPS_COR (0.2)
00044 #define EPS_PEN (0.1)
00045 #define ADAPTIVE_LOOKBACK (20)
00046 #define VERBOSE (1)
00047 #define WARM_START (0)
00048
00049 #ifdef MATLAB_MEX_FILE
00050 #include "mex.h"
00051 #define abip_printf mexPrintf
00052 #define _abip_free mxFree
00053 #define _abip_malloc mxMalloc
00054 #define _abip_calloc mxCalloc
00055 #define _abip_realloc mxRealloc
00056 #elif defined PYTHON
00057 #include <Python.h>
00058 #include <stdlib.h>
00059 #define abip_printf(...)
00060 {
00061     PyGILState_STATE gilstate = PyGILState_Ensure();
00062     PySys_WriteStdout(__VA_ARGS__);
00063     PyGILState_Release(gilstate);
00064 }
00065 #define _abip_free free
00066 #define _abip_malloc malloc
00067 #define _abip_calloc calloc
00068 #define _abip_realloc realloc
00069 #else
00070 #include <stdio.h>
00071 #include <stdlib.h>
00072 #define abip_printf printf
00073 #define _abip_free free
00074 #define _abip_malloc malloc
00075 #define _abip_calloc calloc
00076 #define _abip_realloc realloc
00077 #endif
00078
00079 #define abip_free(x) \
00080     _abip_free(x); \
00081     x = ABIP_NULL
00082 #define abip_malloc(x) _abip_malloc(x)

```

```

00083 #define abip_malloc(x, y) _abip_malloc(x, y)
00084 #define abip_realloc(x, y) _abip_realloc(x, y)
00085
00086 #ifdef DLONG
00087 #ifdef _WIN64
00088 typedef __int64 abip_int;
00089 #else
00090 typedef long abip_int;
00091 #endif
00092 #else
00093 typedef int abip_int;
00094 #endif
00095
00096 #ifndef SFLOAT
00097 typedef double abip_float;
00098 #ifndef NAN
00099 #define NAN ((scs_float)0x7ff8000000000000)
00100 #endif
00101 #ifndef INFINITY
00102 #define INFINITY NAN
00103 #endif
00104 #else
00105 typedef float abip_float;
00106 #ifndef NAN
00107 #define NAN ((float)0x7fc00000)
00108 #endif
00109 #ifndef INFINITY
00110 #define INFINITY NAN
00111 #endif
00112 #endif
00113
00114 #define ABIP_NULL 0
00115
00116 #ifndef MAX
00117 #define MAX(a, b) ((a) > (b)) ? (a) : (b)
00118 #endif
00119
00120 #ifndef MIN
00121 #define MIN(a, b) ((a) < (b)) ? (a) : (b)
00122 #endif
00123
00124 #ifndef ABS
00125 #define ABS(x) ((x) < 0) ? -(x) : (x)
00126 #endif
00127
00128 #ifndef POWF
00129 #ifdef SFLOAT
00130 #define POWF powf
00131 #else
00132 #define POWF pow
00133 #endif
00134 #endif
00135
00136 #ifndef SQRTF
00137 #ifdef SFLOAT
00138 #define SQRTF sqrtf
00139 #else
00140 #define SQRTF sqrt
00141 #endif
00142 #endif
00143
00144 #if EXTRA_VERBOSE > 1
00145 #if (defined _WIN32 || defined _WIN64 || defined _WINDLL)
00146 #define __func__ __FUNCTION__
00147 #endif
00148 #define DEBUG_FUNC abip_printf("IN function: %s, time: %4f ms, file: %s, line: %i\n", __func__,
    ABIP(tocq)(&global_timer), __FILE__, __LINE__);
00149 #define RETURN
00150     abip_printf("EXIT function: %s, time: %4f ms, file: %s, line: %i\n", __func__,
    ABIP(tocq)(&global_timer), __FILE__, __LINE__); \
00151     return
00152 #else
00153 #define DEBUG_FUNC
00154 #define RETURN return
00155 #endif
00156
00157 #define EPS_TOL (1E-18)
00158 #define SAFEDIV_POS(X, Y) ((Y) < EPS_TOL ? (X) / EPS_TOL : (X) / (Y))
00159
00160 #define CONVERGED_INTERVAL (1)
00161 #define INDETERMINATE_TOL (1e-9)
00162
00163 #ifdef __cplusplus
00164 }
00165 #endif
00166 #endif

```

5.56 include/linalg.h File Reference

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

Functions

- void **ABIP**() **set_as_scaled_array** (abip_float *x, const abip_float *a, const abip_float b, abip_int len)
compute $x = b \cdot a$
- void **ABIP**() **set_as_sqrt** (abip_float *x, const abip_float *v, abip_int len)
compute $x = \sqrt{v}$
- void **ABIP**() **set_as_sq** (abip_float *x, const abip_float *v, abip_int len)
compute $x = v^2$
- void **ABIP**() **scale_array** (abip_float *a, const abip_float b, abip_int len)
compute $a \leftarrow a \cdot b$
- abip_float **ABIP**() **dot** (const abip_float *x, const abip_float *y, abip_int len)
compute $x \cdot y$
- abip_float **ABIP**() **norm_sq** (const abip_float *v, abip_int len)
compute $\|v\|_2^2$
- abip_float **ABIP**() **norm** (const abip_float *v, abip_int len)
compute $\|v\|_2$
- abip_float **ABIP**() **norm_inf** (const abip_float *a, abip_int len)
compute the infinity norm
- abip_float **ABIP**() **norm_one** (const abip_float *v, abip_int len)
compute L1 norm
- abip_float **ABIP**() **norm_one_sqrt** (const abip_float *v, abip_int len)
compute square root L1 norm
- abip_float **ABIP**() **norm_inf_sqrt** (const abip_float *v, abip_int len)
compute square root infinity norm
- abip_float **ABIP**() **min_abs_sqrt** (const abip_float *a, abip_int len, abip_float ref)
compute square root of the minimal absolute value
- void **ABIP**() **add_array** (abip_float *a, const abip_float b, abip_int len)
compute $a \leftarrow a + b$
- void **ABIP**() **add_scaled_array** (abip_float *a, const abip_float *b, abip_int n, const abip_float sc)
compute $a \leftarrow a + sc \cdot b$
- abip_float **ABIP**() **norm_diff** (const abip_float *a, const abip_float *b, abip_int len)
compute $\|a - b\|_2^2$
- abip_float **ABIP**() **norm_inf_diff** (const abip_float *a, const abip_float *b, abip_int len)
compute $\max(|a - b|)$

5.56.1 Function Documentation

5.56.1.1 add_array()

```
void ABIP() add_array (
    abip_float * a,
    const abip_float b,
    abip_int len )
```

compute $a \mathrel{.+}= b$

Definition at line 218 of file [linalg.c](#).

5.56.1.2 add_scaled_array()

```
void ABIP() add_scaled_array (
    abip_float * a,
    const abip_float * b,
    abip_int n,
    const abip_float sc )
```

compute $a \mathrel{.+}= sc*b$

Definition at line 235 of file [linalg.c](#).

5.56.1.3 dot()

```
abip_float ABIP() dot (
    const abip_float * x,
    const abip_float * y,
    abip_int len )
```

compute $x'y$

Definition at line 78 of file [linalg.c](#).

5.56.1.4 min_abs_sqrt()

```
abip_float ABIP() min_abs_sqrt (
    const abip_float * a,
    abip_int len,
    abip_float ref )
```

compute square root of the minimal absolute value

Definition at line 126 of file [linalg.c](#).

5.56.1.5 norm()

```
abip_float ABIP() norm (
    const abip_float * v,
    abip_int len )
```

compute $\|v\|_2$

Definition at line 115 of file [linalg.c](#).

5.56.1.6 norm_diff()

```
abip_float ABIP() norm_diff (
    const abip_float * a,
    const abip_float * b,
    abip_int len )
```

compute $\|a-b\|_2^2$

Definition at line 253 of file [linalg.c](#).

5.56.1.7 norm_inf()

```
abip_float ABIP() norm_inf (
    const abip_float * a,
    abip_int len )
```

compute the infinity norm

Definition at line 182 of file [linalg.c](#).

5.56.1.8 norm_inf_diff()

```
abip_float ABIP() norm_inf_diff (
    const abip_float * a,
    const abip_float * b,
    abip_int len )
```

compute $\max(|a-b|)$

Definition at line 274 of file [linalg.c](#).

5.56.1.9 norm_inf_sqrt()

```
abip_float ABIP() norm_inf_sqrt (
    const abip_float * v,
    abip_int len )
```

compute square root infinity norm

Definition at line 205 of file [linalg.c](#).

5.56.1.10 norm_one()

```
abip_float ABIP() norm_one (
    const abip_float * v,
    abip_int len )
```

compute L1 norm

Definition at line 149 of file [linalg.c](#).

5.56.1.11 norm_one_sqrt()

```
abip_float ABIP() norm_one_sqrt (
    const abip_float * v,
    abip_int len )
```

compute square root L1 norm

Definition at line 167 of file [linalg.c](#).

5.56.1.12 norm_sq()

```
abip_float ABIP() norm_sq (
    const abip_float * v,
    abip_int len )
```

compute $\|v\|_2^2$

Definition at line 97 of file [linalg.c](#).

5.56.1.13 scale_array()

```
void ABIP() scale_array (
    abip_float * a,
    const abip_float b,
    abip_int len )
```

compute $a *= b$

Definition at line 61 of file [linalg.c](#).

5.56.1.14 set_as_scaled_array()

```
void ABIP() set_as_scaled_array (
    abip_float * x,
    const abip_float * a,
    const abip_float b,
    abip_int len )
```

compute $x = b*a$

Definition at line 9 of file [linalg.c](#).

5.56.1.15 set_as_sq()

```
void ABIP() set_as_sq (
    abip_float * x,
    const abip_float * v,
    abip_int len )
```

compute $x = v.^2$

Definition at line 44 of file [linalg.c](#).

5.56.1.16 set_as_sqrt()

```
void ABIP() set_as_sqrt (
    abip_float * x,
    const abip_float * v,
    abip_int len )
```

compute $x = \text{sqrt}(v)$

Definition at line 27 of file [linalg.c](#).

5.57 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
00011 void ABIP(set_as_scaled_array)
00012 (
00013     abip_float *x,
00014     const abip_float *a,
00015     const abip_float b,
00016     abip_int len
00017 );
00018
00019 void ABIP(set_as_sqrt)
00020 (
00021     abip_float *x,
00022     const abip_float *v,
00023     abip_int len
00024 );
00025
00026 void ABIP(set_as_sq)
00027 (
00028     abip_float *x,
00029     const abip_float *v,
00030     abip_int len
00031 );
00032
00033 void ABIP(scale_array)
00034 (
00035     abip_float *a,
00036     const abip_float b,
00037     abip_int len
00038 );
00039
00040
00041 abip_float ABIP(dot)
00042 (
00043     const abip_float *x,
00044     const abip_float *y,
00045     abip_int len
00046 );
00047
00048 abip_float ABIP(norm_sq)
00049 (
00050     const abip_float *v,
00051     abip_int len
00052 );
00053
00054 abip_float ABIP(norm)
00055 (
00056     const abip_float *v,
00057     abip_int len
00058 );
00059
00060 abip_float ABIP(norm_inf)
00061 (
00062     const abip_float *a,
00063     abip_int len
00064 );
00065
00066 abip_float ABIP(norm_one)
00067 (
00068     const abip_float *v,
00069     abip_int len
00070 );
00071
00072 abip_float ABIP(norm_one_sqrt)
00073 (
00074     const abip_float *v,
00075     abip_int len
00076 );
00077
00078 abip_float ABIP(norm_inf_sqrt)
00079 (
00080     const abip_float *v,
00081     abip_int len
00082 );

```



```

00083
00084 abip_float ABIP(min_abs_sqrt)
00085 (
00086     const abip_float *a,
00087     abip_int len,
00088     abip_float ref
00089 );
00090
00091
00092 void ABIP(add_array)
00093 (
00094     abip_float *a,
00095     const abip_float b,
00096     abip_int len
00097 );
00098
00099 void ABIP(add_scaled_array)
00100 (
00101     abip_float *a,
00102     const abip_float *b,
00103     abip_int n,
00104     const abip_float sc
00105 );
00106
00107 abip_float ABIP(norm_diff)
00108 (
00109     const abip_float *a,
00110     const abip_float *b,
00111     abip_int len
00112 );
00113
00114 abip_float ABIP(norm_inf_diff)
00115 (
00116     const abip_float *a,
00117     const abip_float *b,
00118     abip_int len
00119 );
00120
00121 #ifdef __cplusplus
00122 }
00123 #endif
00124 #endif

```

5.58 include/linsys.h File Reference

```
#include "abip.h"
```

Functions

- [ABIPLinSysWork *ABIP\(\) init_lin_sys_work](#) (const [ABIPMatrix *A](#), const [ABIPSettings *stgs](#))
- [abip_int ABIP\(\) solve_lin_sys](#) (const [ABIPMatrix *A](#), const [ABIPSettings *stgs](#), [ABIPLinSysWork *p](#), [abip_float *b](#), const [abip_float *s](#), [abip_int](#) iter)
- [void ABIP\(\) free_lin_sys_work](#) ([ABIPLinSysWork *p](#))
- [void ABIP\(\) free_lin_sys_work_pds](#) ([ABIPLinSysWork *p](#), [ABIPMatrix *A](#))
- [void ABIP\(\) accum_by_Atrans](#) (const [ABIPMatrix *A](#), [ABIPLinSysWork *p](#), const [abip_float *x](#), [abip_float *y](#))
- [void ABIP\(\) accum_by_A](#) (const [ABIPMatrix *A](#), [ABIPLinSysWork *p](#), const [abip_float *x](#), [abip_float *y](#))
- [abip_int ABIP\(\) validate_lin_sys](#) (const [ABIPMatrix *A](#))
 - validate the linear system*
- [char *ABIP\(\) get_lin_sys_method](#) (const [ABIPMatrix *A](#), const [ABIPSettings *stgs](#))
- [char *ABIP\(\) get_lin_sys_summary](#) ([ABIPLinSysWork *p](#), const [ABIPInfo *info](#))
- [void ABIP\(\) normalize_A](#) ([ABIPMatrix *A](#), const [ABIPSettings *stgs](#), [ABIPScaling *scal](#))
- [void ABIP\(\) un_normalize_A](#) ([ABIPMatrix *A](#), const [ABIPSettings *stgs](#), const [ABIPScaling *scal](#))
- [void ABIP\(\) free_A_matrix](#) ([ABIPMatrix *A](#))
 - set the memory of matrix A free*
- [abip_int ABIP\(\) copy_A_matrix](#) ([ABIPMatrix **dstp](#), const [ABIPMatrix *src](#))
 - copy matrix A*

5.58.1 Function Documentation

5.58.1.1 accum_by_A()

```
void ABIP() accum_by_A (
    const ABIPMatrix * A,
    ABIPLinSysWork * p,
    const abip_float * x,
    abip_float * y )
```

Definition at line 205 of file [direct.c](#).

5.58.1.2 accum_by_Atrans()

```
void ABIP() accum_by_Atrans (
    const ABIPMatrix * A,
    ABIPLinSysWork * p,
    const abip_float * x,
    abip_float * y )
```

Definition at line 200 of file [direct.c](#).

5.58.1.3 copy_A_matrix()

```
abip_int ABIP() copy_A_matrix (
    ABIPMatrix ** dstp,
    const ABIPMatrix * src )
```

copy matrix A

Definition at line 10 of file [common.c](#).

5.58.1.4 free_A_matrix()

```
void ABIP() free_A_matrix (
    ABIPMatrix * A )
```

set the memory of matrix A free

Definition at line 99 of file [common.c](#).

5.58.1.5 free_lin_sys_work()

```
void ABIP() free_lin_sys_work (
    ABIPLinSysWork * p )
```

Definition at line 28 of file [direct.c](#).

5.58.1.6 free_lin_sys_work_pds()

```
void ABIP() free_lin_sys_work_pds (
    ABIPLinSysWork * p,
    ABIPMatrix * A )
```

5.58.1.7 get_lin_sys_method()

```
char *ABIP() get_lin_sys_method (
    const ABIPMatrix * A,
    const ABIPSettings * stgs )
```

Definition at line 5 of file [direct.c](#).

5.58.1.8 get_lin_sys_summary()

```
char *ABIP() get_lin_sys_summary (
    ABIPLinSysWork * p,
    const ABIPInfo * info )
```

Definition at line 15 of file [direct.c](#).

5.58.1.9 init_lin_sys_work()

```
ABIPLinSysWork *ABIP() init_lin_sys_work (
    const ABIPMatrix * A,
    const ABIPSettings * stgs )
```

Definition at line 273 of file [direct.c](#).

5.58.1.10 normalize_A()

```
void ABIP() normalize_A (
    ABIPMatrix * A,
    const ABIPSettings * stgs,
    ABIPScaling * scal )
```

Definition at line 210 of file [direct.c](#).

5.58.1.11 solve_lin_sys()

```
abip_int ABIP() solve_lin_sys (
    const ABIPMatrix * A,
    const ABIPSettings * stgs,
    ABIPLinSysWork * p,
    abip_float * b,
    const abip_float * s,
    abip_int iter )
```

Definition at line 305 of file [direct.c](#).

5.58.1.12 un_normalize_A()

```
void ABIP() un_normalize_A (
    ABIPMatrix * A,
    const ABIPSettings * stgs,
    const ABIPScaling * scal )
```

Definition at line 214 of file [direct.c](#).

5.58.1.13 validate_lin_sys()

```
abip_int ABIP() validate_lin_sys (
    const ABIPMatrix * A )
```

validate the linear system

Definition at line 45 of file [common.c](#).

5.59 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 "abip.h"
00009
00010 ABIPLinSysWork *ABIP(init_lin_sys_work)(const ABIPMatrix *A, const ABIPSettings *stgs);
00011
00012 abip_int ABIP(solve_lin_sys)
00013 (
00014     const ABIPMatrix *A,
00015     const ABIPSettings *stgs,
00016     ABIPLinSysWork *p,
00017     abip_float *b,
00018     const abip_float *s,
00019     abip_int iter
00020 );
00021
00022 void ABIP(free_lin_sys_work)
00023 (
00024     ABIPLinSysWork *p
00025 );
00026
00027 void ABIP(free_lin_sys_work_pds)
00028 (
00029     ABIPLinSysWork *p,
00030     ABIPMatrix *A
00031 );
00032
00033 /* forms y += A'*x */
00034 void ABIP(accum_by_Atrans)
00035 (
00036     const ABIPMatrix *A,
00037     ABIPLinSysWork *p,
00038     const abip_float *x,
00039     abip_float *y
00040 );
00041
00042 /* forms y += A*x */
00043 void ABIP(accum_by_A)
00044 (
00045     const ABIPMatrix *A,
00046     ABIPLinSysWork *p,
00047     const abip_float *x,
00048     abip_float *y
00049 );
00050
00051 abip_int ABIP(validate_lin_sys)
00052 (
00053     const ABIPMatrix *A
00054 );
00055
00056 char *ABIP(get_lin_sys_method)
00057 (
00058     const ABIPMatrix *A,
00059     const ABIPSettings *stgs
00060 );
00061
00062 char *ABIP(get_lin_sys_summary)
00063 (
00064     ABIPLinSysWork *p,
00065     const ABIPInfo *info
00066 );
00067
00068 void ABIP(normalize_A)
00069 (
00070     ABIPMatrix *A,
00071     const ABIPSettings *stgs,
00072     ABIPScaling *scal
00073 );
00074
00075 void ABIP(un_normalize_A)
00076 (
00077     ABIPMatrix *A,
00078     const ABIPSettings *stgs,
00079     const ABIPScaling *scal
00080 );
00081
00082 void ABIP(free_A_matrix)

```

```

00083 (
00084     ABIPMatrix *A
00085 );
00086
00087 abip_int ABIP(copy_A_matrix)
00088 (
00089     ABIPMatrix **dstp,
00090     const ABIPMatrix *src
00091 );
00092
00093 #ifdef __cplusplus
00094 }
00095 #endif
00096
00097 #endif

```

5.60 include/normalize.h File Reference

```
#include "abip.h"
```

Functions

- void [ABIP\(\)](#) [normalize_b_c](#) ([ABIPWork](#) *w)
normalize b and c
- void [ABIP\(\)](#) [calc_scaled_resids](#) ([ABIPWork](#) *w, [ABIPResiduals](#) *r)
calculate the scaled residuals
- void [ABIP\(\)](#) [normalize_warm_start](#) ([ABIPWork](#) *w)
normalize the warm start solution
- void [ABIP\(\)](#) [un_normalize_sol](#) ([ABIPWork](#) *w, [ABIPSolution](#) *sol)
recover the optimal solution

5.60.1 Function Documentation

5.60.1.1 [calc_scaled_resids\(\)](#)

```

void ABIP\(\) calc\_scaled\_resids (
    ABIPWork * w,
    ABIPResiduals * r )

```

calculate the scaled residuals

Definition at line [44](#) of file [normalize.c](#).

5.60.1.2 [normalize_b_c\(\)](#)

```

void ABIP\(\) normalize\_b\_c (
    ABIPWork * w )

```

normalize b and c

Definition at line [11](#) of file [normalize.c](#).

5.60.1.3 normalize_warm_start()

```
void ABIP() normalize_warm_start (
    ABIPWork * w )
```

normalize the warm start solution

Definition at line 100 of file [normalize.c](#).

5.60.1.4 un_normalize_sol()

```
void ABIP() un_normalize_sol (
    ABIPWork * w,
    ABIPSolution * sol )
```

recover the optimal solution

Definition at line 133 of file [normalize.c](#).

5.61 normalize.h

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

```
00001 #ifndef NORMALIZE_H_GUARD
00002 #define NORMALIZE_H_GUARD
00003
00004 #ifdef __cplusplus
00005 extern "C" {
00006 #endif
00007
00008 #include "abip.h"
00009
00010 void ABIP(normalize_b_c)
00011 (
00012     ABIPWork *w
00013 );
00014
00015 void ABIP(calc_scaled_resids)
00016 (
00017     ABIPWork *w,
00018     ABIPResiduals *r
00019 );
00020
00021 void ABIP(normalize_warm_start)
00022 (
00023     ABIPWork *w
00024 );
00025
00026 void ABIP(un_normalize_sol)
00027 (
00028     ABIPWork *w,
00029     ABIPSolution *sol
00030 );
00031
00032 #ifdef __cplusplus
00033 }
00034 #endif
00035 #endif
```

5.62 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)
 - define toc function*
- [abip_float](#) [ABIP](#)() [str_toc](#) (char *str, [ABIP](#)(timer) *t)
 - store time consumed*
- [abip_float](#) [ABIP](#)() [tocq](#) ([ABIP](#)(timer) *t)
- void [ABIP](#)() [print_data](#) (const [ABIPData](#) *d)
 - print some parameters*
- void [ABIP](#)() [print_work](#) (const [ABIPWork](#) *w)
 - print the iterates*
- void [ABIP](#)() [print_array](#) (const [abip_float](#) *arr, [abip_int](#) n, const char *name)
 - print array*
- void [ABIP](#)() [set_default_settings](#) ([ABIPData](#) *d)
 - set default setting*
- void [ABIP](#)() [free_sol](#) ([ABIPSolution](#) *sol)
 - set the memory of solution free*
- void [ABIP](#)() [free_data](#) ([ABIPData](#) *d)
 - set the memory of problem data free*

5.62.1 Function Documentation

5.62.1.1 [ABIP](#)()

```
ABIP (
    timer )
```

Definition at line 1 of file [util.h](#).

5.62.1.2 [free_data](#)()

```
void ABIP() free_data (
    ABIPData * d )
```

set the memory of problem data free

Definition at line 226 of file [util.c](#).

5.62.1.3 free_sol()

```
void ABIP() free_sol (
    ABIPSolution * sol )
```

set the memory of solution free

Definition at line 259 of file [util.c](#).

5.62.1.4 print_array()

```
void ABIP() print_array (
    const abip_float * arr,
    abip_int n,
    const char * name )
```

print array

Definition at line 191 of file [util.c](#).

5.62.1.5 print_data()

```
void ABIP() print_data (
    const ABIPData * d )
```

print some parameters

Definition at line 162 of file [util.c](#).

5.62.1.6 print_work()

```
void ABIP() print_work (
    const ABIPWork * w )
```

print the iterates

Definition at line 133 of file [util.c](#).

5.62.1.7 `set_default_settings()`

```
void ABIP() set_default_settings (
    ABIPData * d )
```

set default setting

Definition at line 288 of file [util.c](#).

5.62.1.8 `str_toc()`

```
abip_float ABIP() str_toc (
    char * str,
    ABIP(timer) * t )
```

store time consumed

Definition at line 120 of file [util.c](#).

5.62.1.9 `tic()`

```
void ABIP() tic (
    ABIP(timer) * t )
```

Definition at line 73 of file [util.c](#).

5.62.1.10 `toc()`

```
abip_float ABIP() toc (
    ABIP(timer) * t )
```

define toc function

Definition at line 108 of file [util.c](#).

5.62.1.11 `tocq()`

```
abip_float ABIP() tocq (
    ABIP(timer) * t )
```

Definition at line 81 of file [util.c](#).

5.63 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_sol) (ABIPSolution *sol);
00060 void ABIP(free_data) (ABIPData *d);
00061
00062 #ifdef __cplusplus
00063 }
00064 #endif
00065 #endif

```

5.64 interface/abip_direct.m File Reference

Functions

- `s t A` in $R (m * n)$ and `b` in R^m . % % this uses the direct linear equation solver [version](#) of ABIP. % % data must consist of [data.A](#)

Variables

- `function` [x, y, s, info]
- `s t Ax = b`
- `s t x`
- `s t A` in data `b`
- `s t A` in data data `c`
- `s t A` in data data where `A`

5.64.1 Function Documentation

5.64.1.1 `R()`

```
s t A in R (
    m * n )
```

5.64.2 Variable Documentation

5.64.2.1 `A`

```
s t A in data data where A
```

Definition at line 15 of file `abip_direct.m`.

5.64.2.2 `Ax`

```
s t Ax = b
```

Definition at line 9 of file `abip_direct.m`.

5.64.2.3 `b`

```
s t A in data data where b
```

Definition at line 15 of file `abip_direct.m`.

5.64.2.4 c

abip_version c

Definition at line 15 of file [abip_direct.m](#).

5.64.2.5 function

function[x, y, s, info]

Initial value:

```
= abip_direct(data, params)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% ADMM-Based Interior-Point Method for solving linear programs (abip_direct):
%
% This implements a LP solver using sparse LDL^T factorization. It solves:
%
% min. c'x
```

Definition at line 1 of file [abip_direct.m](#).

5.64.2.6 x

s t x

Initial value:

```
=0.
%
% where x \in R^n
```

Definition at line 9 of file [abip_direct.m](#).

5.65 abip_direct.m

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

```
00001 function [x, y, s, info] = abip_direct(data, params)
00002
00003 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
00004 %% ADMM-Based Interior-Point Method for solving linear programs (abip_direct):
00005 %
00006 % This implements a LP solver using sparse LDL^T factorization. It solves:
00007 %
00008 % min. c'x,
00009 % s.t. Ax = b, x>=0.
00010 %
00011 % where x \in R^n, A \in R^(m*n) and b \in R^m.
00012 %
00013 % this uses the direct linear equation solver version of ABIP.
00014 %
00015 % data must consist of data.A, data.b, data.c, where A,b,c used as above.
00016 %
00017 % Optional fields in the params struct are:
00018 %   max_ipm_iters : maximum number of IPM iterations.
00019 %   max_admm_iters : maximum number of ADMM iterations.
00020 %   eps : quitting tolerance.
00021 %   sigma : aggressiveness measurement of the IPM framework.
00022 %   alpha : over-relaxation parameter, between (0,2), alpha=1 is unrelaxed.
00023 %   normalize : heuristic nomarlization procedure, between 0 and 1, off or on.
00024 %   scale : heuristic rescale procedure, only used if normalize=1.
00025 %   adaptive : heuristic barzilai-borwein spectral procedure.
00026 %   verbose : verbosity level (0 or 1)
00027 %
00028 % to warm-start the solver add guesses for (x, y, s) to the data struct
00029 %
00030 error ('abip_direct mexFunction not found') ;
```

5.66 interface/abip_indirect.m File Reference

Functions

- s t [A](#) in [R](#) ($m \times n$) and [b](#) \in R^m . % % this uses the indirect linear equation solver [version](#) of ABIP. % % data must consist of [data.A](#)

Variables

- [function](#) [[x](#), [y](#), [s](#), [info](#)]
- s t [Ax](#) = [b](#)
- s t [x](#)
- s t [A](#) in data [b](#)
- s t [A](#) in data data [c](#)
- s t [A](#) in data data where [A](#)

5.66.1 Function Documentation

5.66.1.1 [R\(\)](#)

s t [A](#) in [R](#) (
 $m \times n$)

5.66.2 Variable Documentation

5.66.2.1 [A](#)

s t [A](#) in data data where [A](#)

Definition at line [15](#) of file [abip_indirect.m](#).

5.66.2.2 [Ax](#)

s t [Ax](#) = [b](#)

Definition at line [9](#) of file [abip_indirect.m](#).

5.66.2.3 b

s t A in data data where b

Definition at line 15 of file [abip_indirect.m](#).

5.66.2.4 c

s t A in data data c

Definition at line 15 of file [abip_indirect.m](#).

5.66.2.5 function

```
function[x, y, s, info]
```

Initial value:

```
= abip_indirect(data, params)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% ADMM-Based Interior-Point Method for solving linear programs (abip_indirect):
%%
%% This implements a LP solver using sparse LDL^T factorization. It solves:
%%
%% min. c'x
```

Definition at line 1 of file [abip_indirect.m](#).

5.66.2.6 x

s t x

Initial value:

```
=0.
%
% where x \in R^n
```

Definition at line 9 of file [abip_indirect.m](#).

5.67 abip_indirect.m

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

```
00001 function [x, y, s, info] = abip_indirect(data, params)
00002
00003 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
00004 %% ADMM-Based Interior-Point Method for solving linear programs (abip_indirect):
00005 %
00006 % This implements a LP solver using sparse LDL^T factorization. It solves:
00007 %
00008 % min. c'x,
00009 % s.t. Ax = b, x>=0.
00010 %
00011 % where x \in R^n, A \in R^(m*n) and b \in R^m.
00012 %
00013 % this uses the indirect linear equation solver version of ABIP.
00014 %
00015 % data must consist of data.A, data.b, data.c, where A,b,c used as above.
00016 %
00017 % Optional fields in the params struct are:
00018 %   max_ipm_iters :      maximum number of IPM iterations.
00019 %   max_admm_iters :    maximum number of ADMM iterations.
00020 %   eps :               quitting tolerance.
00021 %   sigma :            aggressiveness measurement of the IPM framework.
00022 %   alpha :            over-relaxation parameter, between (0,2), alpha=1 is unrelaxed.
00023 %   normalize :        heuristic nomarlization procedure, between 0 and 1, off or on.
00024 %   scale :            heuristic rescale procedure, only used if normalize=1.
00025 %   adaptive :         heuristic barzilai-borwein spectral procedure.
00026 %   verbose :          verbosity level (0 or 1)
00027 %
00028 % to warm-start the solver add guesses for (x, y, s) to the data struct
00029 %
00030 error ('abip_indirect mexFunction not found') ;
```

5.68 linsys/abip_pardiso.c File Reference

```
#include "abip.h"
#include "cs.h"
#include "amd.h"
#include "ldl.h"
#include "direct.h"
#include "abip_pardiso.h"
```

Functions

- [abip_int pardisoFactorize](#) ([ABIPLinSysWork](#) *p, [cs](#) *A)
factorize matrix by pardiso
- void [pardisoFree](#) ([ABIPLinSysWork](#) *p, [ABIPMatrix](#) *A)
Free the internal structure of pardiso.
- void [pardisoSolve](#) ([ABIPLinSysWork](#) *p, [ABIPMatrix](#) *A, [abip_float](#) *b)
*Solve the linear system $S * X = B$ using Pardiso.*

5.68.1 Function Documentation

5.68.1.1 pardisoFactorize()

```
abip_int pardisoFactorize (
    ABIPLinSysWork * p,
    cs * A )
```

factorize matrix by pardiso

Definition at line 23 of file [abip_pardiso.c](#).

5.68.1.2 pardisoFree()

```
void pardisoFree (
    ABIPLinSysWork * p,
    ABIPMatrix * A )
```

Free the internal structure of pardiso.

Definition at line 51 of file [abip_pardiso.c](#).

5.68.1.3 pardisoSolve()

```
void pardisoSolve (
    ABIPLinSysWork * p,
    ABIPMatrix * A,
    abip_float * b )
```

Solve the linear system $S * X = B$ using Pardiso.

Definition at line 66 of file [abip_pardiso.c](#).

5.69 abip_pardiso.c

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

```
00001 #include "abip.h"
00002 #include "cs.h"
00003 #include "amd.h"
00004 #include "ldl.h"
00005 #include "direct.h"
00006 #include "abip_pardiso.h"
00010 static void printMat( cs *A ) {
00011     abip_int *Ap = A->p, *Ai = A->i; abip_float *Ax = A->x;
00012     abip_printf("Matrix dimension: %d by %d \n", A->m, A->n);
00013     abip_printf("%8s %8s %8s \n", "i", "j", "val");
00014     for (abip_int i = 0, j; i < A->m; ++i) {
00015         for (j = Ap[i]; j < Ap[i + 1]; ++j) {
00016             abip_printf("%8d %8d %8.2e \n", Ai[j], i, Ax[j]);
00017         }
00018     }
00019 }
00023 extern abip_int pardisoFactorize( ABIPLinSysWork *p, cs *A ) {
00024     /* Factorize the spsMat matrix */
00025     abip_int phase = PARDISO_SYM_FAC, error = PARDISO_OK;
00026     pardiso(p->pardiso_work, &maxfct, &mnum, &mtype, &phase, &A->n,
00027         A->x, A->p, A->i, p->P, &idummy, PARDISO_PARAMS_LDL,
```

```

00028         &msglvl, NULL, NULL, &error);
00029
00030     if (!p->i) {
00031         p->i = (abip_int *) abip_malloc((A->m + 1), sizeof(abip_int));
00032     }
00033
00034     if (!p->j) {
00035         p->j = (abip_int *) abip_malloc(A->p[A->m], sizeof(abip_int));
00036     }
00037
00038     memcpy(p->i, A->p, sizeof(abip_int) * (A->m + 1));
00039     memcpy(p->j, A->i, sizeof(abip_int) * A->p[A->m]);
00040
00041     if (error) {
00042         abip_printf("[Pardiso Error]: Matrix factorization failed."
00043             " Error code: %d \n", error);
00044     }
00045
00046     return error;
00047 }
00051 extern void pardisoFree( ABIPLinSysWork *p, ABIPMatrix *A ) {
00052     abip_int phase = PARDISO_FREE, error = PARDISO_OK, n = A->m + A->n, one = 1;
00053     pardiso(p->pardiso_work, &maxfct, &mnum, &mtype, &phase, &n,
00054         NULL, p->i, p->j, &idummy, &one,
00055         PARDISO_PARAMS_LDL, &msglvl, &ddummy, &ddummy, &error);
00056     abip_free(p->i); abip_free(p->j);
00057     if (error) {
00058         abip_printf("[Pardiso Error]: Pardiso free failed."
00059             " Error code %d \n", error);
00060     }
00061 }
00062
00066 extern void pardisoSolve( ABIPLinSysWork *p, ABIPMatrix *A, abip_float *b ) {
00067
00068     abip_int phase = PARDISO_SOLVE, error = PARDISO_OK, nrhs = 1, n = A->m + A->n;
00069     pardiso(p->pardiso_work, &maxfct, &mnum, &mtype, &phase, &n,
00070         NULL, p->i, p->j, &idummy, &nrhs, PARDISO_PARAMS_LDL,
00071         &msglvl, b, p->D, &error);
00072     if (error) {
00073         abip_printf("[Pardiso Error]: Pardiso solve failed."
00074             " Error code %d \n", error);
00075     }
00076 }
00077

```

5.70 linsys/abip_pardiso.h File Reference

```

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

```

Macros

- #define PARDISO_OK (0)
- #define PARDISOINDEX (64)
- #define PARDISO_SYM (11)
- #define PARDISO_FAC (22)
- #define PARDISO_SYM_FAC (12)
- #define PARDISO_SOLVE (33)
- #define PARDISO_FORWARD (331)
- #define PARDISO_BACKWARD (333)
- #define PARDISO_FREE (-1)
- #define SYMBOLIC 3
- #define PIVOTING 2
- #define FACTORIZE 0

Functions

- void [pardisoinit](#) (void *, const [abip_int](#) *, [abip_int](#) *)
- void [pardiso](#) (void *, [abip_int](#) *, [abip_int](#) *, [abip_int](#) *, [abip_int](#) *, [abip_int](#) *, double *, [abip_int](#) *, [abip_int](#) *, [abip_int](#) *, [abip_int](#) *, [abip_int](#) *, double *, double *, [abip_int](#) *)
- [abip_int](#) [pardisoFactorize](#) ([ABIPLinSysWork](#) *p, [cs](#) *A)
factorize matrix by pardiso
- void [pardisoFree](#) ([ABIPLinSysWork](#) *p, [ABIPMatrix](#) *A)
Free the internal structure of pardiso.
- void [pardisoSolve](#) ([ABIPLinSysWork](#) *p, [ABIPMatrix](#) *A, [abip_float](#) *b)
*Solve the linear system $S * X = B$ using Pardiso.*

5.70.1 Macro Definition Documentation

5.70.1.1 FACTORIZE

```
#define FACTORIZE 0
```

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

5.70.1.2 PARDISO_BACKWARD

```
#define PARDISO_BACKWARD (333)
```

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

5.70.1.3 PARDISO_FAC

```
#define PARDISO_FAC ( 22)
```

Definition at line 12 of file [abip_pardiso.h](#).

5.70.1.4 PARDISO_FORWARD

```
#define PARDISO_FORWARD (331)
```

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

5.70.1.5 PARDISO_FREE

```
#define PARDISO_FREE ( -1)
```

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

5.70.1.6 PARDISO_OK

```
#define PARDISO_OK ( 0)
```

Definition at line 9 of file [abip_pardiso.h](#).

5.70.1.7 PARDISO_SOLVE

```
#define PARDISO_SOLVE ( 33)
```

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

5.70.1.8 PARDISO_SYM

```
#define PARDISO_SYM ( 11)
```

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

5.70.1.9 PARDISO_SYM_FAC

```
#define PARDISO_SYM_FAC ( 12)
```

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

5.70.1.10 PARDISOINDEX

```
#define PARDISOINDEX ( 64)
```

Definition at line 10 of file [abip_pardiso.h](#).

5.70.1.11 PIVOTING

```
#define PIVOTING 2
```

Definition at line 29 of file [abip_pardiso.h](#).

5.70.1.12 SYMBOLIC

```
#define SYMBOLIC 3
```

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

5.70.2 Function Documentation

5.70.2.1 pardiso()

```
void pardiso (
    void * ,
    abip_int * ,
    abip_int * ,
    abip_int * ,
    abip_int * ,
    abip_int * ,
    double * ,
    abip_int * ,
    abip_int * ,
    abip_int * ,
    abip_int * ,
    abip_int * ,
    double * ,
    double * ,
    abip_int * )
```

5.70.2.2 pardisoFactorize()

```
abip_int pardisoFactorize (
    ABIPLinSysWork * p,
    cs * A )
```

factorize matrix by pardiso

Definition at line 23 of file [abip_pardiso.c](#).

5.70.2.3 pardisoFree()

```
void pardisoFree (
    ABIPLinSysWork * p,
    ABIPMatrix * A )
```

Free the internal structure of pardiso.

Definition at line 51 of file [abip_pardiso.c](#).

5.70.2.4 pardisoinit()

```
void pardisoinit (
    void * ,
    const abip_int * ,
    abip_int * )
```

5.70.2.5 pardisoSolve()

```
void pardisoSolve (
    ABIPLinSysWork * p,
    ABIPMatrix * A,
    abip_float * b )
```

Solve the linear system $S * X = B$ using Pardiso.

Definition at line 66 of file [abip_pardiso.c](#).

5.71 abip_pardiso.h

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

```
00001 #ifndef abip_pardiso_h
00002 #define abip_pardiso_h
00003
00004 /* Implement the pardiso solver interface for IPM */
00005
00006 #include <stdio.h>
00007 #include "glbopts.h"
00008
00009 #define PARDISO_OK ( 0)
00010 #define PARDISOINDEX ( 64) // Pardiso working array length
00011 #define PARDISO_SYM ( 11) // Pardiso symbolic analysis
00012 #define PARDISO_FAC ( 22) // Pardiso numerical factorization
00013 #define PARDISO_SYM_FAC ( 12) // Symbolic analysis and factorization
00014 #define PARDISO_SOLVE ( 33) // Solve linear system
00015 #define PARDISO_FORWARD (331) // Pardiso forward solve
00016 #define PARDISO_BACKWARD (333) // Pardiso backward solve
00017 #define PARDISO_FREE ( -1) // Free internal data structure
00018
00019 // Pardiso default parameters
00020 static abip_int maxfct = 1; // Maximum number of factors
00021 static abip_int mnum = 1; // The matrix used for the solution phase
00022 static abip_int mtype = -2; // Real and symmetric indefinite
00023 static abip_int msglvl = 0; // Print no information
00024 static abip_int idummy = 0; // Dummy variable for taking up space
00025 static abip_float ddummy = 0.0; // Dummy variable for taking up space
```

```

00026
00027 // Pardiso solver
00028 #define SYMBOLIC 3
00029 #define PIVOTING 2
00030 #define FACTORIZE 0
00031
00032 static abip_int PARDISO_PARAMS_LDL[PARDISOINDEX] = {
00033
00034     1, /* Non-default value */ SYMBOLIC, /* P Nested dissection */ 0, /* Reserved */
00035     0, /* No CG */ 0, /* No user permutation */ 1, /* Overwriting */
00036     0, /* Refinement report */ 0, /* Auto ItRef step */ 0, /* Reserved */
00037     8, /* Perturb */ 0, /* Disable scaling */ 0, /* No transpose */
00038     0, /* Disable matching */ 0, /* Report on pivots */ 0, /* Output */
00039     0, /* Output */ 0, /* Output */ -1, /* No report */
00040     0, /* No report */ 0, /* Output */ PIVOTING, /* Pivoting */
00041     0, /* nPosEigVals */ 0, /* nNegEigVals */ FACTORIZE, /* Classic factorize */
00042     0, 0, 0, /* Matrix checker */
00043     0, 0, 0,
00044     0, 0, 0,
00045     0, 1, /* 0-based solve */ 0,
00046     0, 0, 0,
00047     0, 0, 0,
00048     0, 0, 0,
00049     0, 0, 0,
00050     0, 0, 0,
00051     0, 0, 0,
00052     0, 0, /* No diagonal */ 0,
00053     0, 0, 0,
00054     0, 0, 0,
00055     0
00056 };
00057
00058 #ifdef __cplusplus
00059 extern "C" {
00060 #endif
00061
00062 extern void pardisoinit ( void *, const abip_int *, abip_int * );
00063
00064 extern void pardiso ( void *, abip_int *, abip_int *, abip_int *, abip_int *, abip_int *, abip_int *,
00065                     double *, abip_int *, abip_int *, abip_int *, abip_int *, abip_int *,
00066                     abip_int *, double *, double *, abip_int * );
00067
00068 extern abip_int pardisoFactorize( ABIPLinSysWork *p, cs *A );
00069 extern void pardisoFree ( ABIPLinSysWork *p, ABIPMatrix *A );
00070 extern void pardisoSolve ( ABIPLinSysWork *p, ABIPMatrix *A, abip_float *b );
00071 #ifdef __cplusplus
00072 }
00073 #endif
00074
00075
00076 #endif /* abip_pardiso_h */

```

5.72 linsys/amatrix.h File Reference

```
#include "glbopts.h"
```

Data Structures

- struct [ABIP_A_DATA_MATRIX](#)

5.73 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 struct ABIP_A_DATA_MATRIX
00011 {
00012     abip_float *x;
00013     abip_int *i;
00014     abip_int *p;
00015     abip_int m;
00016     abip_int n;
00017 };
00018
00019 #ifdef __cplusplus
00020 }
00021 #endif
00022 #endif

```

5.74 linsys/common.c File Reference

```

#include "common.h"
#include "linsys.h"

```

Macros

- #define `MIN_SCALE` (1e-3)
- #define `MAX_SCALE` (1e3)

Functions

- `abip_int ABIP()` `copy_A_matrix` (`ABIPMatrix **dstp`, `const ABIPMatrix *src`)
copy matrix A
- `abip_int ABIP()` `validate_lin_sys` (`const ABIPMatrix *A`)
validate the linear system
- `void ABIP()` `free_A_matrix` (`ABIPMatrix *A`)
set the memory of matrix A free
- `void ABIP()` `_normalize_A` (`ABIPMatrix *A`, `const ABIPSettings *stgs`, `ABIPScaling *scal`)
normalize matrix A
- `void ABIP()` `_un_normalize_A` (`ABIPMatrix *A`, `const ABIPSettings *stgs`, `const ABIPScaling *scal`)
unnormalize matrix A
- `void ABIP()` `_accum_by_Atrans` (`abip_int n`, `abip_float *Ax`, `abip_int *Ai`, `abip_int *Ap`, `const abip_float *x`, `abip_float *y`)
compute $A^T x$
- `void ABIP()` `_accum_by_A` (`abip_int n`, `abip_float *Ax`, `abip_int *Ai`, `abip_int *Ap`, `const abip_float *x`, `abip_float *y`)
compute Ax
- `abip_float ABIP()` `cumsum` (`abip_int *p`, `abip_int *c`, `abip_int n`)
compute cumulative sum of c

5.74.1 Macro Definition Documentation

5.74.1.1 MAX_SCALE

```
#define MAX_SCALE (1e3)
```

Definition at line 5 of file [common.c](#).

5.74.1.2 MIN_SCALE

```
#define MIN_SCALE (1e-3)
```

Definition at line 4 of file [common.c](#).

5.74.2 Function Documentation

5.74.2.1 _accum_by_A()

```
void ABIP() _accum_by_A (
    abip_int n,
    abip_float * Ax,
    abip_int * Ai,
    abip_int * Ap,
    const abip_float * x,
    abip_float * y )
```

compute Ax

Definition at line 644 of file [common.c](#).

5.74.2.2 _accum_by_Atrans()

```
void ABIP() _accum_by_Atrans (
    abip_int n,
    abip_float * Ax,
    abip_int * Ai,
    abip_int * Ap,
    const abip_float * x,
    abip_float * y )
```

compute $A^T x$

Definition at line 598 of file [common.c](#).

5.74.2.3 `_normalize_A()`

```
void ABIP() _normalize_A (
    ABIPMatrix * A,
    const ABIPSettings * stgs,
    ABIPScaling * scal )
```

normalize matrix A

Definition at line 150 of file [common.c](#).

5.74.2.4 `_un_normalize_A()`

```
void ABIP() _un_normalize_A (
    ABIPMatrix * A,
    const ABIPSettings * stgs,
    const ABIPScaling * scal )
```

unnormalize matrix A

Definition at line 569 of file [common.c](#).

5.74.2.5 `copy_A_matrix()`

```
abip_int ABIP() copy_A_matrix (
    ABIPMatrix ** dstp,
    const ABIPMatrix * src )
```

copy matrix A

Definition at line 10 of file [common.c](#).

5.74.2.6 `cumsum()`

```
abip_float ABIP() cumsum (
    abip_int * p,
    abip_int * c,
    abip_int n )
```

compute cumulative sum of c

Definition at line 700 of file [common.c](#).

5.74.2.7 free_A_matrix()

```
void ABIP() free_A_matrix (
    ABIPMatrix * A )
```

set the memory of matrix A free

Definition at line 99 of file [common.c](#).

5.74.2.8 validate_lin_sys()

```
abip_int ABIP() validate_lin_sys (
    const ABIPMatrix * A )
```

validate the linear system

Definition at line 45 of file [common.c](#).

5.75 common.c

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

```
00001 #include "common.h"
00002 #include "linsys.h"
00003
00004 #define MIN_SCALE (1e-3)
00005 #define MAX_SCALE (1e3)
00006
00010 abip_int ABIP(copy_A_matrix)
00011 (
00012     ABIPMatrix **dstp,
00013     const ABIPMatrix *src
00014 )
00015 {
00016     abip_int Annz = src->p[src->n];
00017     ABIPMatrix *A = (ABIPMatrix *)abip_calloc(1, sizeof(ABIPMatrix));
00018     if (!A)
00019     {
00020         return 0;
00021     }
00022     A->n = src->n;
00023     A->m = src->m;
00024     A->x = (abip_float *) abip_malloc(sizeof(abip_float) * Annz);
00025     A->i = (abip_int *) abip_malloc(sizeof(abip_int) * Annz);
00026     A->p = (abip_int *) abip_malloc(sizeof(abip_int) * (src->n + 1));
00027
00028     if (!A->x || !A->i || !A->p)
00029     {
00030         abip_free(A->x); abip_free(A->i); abip_free(A->p); abip_free(A);
00031         return 0;
00032     }
00033
00034     memcpy(A->x, src->x, sizeof(abip_float) * Annz);
00035     memcpy(A->i, src->i, sizeof(abip_int) * Annz);
00036     memcpy(A->p, src->p, sizeof(abip_int) * (src->n + 1));
00037
00038     *dstp = A;
00039     return 1;
00040 }
00041
00045 abip_int ABIP(validate_lin_sys)
00046 (
00047     const ABIPMatrix *A
00048 )
00049 {
00050     abip_int i;
00051     abip_int r_max;
00052     abip_int Annz;
```

```

00053
00054     if (!A->x || !A->i || !A->p)
00055     {
00056         abip_printf("ERROR: incomplete data!\n");
00057         return -1;
00058     }
00059
00060     for (i = 0; i < A->n; ++i)
00061     {
00062         if (A->p[i] == A->p[i + 1])
00063         {
00064             abip_printf("WARN: the %li-th column empty!\n", (long)i);
00065         }
00066         else if (A->p[i] > A->p[i + 1])
00067         {
00068             abip_printf("ERROR: the column pointers decreases!\n");
00069             return -1;
00070         }
00071     }
00072
00073     Annz = A->p[A->n];
00074     if (((abip_float)Annz / A->m > A->n) || (Annz <= 0))
00075     {
00076         abip_printf("ERROR: the number of nonzeros in A = %li, outside of valid range!\n", (long)
Annz);
00077         return -1;
00078     }
00079
00080     r_max = 0;
00081     for (i = 0; i < Annz; ++i)
00082     {
00083         if (A->i[i] > r_max)
00084         {
00085             r_max = A->i[i];
00086         }
00087     }
00088     if (r_max > A->m - 1)
00089     {
00090         abip_printf("ERROR: the number of rows in A is inconsistent with input dimension!\n");
00091         return -1;
00092     }
00093
00094     return 0;
00095 }
00099 void ABIP(free_A_matrix)
00100 {
00101     ABIPMatrix *A
00102 }
00103 {
00104     if (A->x)
00105     {
00106         abip_free(A->x);
00107     }
00108     if (A->i)
00109     {
00110         abip_free(A->i);
00111     }
00112     if (A->p)
00113     {
00114         abip_free(A->p);
00115     }
00116     abip_free(A);
00117 }
00118
00119
00120 #if EXTRA_VERBOSE > 0
00121
00122 static void print_A_matrix
00123 {
00124     const ABIPMatrix *A
00125 }
00126 {
00127     abip_int i;
00128     abip_int j;
00129
00130     if (A->p[A->n] < 2500)
00131     {
00132         abip_printf("\n");
00133         for (i = 0; i < A->n; ++i)
00134         {
00135             abip_printf("Col %li: ", (long)i);
00136             for (j = A->p[i]; j < A->p[i + 1]; j++)
00137             {
00138                 abip_printf("A[%li,%li] = %4f, ", (long)A->i[j], (long)i, A->x[j]);
00139             }
00140             abip_printf("norm col = %4f\n", ABIP(norm)(&(A->x[A->p[i]]), A->p[i + 1] - A->p[i]));
00141         }

```

```

00142         abip_printf("norm A = %4f\n", ABIP(norm)(A->x, A->p[A->n]));
00143     }
00144 }
00145 #endif
00146
00150 void ABIP(_normalize_A)
00151 (
00152     ABIPMatrix *A,
00153     const ABIPSettings *stgs,
00154     ABIPScaling *scal
00155 )
00156 {
00157     abip_float *D = (abip_float *) abip_malloc(A->m * sizeof(abip_float));
00158     abip_float *E = (abip_float *) abip_malloc(A->n * sizeof(abip_float));
00159     abip_float *nms = (abip_float *) abip_calloc(A->m, sizeof(abip_float));
00160
00161     abip_float *D_pc = (abip_float *) abip_malloc(A->m * sizeof(abip_float)); // for pc rescale
00162     abip_float *E_pc = (abip_float *) abip_malloc(A->n * sizeof(abip_float));
00163     abip_float *D_origin = (abip_float *) abip_malloc(A->m * sizeof(abip_float));
00164     abip_float *E_origin = (abip_float *) abip_malloc(A->n * sizeof(abip_float));
00165     abip_float *D_temp = (abip_float *) abip_malloc(A->m * sizeof(abip_float));
00166     abip_float *E_temp = (abip_float *) abip_malloc(A->n * sizeof(abip_float));
00167     abip_float *D_ruiz = (abip_float *) abip_malloc(A->m * sizeof(abip_float)); // for ruiz rescale
00168     abip_float *E_ruiz = (abip_float *) abip_malloc(A->n * sizeof(abip_float));
00169     abip_float *D_qp = (abip_float *) abip_malloc(A->m * sizeof(abip_float)); // for the trial rescale
00170     used in qcp
    abip_float *E_qp = (abip_float *) abip_malloc(A->n * sizeof(abip_float));
00171
00172     abip_float min_row_scale = MIN_SCALE * SQRTF((abip_float)A->n);
00173     abip_float max_row_scale = MAX_SCALE * SQRTF((abip_float)A->n);
00174     abip_float min_col_scale = MIN_SCALE * SQRTF((abip_float)A->m);
00175     abip_float max_col_scale = MAX_SCALE * SQRTF((abip_float)A->m);
00176
00177     abip_int i;
00178     abip_int j;
00179     abip_int c1;
00180     abip_int c2;
00181
00182     //
00183     abip_int k;
00184     abip_int ruiz_iter = stgs->ruiz_iter;
00185     abip_float tmp;
00186     // -----
00187
00188     abip_float wrk;
00189     abip_float e;
00190
00191     #if EXTRA_VERBOSE > 0
00192     ABIP(timer) normalize_timer;
00193     ABIP(tic)(&normalize_timer);
00194     abip_printf("normalizing A\n");
00195     print_A_matrix(A);
00196     #endif
00197
00198     memset(D, 0, A->m * sizeof(abip_float));
00199     memset(E, 0, A->n * sizeof(abip_float));
00200
00201     //
00202     memset(D_pc, 0, A->m * sizeof(abip_float));
00203     memset(E_pc, 0, A->n * sizeof(abip_float));
00204     memset(D_origin, 0, A->m * sizeof(abip_float));
00205     memset(E_origin, 0, A->n * sizeof(abip_float));
00206     memset(D_qp, 0, A->m * sizeof(abip_float));
00207     memset(E_qp, 0, A->n * sizeof(abip_float));
00208
00209     for (i = 0; i < A->m; ++i){
00210         D_ruiz[i] = 1.0;
00211     }
00212
00213     for (i = 0; i < A->n; ++i){
00214         E_ruiz[i] = 1.0;
00215     }
00216
00217     if (stgs->pc_ruiz_rescale)
00218     {
00219         // for pc rescaling
00220         for (i = 0; i < A->n; ++i)
00221         {
00222             c1 = A->p[i + 1] - A->p[i];
00223             e = ABIP(norm_one_sqrt)(&(A->x[A->p[i]]), c1); // compute sqrt of 1 norm
00224             if (e < min_col_scale){
00225                 e = 1;
00226             }
00227             else if (e > max_col_scale){
00228                 e = max_col_scale;
00229             }
00230             ABIP(scale_array)(&(A->x[A->p[i]]), 1.0 / e, c1); // scale A

```

```

00231     E_pc[i] = e;
00232 }
00233
00234 for (i = 0; i < A->n; ++i)
00235 {
00236     c1 = A->p[i];
00237     c2 = A->p[i + 1];
00238     for (j = c1; j < c2; ++j)
00239     {
00240         wrk = A->x[j];
00241         D_pc[A->i[j]] += ABS(wrk);
00242     }
00243 }
00244
00245 for (i = 0; i < A->m; ++i)
00246 {
00247     D_pc[i] = SQRTF(D_pc[i]);
00248     if (D_pc[i] < min_row_scale)
00249     {
00250         D_pc[i] = 1;
00251     }
00252     else if (D_pc[i] > max_row_scale)
00253     {
00254         D_pc[i] = max_row_scale;
00255     }
00256 }
00257
00258 for (i = 0; i < A->n; ++i)
00259 {
00260     for (j = A->p[i]; j < A->p[i + 1]; ++j)
00261     {
00262         A->x[j] /= D_pc[A->i[j]];
00263     }
00264 }
00265 abip_printf("Done the pc rescaling!\n");
00266 }
00267 else
00268 {
00269     for (i = 0; i < A->m; ++i){
00270         D_pc[i] = 1.0;
00271     }
00272
00273     for (i = 0; i < A->n; ++i){
00274         E_pc[i] = 1.0;
00275     }
00276 }
00277
00278 // for origin rescaling
00279 if (stgs->origin_rescale)
00280 {
00281     for (i = 0; i < A->n; ++i)
00282     {
00283         c1 = A->p[i + 1] - A->p[i];
00284         e = ABIP(norm) (&(A->x[A->p[i]]), c1);
00285         if (e < min_col_scale){
00286             e = 1;
00287         }
00288         else if (e > max_col_scale){
00289             e = max_col_scale;
00290         }
00291         ABIP(scale_array) (&(A->x[A->p[i]]), 1.0 / e, c1); // scale A
00292         E_origin[i] = e;
00293     }
00294
00295     for (i = 0; i < A->n; ++i)
00296     {
00297         c1 = A->p[i];
00298         c2 = A->p[i + 1];
00299         for (j = c1; j < c2; ++j)
00300         {
00301             wrk = A->x[j];
00302             D_origin[A->i[j]] += wrk * wrk;
00303         }
00304     }
00305
00306     for (i = 0; i < A->m; ++i)
00307     {
00308         D_origin[i] = SQRTF(D_origin[i]);
00309         if (D_origin[i] < min_row_scale)
00310         {
00311             D_origin[i] = 1;
00312         }
00313         else if (D_origin[i] > max_row_scale)
00314         {
00315             D_origin[i] = max_row_scale;
00316         }
00317     }

```

```

00318
00319     for (i = 0; i < A->n; ++i)
00320     {
00321         for (j = A->p[i]; j < A->p[i + 1]; ++j)
00322         {
00323             A->x[j] /= D_origin[A->i[j]];
00324         }
00325     }
00326     abip_printf("Done the origin rescaling!\n");
00327 }
00328 else
00329 {
00330     for (i = 0; i < A->m; ++i){
00331         D_origin[i] = 1.0;
00332     }
00333
00334     for (i = 0; i < A->n; ++i){
00335         E_origin[i] = 1.0;
00336     }
00337 }
00338
00339 if (stgs->pc_ruiz_rescale)
00340 {
00341     for(k = 0; k < ruiz_iter; ++k){
00342
00343         memset(D_temp, 0, A->m * sizeof(abip_float));
00344         memset(E_temp, 0, A->n * sizeof(abip_float));
00345
00346         // find E_temp
00347         for (i = 0; i < A->n; ++i)
00348         {
00349             c1 = A->p[i + 1] - A->p[i];
00350             e = ABIP(norm_inf_sqrt) (&(A->x[A->p[i]]), c1);
00351
00352             if (e < min_col_scale){
00353                 e = 1;
00354             }
00355             else if (e > max_col_scale){
00356                 e = max_col_scale;
00357             }
00358
00359             ABIP(scale_array) (&(A->x[A->p[i]]), 1.0 / e, c1); // scale A
00360             E_temp[i] = e;
00361         }
00362
00363         // find D_temp
00364         for (i = 0; i < A->n; ++i)
00365         {
00366             c1 = A->p[i];
00367             c2 = A->p[i + 1];
00368
00369             for (j = c1; j < c2; ++j)
00370             {
00371                 wrk = ABS(A->x[j]); //abs j-th nnz
00372
00373                 if (wrk >= D_temp[A->i[j]])
00374                 {
00375                     D_temp[A->i[j]] = wrk;
00376                 }
00377             }
00378         }
00379
00380         for (i = 0; i < A->m; ++i)
00381         {
00382             D_temp[i] = SQRTF(D_temp[i]);
00383
00384             if (D_temp[i] < min_row_scale)
00385             {
00386                 D_temp[i] = 1;
00387             }
00388             else if (D_temp[i] > max_row_scale)
00389             {
00390                 D_temp[i] = max_row_scale;
00391             }
00392         }
00393
00394         for (i = 0; i < A->n; ++i)
00395         {
00396             for (j = A->p[i]; j < A->p[i + 1]; ++j)
00397             {
00398                 A->x[j] /= D_temp[A->i[j]];
00399             }
00400         }
00401
00402         // record E_ruiz
00403         for (i = 0; i < A->n; ++i){
00404             E_ruiz[i] = E_ruiz[i] * E_temp[i];

```

```

00405         }
00406         // record D_ruiz
00407         for (i = 0; i < A->m; ++i){
00408             D_ruiz[i] = D_ruiz[i] * D_temp[i];
00409         }
00410     }
00411     }
00412     abip_printf("Done the ruiz rescaling!\n");
00413 }
00414
00415 if (stgs->qp_rescale)
00416 {
00417     memset(D_temp, 0, A->m * sizeof(abip_float));
00418
00419     for (i = 0; i < A->n; ++i)
00420     {
00421         c1 = A->p[i + 1] - A->p[i];
00422         e = ABIP(norm_inf)(&(A->x[A->p[i]]), c1); // compute the max abs
00423         tmp = ABIP(min_abs_sqrt)(&(A->x[A->p[i]]), c1, e); // compute the sqrt non-zero min
00424         e = tmp * SQRTF(e);
00425
00426         // control E_qp
00427         if (e < min_col_scale){
00428             e = 1;
00429         }
00430         else if (e > max_col_scale){
00431             e = max_col_scale;
00432         }
00433
00434         // abip_printf("1.0/e is: %f\n", 1.0 / e);
00435
00436         ABIP(scale_array)(&(A->x[A->p[i]]), 1.0 / e, c1); // scale A
00437         E_qp[i] = e;
00438     }
00439
00440     // rescaling trick from QP. Find D_qp
00441     // find inf norm and min nonzero abs
00442     for (i = 0; i < A->n; ++i)
00443     {
00444         c1 = A->p[i];
00445         c2 = A->p[i + 1];
00446         for (j = c1; j < c2; ++j)
00447         {
00448             wrk = ABS(A->x[j]);
00449             if (wrk >= D_qp[A->i[j]])
00450             {
00451                 D_qp[A->i[j]] = wrk;
00452             }
00453         }
00454     }
00455
00456     // Initialize D_temp
00457     for (i = 0; i < A->m; ++i)
00458     {
00459         D_temp[i] = D_qp[i];
00460     }
00461
00462     // compute the min nonzero abs and save it in D_temp
00463     for (i = 0; i < A->n; ++i)
00464     {
00465         c1 = A->p[i];
00466         c2 = A->p[i+1];
00467         for (j = c1; j < c2; ++j)
00468         {
00469             wrk = ABS(A->x[j]);
00470             if (wrk <= D_temp[A->i[j]] && wrk > 0){
00471                 D_temp[A->i[j]] = wrk;
00472             }
00473         }
00474     }
00475
00476     // find and control D_qp
00477
00478     for (i = 0; i < A->m; ++i)
00479     {
00480         D_qp[i] = SQRTF(D_qp[i] * D_temp[i]);
00481         if (D_qp[i] < min_row_scale)
00482         {
00483             D_qp[i] = 1;
00484         }
00485         else if (D_qp[i] > max_row_scale)
00486         {
00487             D_qp[i] = max_row_scale;
00488         }
00489     }
00490
00491     for (i = 0; i < A->n; ++i)

```



```

00492     {
00493         for (j = A->p[i]; j < A->p[i + 1]; ++j)
00494         {
00495             A->x[j] /= D_qp[A->i[j]];
00496         }
00497     }
00498     abip_printf("Done the QP rescaling\n");
00499 }
00500 else
00501 {
00502     for (i = 0; i < A->m; ++i){
00503         D_qp[i] = 1.0;
00504     }
00505     for (i = 0; i < A->n; ++i){
00506         E_qp[i] = 1.0;
00507     }
00508 }
00509 }
00510
00511 // combine the above rescaling tricks
00512 for (i = 0; i < A->m; ++i)
00513 {
00514     D[i] = D_pc[i] * D_ruiz[i] * D_origin[i] * D_qp[i];
00515 }
00516
00517 for (i = 0; i < A->n; ++i)
00518 {
00519     E[i] = E_pc[i] * E_ruiz[i] * E_origin[i] * E_qp[i];
00520 }
00521 // -----
00522
00523 for (i = 0; i < A->n; ++i)
00524 {
00525     for (j = A->p[i]; j < A->p[i + 1]; ++j)
00526     {
00527         wrk = A->x[j];
00528         nms[A->i[j]] += wrk * wrk;
00529     }
00530 }
00531
00532 scal->mean_norm_row_A = 0.0;
00533 for (i = 0; i < A->m; ++i)
00534 {
00535     scal->mean_norm_row_A += SQRTF(nms[i]) / A->m;
00536 }
00537
00538 abip_free(nms);
00539
00540 scal->mean_norm_col_A = 0.0;
00541 for (i = 0; i < A->n; ++i)
00542 {
00543     c1 = A->p[i + 1] - A->p[i];
00544     scal->mean_norm_col_A += ABIP(norm) (&(A->x[A->p[i]]), c1) / A->n;
00545 }
00546
00547 if (stgs->scale != 1)
00548 {
00549     ABIP(scale_array) (A->x, stgs->scale, A->p[A->n]);
00550 }
00551
00552 scal->D = D;
00553 scal->E = E;
00554
00555 abip_free(nms);
00556 abip_free(D_pc); abip_free(E_pc); abip_free(D_origin);
00557 abip_free(E_origin); abip_free(D_temp); abip_free(E_temp);
00558 abip_free(D_ruiz); abip_free(E_ruiz); abip_free(D_qp); abip_free(E_qp);
00559
00560 #if EXTRA_VERBOSE > 0
00561     abip_printf("finished normalizing A, time: %1.2e seconds. \n", ABIP(tocq)(&normalize_timer) /
00562         1e3);
00563     print_A_matrix(A);
00564 #endif
00565 }
00566 void ABIP(_un_normalize_A)
00567 {
00568     ABIPMatrix *A,
00569     const ABIPSettings *stgs,
00570     const ABIPScaling *scal
00571 }
00572 {
00573     abip_int i;
00574     abip_int j;
00575
00576     abip_float *D = scal->D;
00577     abip_float *E = scal->E;

```

```

00581
00582     for (i = 0; i < A->n; ++i)
00583     {
00584         for (j = A->p[i]; j < A->p[i + 1]; ++j)
00585         {
00586             A->x[j] *= D[A->i[j]];
00587         }
00588     }
00589
00590     for (i = 0; i < A->n; ++i)
00591     {
00592         ABIP(scale_array) (&(A->x[A->p[i]]), E[i] / stgs->scale, A->p[i + 1] - A->p[i]);
00593     }
00594 }
00598 void ABIP(_accum_by_Atrans)
00599 (
00600     abip_int n,
00601     abip_float *Ax,
00602     abip_int *Ai,
00603     abip_int *Ap,
00604     const abip_float *x,
00605     abip_float *y
00606 )
00607 {
00608     abip_int p;
00609     abip_int j;
00610
00611     abip_int c1;
00612     abip_int c2;
00613     abip_float yj;
00614
00615     #if EXTRA_VERBOSE > 0
00616         ABIP(timer) mult_by_Atrans_timer;
00617         ABIP(tic) (&mult_by_Atrans_timer);
00618     #endif
00619
00620     #ifdef _OPENMP
00621     #pragma omp parallel for private(p, c1, c2, yj)
00622     #endif
00623
00624     for (j = 0; j < n; j++)
00625     {
00626         yj = y[j];
00627         c1 = Ap[j];
00628         c2 = Ap[j + 1];
00629         for (p = c1; p < c2; p++)
00630         {
00631             yj += Ax[p] * x[Ai[p]];
00632         }
00633         y[j] = yj;
00634     }
00635
00636     #if EXTRA_VERBOSE > 0
00637     abip_printf("mult By A trans time: %1.2e seconds. \n", ABIP(tocq) (&mult_by_Atrans_timer) / 1e3);
00638     #endif
00639 }
00640
00644 void ABIP(_accum_by_A)
00645 (
00646     abip_int n,
00647     abip_float *Ax,
00648     abip_int *Ai,
00649     abip_int *Ap,
00650     const abip_float *x,
00651     abip_float *y
00652 )
00653 {
00654     abip_int p;
00655     abip_int j;
00656
00657     abip_int c1;
00658     abip_int c2;
00659     abip_float xj;
00660
00661     #if EXTRA_VERBOSE > 0
00662         ABIP(timer) mult_by_A_timer;
00663         ABIP(tic) (&mult_by_A_timer);
00664     #endif
00665
00666     #ifdef _OPENMP
00667     #pragma omp parallel for private(p, c1, c2, xj)
00668     #endif
00669
00670     for (j = 0; j < n; j++)
00671     {
00672         xj = x[j];
00673         c1 = Ap[j];
00674         c2 = Ap[j + 1];
00675         for (p = c1; p < c2; p++)

```

```

00674     {
00675 #pragma omp atomic
00676     y[Ai[p]] += Ax[p] * xj;
00677     }
00678 }
00679 #endif
00680
00681     for (j = 0; j < n; j++)
00682     {
00683         xj = x[j];
00684         c1 = Ap[j];
00685         c2 = Ap[j + 1];
00686         for (p = c1; p < c2; p++)
00687         {
00688             y[Ai[p]] += Ax[p] * xj;
00689         }
00690     }
00691
00692 #if EXTRA_VERBOSE > 0
00693     abip_printf("mult By A time: %1.2e seconds \n", ABIP(tocq) (&mult_by_A_timer) / 1e3);
00694 #endif
00695 }
00696
00700 abip_float ABIP(cumsum)
00701 (
00702     abip_int *p,
00703     abip_int *c,
00704     abip_int n
00705 )
00706 {
00707     abip_int i;
00708     abip_float nz = 0;
00709     abip_float nz2 = 0;
00710
00711     if (!p || !c)
00712     {
00713         return (-1);
00714     }
00715
00716     for (i = 0; i < n; i++)
00717     {
00718         p[i] = nz;
00719         nz += c[i];
00720         nz2 += c[i];
00721         c[i] = p[i];
00722     }
00723
00724     p[n] = nz;
00725     return nz2;
00726 }

```

5.76 linsys/common.h File Reference

```

#include "abip.h"
#include "amatrix.h"
#include "linsys.h"
#include "linalg.h"
#include "util.h"

```

Functions

- void **ABIP()** **_accum_by_Atrans** (abip_int n, abip_float *Ax, abip_int *Ai, abip_int *Ap, const abip_float *x, abip_float *y)
compute $A^T x$
- void **ABIP()** **_accum_by_A** (abip_int n, abip_float *Ax, abip_int *Ai, abip_int *Ap, const abip_float *x, abip_float *y)
compute Ax
- void **ABIP()** **_normalize_A** (ABIPMatrix *A, const ABIPSettings *stgs, ABIPScaling *scal)
normalize matrix A

- void `ABIP()` `_un_normalize_A` (`ABIPMatrix` *`A`, const `ABIPSettings` *`stgs`, const `ABIPScaling` *`scal`)
unnormalize matrix A
- `abip_float` `ABIP()` `cumsum` (`abip_int` *`p`, `abip_int` *`c`, `abip_int` `n`)
compute cumulative sum of c

5.76.1 Function Documentation

5.76.1.1 `_accum_by_A()`

```
void ABIP() _accum_by_A (
    abip_int n,
    abip_float * Ax,
    abip_int * Ai,
    abip_int * Ap,
    const abip_float * x,
    abip_float * y )
```

compute Ax

Definition at line 644 of file `common.c`.

5.76.1.2 `_accum_by_Atrans()`

```
void ABIP() _accum_by_Atrans (
    abip_int n,
    abip_float * Ax,
    abip_int * Ai,
    abip_int * Ap,
    const abip_float * x,
    abip_float * y )
```

compute $A^T x$

Definition at line 598 of file `common.c`.

5.76.1.3 `_normalize_A()`

```
void ABIP() _normalize_A (
    ABIPMatrix * A,
    const ABIPSettings * stgs,
    ABIPScaling * scal )
```

normalize matrix A

Definition at line 150 of file `common.c`.

5.76.1.4 `_un_normalize_A()`

```
void ABIP() _un_normalize_A (
    ABIPMatrix * A,
    const ABIPSettings * stgs,
    const ABIPScaling * scal )
```

unnormalize matrix A

Definition at line 569 of file [common.c](#).

5.76.1.5 `cumsum()`

```
abip_float ABIP() cumsum (
    abip_int * p,
    abip_int * c,
    abip_int n )
```

compute cumulative sum of c

Definition at line 700 of file [common.c](#).

5.77 common.h

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

```
00001 #ifndef COMMON_H_GUARD
00002 #define COMMON_H_GUARD
00003
00004 #ifdef __cplusplus
00005 extern "C" {
00006 #endif
00007
00008 #include "abip.h"
00009 #include "amatrix.h"
00010 #include "linsys.h"
00011 #include "linalg.h"
00012 #include "util.h"
00013
00014 void ABIP(_accum_by_Atrans)
00015 (
00016     abip_int n,
00017     abip_float *Ax,
00018     abip_int *Ai,
00019     abip_int *Ap,
00020     const abip_float *x,
00021     abip_float *y
00022 );
00023
00024 void ABIP(_accum_by_A)
00025 (
00026     abip_int n,
00027     abip_float *Ax,
00028     abip_int *Ai,
00029     abip_int *Ap,
00030     const abip_float *x,
00031     abip_float *y
00032 );
00033
00034 void ABIP(_normalize_A)
00035 (
00036     ABIPMatrix *A,
00037     const ABIPSettings *stgs,
00038     ABIPScaling *scal
00039 );
00040
```

```

00041 void ABIP(_un_normalize_A)
00042 (
00043     ABIPMatrix *A,
00044     const ABIPSettings *stgs,
00045     const ABIPScaling *scal
00046 );
00047
00048 abip_float ABIP(cumsum)
00049 (
00050     abip_int *p,
00051     abip_int *c,
00052     abip_int n
00053 );
00054
00055 #ifdef __cplusplus
00056 }
00057 #endif
00058
00059 #endif

```

5.78 linsys/direct.c File Reference

```
#include "direct.h"
```

Functions

- char *ABIP() get_lin_sys_method (const ABIPMatrix *A, const ABIPSettings *stgs)
- char *ABIP() get_lin_sys_summary (ABIPLinSysWork *p, const ABIPLinSysInfo *info)
- void ABIP() free_lin_sys_work (ABIPLinSysWork *p)
- cs * form_kkt (const ABIPMatrix *A, const ABIPSettings *s)
- abip_int _ldl_init (cs *A, abip_int P[], abip_float **info)
- abip_int _ldl_factor (cs *A, abip_int P[], abip_int Pinv[], cs **L, abip_float **D)
- void _ldl_solve (abip_float *x, abip_float b[], cs *L, abip_float D[], abip_int P[], abip_float *bp)
- void ABIP() accum_by_Atrans (const ABIPMatrix *A, ABIPLinSysWork *p, const abip_float *x, abip_float *y)
- void ABIP() accum_by_A (const ABIPMatrix *A, ABIPLinSysWork *p, const abip_float *x, abip_float *y)
- void ABIP() normalize_A (ABIPMatrix *A, const ABIPSettings *stgs, ABIPScaling *scal)
- void ABIP() un_normalize_A (ABIPMatrix *A, const ABIPSettings *stgs, const ABIPScaling *scal)
- abip_int factorize (const ABIPMatrix *A, const ABIPSettings *stgs, ABIPLinSysWork *p)
- ABIPLinSysWork *ABIP() init_lin_sys_work (const ABIPMatrix *A, const ABIPSettings *stgs)
- abip_int ABIP() solve_lin_sys (const ABIPMatrix *A, const ABIPSettings *stgs, ABIPLinSysWork *p, abip_float *b, const abip_float *s, abip_int iter)

5.78.1 Function Documentation

5.78.1.1 _ldl_factor()

```

abip_int _ldl_factor (
    cs * A,
    abip_int P[],
    abip_int Pinv[],
    cs ** L,
    abip_float ** D )

```

Definition at line 121 of file [direct.c](#).

5.78.1.2 `_ldl_init()`

```
abip_int _ldl_init (
    cs * A,
    abip_int P[],
    abip_float ** info )
```

Definition at line 106 of file [direct.c](#).

5.78.1.3 `_ldl_solve()`

```
void _ldl_solve (
    abip_float * x,
    abip_float b[],
    cs * L,
    abip_float D[],
    abip_int P[],
    abip_float * bp )
```

Definition at line 172 of file [direct.c](#).

5.78.1.4 `accum_by_A()`

```
void ABIP() accum_by_A (
    const ABIPMatrix * A,
    ABIPLinSysWork * p,
    const abip_float * x,
    abip_float * y )
```

Definition at line 205 of file [direct.c](#).

5.78.1.5 `accum_by_Atrans()`

```
void ABIP() accum_by_Atrans (
    const ABIPMatrix * A,
    ABIPLinSysWork * p,
    const abip_float * x,
    abip_float * y )
```

Definition at line 200 of file [direct.c](#).

5.78.1.6 factorize()

```
abip_int factorize (
    const ABIPMatrix * A,
    const ABIPSettings * stgs,
    ABIPLinSysWork * p )
```

Definition at line 218 of file [direct.c](#).

5.78.1.7 form_kkt()

```
cs * form_kkt (
    const ABIPMatrix * A,
    const ABIPSettings * s )
```

Definition at line 49 of file [direct.c](#).

5.78.1.8 free_lin_sys_work()

```
void ABIP() free_lin_sys_work (
    ABIPLinSysWork * p )
```

Definition at line 28 of file [direct.c](#).

5.78.1.9 get_lin_sys_method()

```
char *ABIP() get_lin_sys_method (
    const ABIPMatrix * A,
    const ABIPSettings * stgs )
```

Definition at line 5 of file [direct.c](#).

5.78.1.10 get_lin_sys_summary()

```
char *ABIP() get_lin_sys_summary (
    ABIPLinSysWork * p,
    const ABIPInfo * info )
```

Definition at line 15 of file [direct.c](#).

5.78.1.11 init_lin_sys_work()

```
ABIPLinSysWork *ABIP() init_lin_sys_work (
    const ABIPMatrix * A,
    const ABIPSettings * stgs )
```

Definition at line 273 of file [direct.c](#).

5.78.1.12 normalize_A()

```
void ABIP() normalize_A (
    ABIPMatrix * A,
    const ABIPSettings * stgs,
    ABIPScaling * scal )
```

Definition at line 210 of file [direct.c](#).

5.78.1.13 solve_lin_sys()

```
abip_int ABIP() solve_lin_sys (
    const ABIPMatrix * A,
    const ABIPSettings * stgs,
    ABIPLinSysWork * p,
    abip_float * b,
    const abip_float * s,
    abip_int iter )
```

Definition at line 305 of file [direct.c](#).

5.78.1.14 un_normalize_A()

```
void ABIP() un_normalize_A (
    ABIPMatrix * A,
    const ABIPSettings * stgs,
    const ABIPScaling * scal )
```

Definition at line 214 of file [direct.c](#).

5.79 direct.c

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

```

00001 #include "direct.h"
00002
00003 // use ldl to solve the linear system
00004
00005 char *ABIP(get_lin_sys_method) ( const ABIPMatrix *A, const ABIPSettings *stgs ) {
00006     char *tmp = (char *) abip_malloc(sizeof(char) * 128);
00007     #ifdef ABIP_PARDISO
00008         sprintf(tmp, "sparse-direct-intel-pardiso, nnz in A = %li", (long)A->p[A->n]);
00009     #else
00010         sprintf(tmp, "sparse-direct, nnz in A = %li", (long)A->p[A->n]);
00011     #endif
00012     return tmp;
00013 }
00014
00015 char *ABIP(get_lin_sys_summary) ( ABIPLinSysWork *p, const ABIPInfo *info ) {
00016     char *str = (char *) abip_malloc(sizeof(char) * 128);
00017     #ifdef ABIP_PARDISO
00018         sprintf(str, "Linear system avg solve time: %1.2es\n", p->total_solve_time / (info->admm_iter + 1)
00019             / 1e3);
00019     #else
00020         abip_int n = p->L->n;
00021         sprintf(str, "\tLin-sys: nnz in L factor: %li, avg solve time: %1.2es\n",
00022             (long)(p->L->p[n] + n), p->total_solve_time / (info->admm_iter + 1) / 1e3);
00023     #endif
00024     p->total_solve_time = 0; return str;
00025 }
00026
00027 void ABIP(free_lin_sys_work) ( ABIPLinSysWork *p ) {
00028     if (p) {
00029         if (p->L) { ABIP(cs_spfree) (p->L); }
00030         if (p->P) { abip_free(p->P); }
00031         if (p->bp) { abip_free(p->bp); }
00032         if (p->D) { abip_free(p->D); }
00033         abip_free(p);
00034     }
00035 }
00036
00037 #ifdef ABIP_PARDISO
00038 void ABIP(free_lin_sys_work_pds) ( ABIPLinSysWork *p, ABIPMatrix *A ) {
00039     if (p) {
00040         pardisoFree(p, A);
00041         if (p->P) { abip_free(p->P); }
00042         if (p->D) { abip_free(p->D); }
00043         abip_free(p);
00044     }
00045 }
00046 #endif
00047
00048 cs *form_kkt ( const ABIPMatrix *A, const ABIPSettings *s ) {
00049     abip_int j, k, kk; cs *K_cs;
00050     const abip_int Annz = A->p[A->n];
00051     const abip_int Knnzmax = A->m + A->n + Annz;
00052     cs *K = ABIP(cs_spalloc) (A->m + A->n, A->m + A->n, Knnzmax, 1, 1);
00053     #if EXTRA_VERBOSE > 0
00054         abip_printf("forming KKT\n");
00055     #endif
00056     if (!K)
00057     {
00058         return ABIP_NULL;
00059     }
00060     kk = 0;
00061     for (k = 0; k < A->m; k++)
00062     {
00063         K->i[kk] = k;
00064         K->p[kk] = k;
00065         K->x[kk] = s->rho_y;
00066         kk++;
00067     }
00068     for (j = 0; j < A->n; j++)
00069     {
00070         for (k = A->p[j]; k < A->p[j + 1]; k++)
00071         {
00072             K->p[kk] = j + A->m;
00073             K->i[kk] = A->i[k];
00074             K->x[kk] = A->x[k];
00075         }
00076     }

```

```

00082         kk++;
00083     }
00084 }
00085 for (k = 0; k < A->n; k++)
00086 {
00087     K->i[kk] = k + A->m;
00088     K->p[kk] = k + A->m;
00089     K->x[kk] = -1;
00090     kk++;
00091 }
00092
00093 K->nnz = Knnzmax;
00094 K_cs = ABIP(cs_compress) (K);
00095
00096 #ifdef ABIP_PARDISO
00097     cs *KT = ABIP(cs_transpose) (K_cs, 1);
00098     ABIP(cs_spfree) (K); ABIP(cs_spfree) (K_cs);
00099     return (KT);
00100 #else
00101     ABIP(cs_spfree) (K);
00102     return (K_cs);
00103 #endif
00104 }
00105
00106 abip_int_ldl_init ( cs *A, abip_int P[], abip_float **info ) {
00107     #ifdef ABIP_PARDISO
00108         return 0;
00109     #else
00110         *info = (abip_float *)abip_malloc(AMD_INFO * sizeof(abip_float));
00111     #endif
00112     #ifdef DLONG
00113         return (amd_l_order(A->n, A->p, A->i, P, (abip_float *) ABIP_NULL, *info));
00114     #else
00115         return (amd_order(A->n, A->p, A->i, P, (abip_float *) ABIP_NULL, *info));
00116     #endif
00117 }
00118 #endif
00119 }
00120
00121 abip_int_ldl_factor ( cs *A, abip_int P[], abip_int Pinv[], cs **L, abip_float **D ) {
00122     #ifdef ABIP_PARDISO
00123         return 0;
00124     #else
00125         abip_int kk;
00126         abip_int n = A->n;
00127
00128         abip_int *Parent = (abip_int *)abip_malloc(n * sizeof(abip_int));
00129         abip_int *Lnnz = (abip_int *)abip_malloc(n * sizeof(abip_int));
00130         abip_int *Flag = (abip_int *)abip_malloc(n * sizeof(abip_int));
00131         abip_int *Pattern = (abip_int *)abip_malloc(n * sizeof(abip_int));
00132         abip_float *Y = (abip_float *)abip_malloc(n * sizeof(abip_float));
00133         (*L)->p = (abip_int *)abip_malloc((1 + n) * sizeof(abip_int));
00134
00135         /*abip_int Parent[n], Lnnz[n], Flag[n], Pattern[n]; */
00136         /*abip_float Y[n]; */
00137
00138         LDL_symbolic(n, A->p, A->i, (*L)->p, Parent, Lnnz, Flag, P, Pinv);
00139
00140         (*L)->nnzmax = ((*L)->p + n);
00141         (*L)->x = (abip_float *)abip_malloc((*L)->nnzmax * sizeof(abip_float));
00142         (*L)->i = (abip_int *)abip_malloc((*L)->nnzmax * sizeof(abip_int));
00143         *D = (abip_float *)abip_malloc(n * sizeof(abip_float));
00144
00145         if (!( *D ) || !( *L )->i || !( *L )->x || !Y || !Pattern || !Flag || !Lnnz || !Parent)
00146         {
00147             abip_free(Pattern); abip_free(Y);
00148             return -1;
00149         }
00150     #endif
00151
00152     #if EXTRA_VERBOSE > 0
00153         abip_printf("numeric factorization\n");
00154     #endif
00155
00156     kk = LDL_numeric(n, A->p, A->i, A->x, (*L)->p, Parent, Lnnz, (*L)->i, (*L)->x, *D, Y, Pattern,
00157         Flag, P, Pinv);
00158
00159     #if EXTRA_VERBOSE > 0
00160         abip_printf("finished numeric factorization\n");
00161     #endif
00162
00163     abip_free(Parent);
00164     abip_free(Lnnz);
00165     abip_free(Flag);
00166     abip_free(Pattern);
00167     abip_free(Y);
00168     return (kk - n);

```

```

00168 #endif
00169
00170 }
00171
00172 void _ldl_solve ( abip_float *x, abip_float b[], cs *L, abip_float D[],
00173                  abip_int P[], abip_float *bp ) {
00174 #ifdef ABIP_PARDISO
00175     return;
00176 #else
00177     abip_int n = L->n;
00178     if (P == ABIP_NULL)
00179     {
00180         if (x != b)
00181         {
00182             memcpy(x, b, n * sizeof(abip_float));
00183         }
00184
00185         LDL_solve(n, x, L->p, L->i, L->x);
00186         LDL_solve(n, x, D);
00187         LDL_solve(n, x, L->p, L->i, L->x);
00188     }
00189     else
00190     {
00191         LDL_perm(n, bp, b, P);
00192         LDL_solve(n, bp, L->p, L->i, L->x);
00193         LDL_solve(n, bp, D);
00194         LDL_solve(n, bp, L->p, L->i, L->x);
00195         LDL_permt(n, x, bp, P);
00196     }
00197 #endif
00198 }
00199
00200 void ABIP(accum_by_Atrans) ( const ABIPMatrix *A, ABIPLinSysWork *p,
00201                             const abip_float *x, abip_float *y ) {
00202     ABIP(_accum_by_Atrans) (A->n, A->x, A->i, A->p, x, y);
00203 }
00204
00205 void ABIP(accum_by_A) ( const ABIPMatrix *A, ABIPLinSysWork *p,
00206                        const abip_float *x, abip_float *y ) {
00207     ABIP(_accum_by_A) (A->n, A->x, A->i, A->p, x, y);
00208 }
00209
00210 void ABIP(normalize_A) ( ABIPMatrix *A, const ABIPSettings *stgs, ABIPScaling *scal ) {
00211     ABIP(_normalize_A) (A, stgs, scal);
00212 }
00213
00214 void ABIP(un_normalize_A) ( ABIPMatrix *A, const ABIPSettings *stgs, const ABIPScaling *scal ) {
00215     ABIP(_un_normalize_A) (A, stgs, scal);
00216 }
00217
00218 abip_int factorize ( const ABIPMatrix *A, const ABIPSettings *stgs, ABIPLinSysWork *p ) {
00219 #ifdef ABIP_PARDISO
00220     abip_int ret_code = 0;
00221     cs *K = form_kkt(A, stgs);
00222     if (!K) { return -1; }
00223     ret_code = pardisoFactorize(p, K);
00224     ABIP(cs_spfree)(K);
00225     return ret_code;
00226 #else
00227     abip_float *info;
00228     abip_int *Pinv;
00229     abip_int amd_status;
00230     abip_int ldl_status;
00231
00232     cs *C;
00233     cs *K = form_kkt(A, stgs);
00234     if (!K)
00235     {
00236         return -1;
00237     }
00238
00239     amd_status = _ldl_init(K, p->P, &info);
00240     if (amd_status < 0)
00241     {
00242         return (amd_status);
00243     }
00244
00245 #if EXTRA_VERBOSE > 0
00246     if (stgs->verbose)
00247     {
00248         abip_printf("Matrix factorization info:\n");
00249
00250 #ifdef DLONG
00251         amd_l_info(info);
00252     #else
00253         amd_info(info);
00254     #endif
00255     }
00256 #endif

```

```

00255 #endif
00256 }
00257 #endif
00258
00259     Pinv = ABIP(cs_pinv) (p->P, A->m + A->n);
00260     C = ABIP(cs_symperm) (K, Pinv, l);
00261     ldl_status = _ldl_factor(C, ABIP_NULL, ABIP_NULL, &p->L, &p->D);
00262
00263     ABIP(cs_spfree) (C);
00264     ABIP(cs_spfree) (K);
00265     abip_free(Pinv);
00266     abip_free(info);
00267
00268     return (ldl_status);
00269 #endif
00270 }
00271
00272
00273 ABIPLinSysWork *ABIP(init_lin_sys_work)
00274 (
00275     const ABIPMatrix *A,
00276     const ABIPSettings *stgs
00277 )
00278 {
00279     ABIPLinSysWork *p = (ABIPLinSysWork *) abip_calloc(1, sizeof(ABIPLinSysWork));
00280     abip_int m_plus_n = A->m + A->n;
00281 #ifndef ABIP_PARDISO
00282     p->P = (abip_int *) abip_malloc(sizeof(abip_int) * m_plus_n);
00283     p->D = (abip_float *) abip_malloc(sizeof(abip_float) * m_plus_n);
00284     if (factorize(A, stgs, p)) {
00285         ABIP(free_lin_sys_work) (p); return ABIP_NULL;
00286     }
00287 #else
00288     p->P = (abip_int *) abip_malloc(sizeof(abip_int) * m_plus_n);
00289     p->L = (cs *) abip_malloc(sizeof(cs));
00290     p->bp = (abip_float *) abip_malloc(m_plus_n * sizeof(abip_float));
00291     p->L->m = m_plus_n;
00292     p->L->n = m_plus_n;
00293     p->L->nnz = -1;
00294
00295     if (factorize(A, stgs, p) < 0)
00296     {
00297         ABIP(free_lin_sys_work) (p);
00298         return ABIP_NULL;
00299     }
00300 #endif
00301     p->total_solve_time = 0.0;
00302     return p;
00303 }
00304
00305 abip_int ABIP(solve_lin_sys)
00306 (
00307     const ABIPMatrix *A,
00308     const ABIPSettings *stgs,
00309     ABIPLinSysWork *p,
00310     abip_float *b,
00311     const abip_float *s,
00312     abip_int iter
00313 )
00314 {
00315     ABIP(timer) linsys_timer;
00316     ABIP(tic) (&linsys_timer);
00317 #ifndef ABIP_PARDISO
00318     pardisoSolve(p, (ABIPMatrix *) A, b);
00319 #else
00320     _ldl_solve(b, b, p->L, p->D, p->P, p->bp);
00321 #endif
00322     p->total_solve_time += ABIP(tocq) (&linsys_timer);
00323 #if EXTRA_VERBOSE > 0
00324     abip_printf("linsys solve time: %1.2es\n", ABIP(tocq) (&linsys_timer) / 1e3);
00325 #endif
00326
00327     return 0;
00328 }

```

5.80 linsys/direct.h File Reference

```

#include "glbopts.h"
#include "abip.h"
#include "cs.h"

```

```
#include "amd.h"
#include "ldl.h"
#include "common.h"
#include "abip_pardiso.h"
```

Data Structures

- struct [ABIP_LIN_SYS_WORK](#)

5.81 direct.h

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

```
00001 #ifndef PRIV_H_GUARD
00002 #define PRIV_H_GUARD
00003
00004 #ifdef __cplusplus
00005 extern "C" {
00006 #endif
00007
00008 #include "glbopts.h"
00009 #include "abip.h"
00010 #include "cs.h"
00011 #include "amd.h"
00012 #include "ldl.h"
00013 #include "common.h"
00014 #include "abip_pardiso.h"
00015
00016 struct ABIP_LIN_SYS_WORK
00017 {
00018     cs *L;
00019     abip_float *D;
00020     abip_int *P;
00021     abip_int *i;
00022     abip_int *j;
00023     abip_float *bp;
00024
00025     void *pardiso_work[PARDISOINDEX];
00026     abip_float total_solve_time;
00027 };
00028
00029 #ifdef __cplusplus
00030 }
00031 #endif
00032 #endif
```

5.82 linsys/indirect.c File Reference

```
#include "indirect.h"
```

Macros

- #define [CG_BEST_TOL](#) 1e-9
- #define [CG_MIN_TOL](#) 1e-1

Functions

- char *ABIP() [get_lin_sys_method](#) (const ABIPMatrix *A, const ABIPSettings *stgs)
- char *ABIP() [get_lin_sys_summary](#) (ABIPLinSysWork *p, const ABIPIInfo *info)
- void ABIP() [free_lin_sys_work](#) (ABIPLinSysWork *p)
- void ABIP() [accum_by_Atrans](#) (const ABIPMatrix *A, ABIPLinSysWork *p, const abip_float *x, abip_float *y)
- void ABIP() [accum_by_A](#) (const ABIPMatrix *A, ABIPLinSysWork *p, const abip_float *x, abip_float *y)
- void ABIP() [normalize_A](#) (ABIPMatrix *A, const ABIPSettings *stgs, ABIPScaling *scal)
- void ABIP() [un_normalize_A](#) (ABIPMatrix *A, const ABIPSettings *stgs, const ABIPScaling *scal)
- ABIPLinSysWork *ABIP() [init_lin_sys_work](#) (const ABIPMatrix *A, const ABIPSettings *stgs)
- abip_int ABIP() [solve_lin_sys](#) (const ABIPMatrix *A, const ABIPSettings *stgs, ABIPLinSysWork *p, abip_float *b, const abip_float *s, abip_int iter)

5.82.1 Macro Definition Documentation

5.82.1.1 CG_BEST_TOL

```
#define CG_BEST_TOL 1e-9
```

Definition at line 3 of file [indirect.c](#).

5.82.1.2 CG_MIN_TOL

```
#define CG_MIN_TOL 1e-1
```

Definition at line 4 of file [indirect.c](#).

5.82.2 Function Documentation

5.82.2.1 accum_by_A()

```
void ABIP() accum_by_A (
    const ABIPMatrix * A,
    ABIPLinSysWork * p,
    const abip_float * x,
    abip_float * y )
```

Definition at line 233 of file [indirect.c](#).

5.82.2.2 accum_by_Atrans()

```
void ABIP() accum_by_Atrans (
    const ABIPMatrix * A,
    ABIPLinSysWork * p,
    const abip_float * x,
    abip_float * y )
```

Definition at line 222 of file [indirect.c](#).

5.82.2.3 free_lin_sys_work()

```
void ABIP() free_lin_sys_work (
    ABIPLinSysWork * p )
```

Definition at line 141 of file [indirect.c](#).

5.82.2.4 get_lin_sys_method()

```
char *ABIP() get_lin_sys_method (
    const ABIPMatrix * A,
    const ABIPSettings * stgs )
```

Definition at line 8 of file [indirect.c](#).

5.82.2.5 get_lin_sys_summary()

```
char *ABIP() get_lin_sys_summary (
    ABIPLinSysWork * p,
    const ABIPInfo * info )
```

Definition at line 20 of file [indirect.c](#).

5.82.2.6 init_lin_sys_work()

```
ABIPLinSysWork *ABIP() init_lin_sys_work (
    const ABIPMatrix * A,
    const ABIPSettings * stgs )
```

Definition at line 282 of file [indirect.c](#).

5.82.2.7 normalize_A()

```
void ABIP() normalize_A (
    ABIPMatrix * A,
    const ABIPSettings * stgs,
    ABIPScaling * scal )
```

Definition at line 244 of file [indirect.c](#).

5.82.2.8 solve_lin_sys()

```
abip_int ABIP() solve_lin_sys (
    const ABIPMatrix * A,
    const ABIPSettings * stgs,
    ABIPLinSysWork * p,
    abip_float * b,
    const abip_float * s,
    abip_int iter )
```

Definition at line 393 of file [indirect.c](#).

5.82.2.9 un_normalize_A()

```
void ABIP() un_normalize_A (
    ABIPMatrix * A,
    const ABIPSettings * stgs,
    const ABIPScaling * scal )
```

Definition at line 253 of file [indirect.c](#).

5.83 indirect.c

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

```
00001 #include "indirect.h"
00002
00003 #define CG_BEST_TOL 1e-9
00004 #define CG_MIN_TOL 1e-1
00005
00006 // use cg to solve the linear system
00007
00008 char *ABIP(get_lin_sys_method)
00009 (
00010     const ABIPMatrix *A,
00011     const ABIPSettings *stgs
00012 )
00013 {
00014     char *str = (char *)abip_malloc(sizeof(char) * 128);
00015     sprintf(str, "sparse-indirect, nnz in A = %li, CG tol ~ 1/iter^(%2.2f)",
00016         (long)A->p[A->n], stgs->cg_rate);
00017     return str;
00018 }
00019
00020 char *ABIP(get_lin_sys_summary)
00021 (
00022     ABIPLinSysWork *p,
```

```

00023  const ABIPInfo *info
00024  )
00025  {
00026      char *str = (char *)abip_malloc(sizeof(char) * 128);
00027      sprintf(str, "\tLin-sys: avg # CG iterations: %2.2f, avg solve time: %1.2es\n",
00028              (abip_float)p->tot_cg_its / (info->admm_iter + 1),
00029              p->total_solve_time / (info->admm_iter + 1) / 1e3);
00030      p->tot_cg_its = 0;
00031      p->total_solve_time = 0;
00032      return str;
00033  }
00034
00035  /* M = inv(diag(RHO_Y * I + AA')) */
00036  static void get_preconditioner
00037  (
00038      const ABIPMatrix *A,
00039      const ABIPSettings *stgs,
00040      ABIPLinSysWork *p
00041  )
00042  {
00043      abip_int i;
00044      abip_int j;
00045      abip_int c1;
00046      abip_int c2;
00047
00048      abip_float wrk;
00049      abip_float *M = p->M;
00050
00051      memset(M, 0, A->m * sizeof(abip_float));
00052
00053      #if EXTRA_VERBOSE > 0
00054          abip_printf("getting pre-conditioner\n");
00055      #endif
00056
00057      for (i = 0; i < A->n; ++i)
00058      {
00059          c1 = A->p[i];
00060          c2 = A->p[i + 1];
00061          for (j = c1; j < c2; ++j)
00062          {
00063              wrk = A->x[j];
00064              M[A->i[j]] += wrk * wrk;
00065          }
00066      }
00067
00068      for (i = 0; i < A->m; ++i)
00069      {
00070          M[i] = 1 / M[i];
00071          /* M[i] = 1; */
00072      }
00073
00074      #if EXTRA_VERBOSE > 0
00075          ABIP(print_array) (M, A->m, "M");
00076          abip_printf("norm M = %4f\n", ABIP(norm) (M, A->m));
00077          abip_printf("finished getting pre-conditioner\n");
00078      #endif
00079  }
00080
00081  static void transpose
00082  (
00083      const ABIPMatrix *A,
00084      ABIPLinSysWork *p
00085  )
00086  {
00087      abip_int *Ci = p->At->i;
00088      abip_int *Cp = p->At->p;
00089      abip_float *Cx = p->At->x;
00090
00091      abip_int m = A->m;
00092      abip_int n = A->n;
00093
00094      abip_int *Ap = A->p;
00095      abip_int *Ai = A->i;
00096      abip_float *Ax = A->x;
00097
00098      abip_int i;
00099      abip_int j;
00100      abip_int q;
00101      abip_int *z;
00102      abip_int c1;
00103      abip_int c2;
00104
00105      #if EXTRA_VERBOSE > 0
00106          ABIP(timer) transpose_timer;
00107          abip_printf("transposing A\n");
00108          ABIP(tic) (&transpose_timer);
00109      #endif

```

```

00110
00111     z = (abip_int *)abip_calloc(m, sizeof(abip_int));
00112
00113     for (i = 0; i < Ap[n]; i++)
00114     {
00115         z[Ai[i]]++;          /* row counts */
00116     }
00117
00118     ABIP(cumsum)(Cp, z, m);    /* row pointers */
00119
00120     for (j = 0; j < n; j++)
00121     {
00122         c1 = Ap[j];
00123         c2 = Ap[j + 1];
00124         for (i = c1; i < c2; i++)
00125         {
00126             q = z[Ai[i]];
00127             Ci[q] = j;        /* place A(i,j) as entry C(j,i) */
00128             Cx[q] = Ax[i];
00129             z[Ai[i]]++;
00130         }
00131     }
00132
00133     abip_free(z);
00134
00135     #if EXTRA_VERBOSE > 0
00136     abip_printf("finished transposing A, time: %1.2es\n",
00137                ABIP(tocq)(&transpose_timer) / 1e3);
00138     #endif
00139 }
00140
00141 void ABIP(free_lin_sys_work)
00142 {
00143     ABIPLinSysWork *p
00144 }
00145 {
00146     if (p)
00147     {
00148         if (p->p)
00149         {
00150             abip_free(p->p);
00151         }
00152
00153         if (p->r)
00154         {
00155             abip_free(p->r);
00156         }
00157
00158         if (p->Gp)
00159         {
00160             abip_free(p->Gp);
00161         }
00162
00163         if (p->tmp)
00164         {
00165             abip_free(p->tmp);
00166         }
00167
00168         if (p->At)
00169         {
00170             ABIPMatrix *At = p->At;
00171
00172             if (At->i)
00173             {
00174                 abip_free(At->i);
00175             }
00176
00177             if (At->x)
00178             {
00179                 abip_free(At->x);
00180             }
00181
00182             if (At->p)
00183             {
00184                 abip_free(At->p);
00185             }
00186
00187             abip_free(At);
00188         }
00189
00190         if (p->z)
00191         {
00192             abip_free(p->z);
00193         }
00194
00195         if (p->M)
00196     {

```

```

00197         abip_free(p->M);
00198     }
00199
00200     abip_free(p);
00201 }
00202 }
00203
00204 /*y = (RHO_Y * I + AA')x */
00205 static void mat_vec
00206 (
00207     const ABIPMatrix *A,
00208     const ABIPSettings *s,
00209     ABIPLinSysWork *p,
00210     const abip_float *x,
00211     abip_float *y
00212 )
00213 {
00214     abip_float *tmp = p->tmp;
00215     memset(tmp, 0, A->n * sizeof(abip_float));
00216     ABIP(accum_by_Atrans)(A, p, x, tmp);
00217     memset(y, 0, A->m * sizeof(abip_float));
00218     ABIP(accum_by_A)(A, p, tmp, y);
00219     ABIP(add_scaled_array)(y, x, A->m, s->rho_y);
00220 }
00221
00222 void ABIP(accum_by_Atrans)
00223 (
00224     const ABIPMatrix *A,
00225     ABIPLinSysWork *p,
00226     const abip_float *x,
00227     abip_float *y
00228 )
00229 {
00230     ABIP(_accum_by_Atrans)(A->n, A->x, A->i, A->p, x, y);
00231 }
00232
00233 void ABIP(accum_by_A)
00234 (
00235     const ABIPMatrix *A,
00236     ABIPLinSysWork *p,
00237     const abip_float *x,
00238     abip_float *y
00239 )
00240 {
00241     ABIP(_accum_by_Atrans)(p->At->n, p->At->x, p->At->i, p->At->p, x, y);
00242 }
00243
00244 void ABIP(normalize_A)
00245 (
00246     ABIPMatrix *A,
00247     const ABIPSettings *stgs,
00248     ABIPScaling *scal)
00249 {
00250     ABIP(_normalize_A)(A, stgs, scal);
00251 }
00252
00253 void ABIP(un_normalize_A)
00254 (
00255     ABIPMatrix *A,
00256     const ABIPSettings *stgs,
00257     const ABIPScaling *scal
00258 )
00259 {
00260     ABIP(_un_normalize_A)(A, stgs, scal);
00261 }
00262
00263 static void apply_pre_conditioner
00264 (
00265     abip_float *M,
00266     abip_float *z,
00267     abip_float *r,
00268     abip_int m,
00269     abip_float *ipzr
00270 )
00271 {
00272     abip_int i;
00273     *ipzr = 0;
00274
00275     for (i = 0; i < m; ++i)
00276     {
00277         z[i] = r[i] * M[i];
00278         *ipzr += z[i] * r[i];
00279     }
00280 }
00281
00282 ABIPLinSysWork *ABIP(init_lin_sys_work)
00283 (

```

```

00284  const ABIPMatrix *A,
00285  const ABIPSettings *stgs
00286  )
00287  {
00288      ABIPLinSysWork *p = (ABIPLinSysWork *)abip_calloc(1, sizeof(ABIPLinSysWork));
00289      p->p = (abip_float *)abip_malloc((A->m) * sizeof(abip_float));
00290      p->r = (abip_float *)abip_malloc((A->m) * sizeof(abip_float));
00291      p->Gp = (abip_float *)abip_malloc((A->m) * sizeof(abip_float));
00292      p->tmp = (abip_float *)abip_malloc((A->n) * sizeof(abip_float));
00293
00294      /* memory for A transpose */
00295      p->At = (ABIPMatrix *)abip_malloc(sizeof(ABIPMatrix));
00296      p->At->m = A->n;
00297      p->At->n = A->m;
00298      p->At->i = (abip_int *)abip_malloc((A->p[A->n]) * sizeof(abip_int));
00299      p->At->p = (abip_int *)abip_malloc((A->m + 1) * sizeof(abip_int));
00300      p->At->x = (abip_float *)abip_malloc((A->p[A->n]) * sizeof(abip_float));
00301      transpose(A, p);
00302
00303      /* preconditioner memory */
00304      p->z = (abip_float *)abip_malloc((A->m) * sizeof(abip_float));
00305      p->M = (abip_float *)abip_malloc((A->m) * sizeof(abip_float));
00306      get_preconditioner(A, stgs, p);
00307
00308      p->total_solve_time = 0;
00309      p->tot_cg_its = 0;
00310
00311      if (!p->p || !p->r || !p->Gp || !p->tmp || !p->At || !p->At->i || !p->At->p || !p->At->x)
00312      {
00313          ABIP(free_lin_sys_work)(p);
00314          return ABIP_NULL;
00315      }
00316
00317      return p;
00318  }
00319
00320  /* solves (I+AA')x = b, s warm start, solution stored in b */
00321  static abip_int pcg
00322  (
00323      const ABIPMatrix *A,
00324      const ABIPSettings *stgs,
00325      ABIPLinSysWork *pr,
00326      const abip_float *s,
00327      abip_float *b,
00328      abip_int max_its,
00329      abip_float tol
00330  )
00331  {
00332      abip_int i;
00333      abip_int m = A->m;
00334
00335      abip_float ipzr;
00336      abip_float ipzr_old;
00337      abip_float alpha;
00338
00339      abip_float *p = pr->p; /* cg direction */
00340      abip_float *Gp = pr->Gp; /* updated CG direction */
00341      abip_float *r = pr->r; /* cg residual */
00342      abip_float *z = pr->z; /* for preconditioning */
00343      abip_float *M = pr->M; /* inverse diagonal preconditioner */
00344
00345      if (s == ABIP_NULL)
00346      {
00347          memcpy(r, b, m * sizeof(abip_float));
00348          memset(b, 0, m * sizeof(abip_float));
00349      }
00350      else
00351      {
00352          mat_vec(A, stgs, pr, s, r);
00353          ABIP(add_scaled_array)(r, b, m, -1);
00354          ABIP(scale_array)(r, -1, m);
00355          memcpy(b, s, m * sizeof(abip_float));
00356      }
00357
00358      /* check to see if we need to run CG at all */
00359      if (ABIP(norm)(r, m) < MIN(tol, 1e-18))
00360      {
00361          return 0;
00362      }
00363
00364      apply_pre_conditioner(M, z, r, m, &ipzr);
00365      memcpy(p, z, m * sizeof(abip_float));
00366
00367      /* main loop */
00368      for (i = 0; i < max_its; ++i)
00369      {
00370          mat_vec(A, stgs, pr, p, Gp);

```

```

00371     alpha = ipzr / ABIP(dot)(p, Gp, m);
00372     ABIP(add_scaled_array)(b, p, m, alpha);
00373     ABIP(add_scaled_array)(r, Gp, m, -alpha);
00374
00375     if (ABIP(norm)(r, m) < tol)
00376     {
00377 #if EXTRA_VERBOSE > 0
00378         abip_printf("tol: %.4e, resid: %.4e, iters: %li\n", tol, ABIP(norm)(r, m), (long)i + 1);
00379 #endif
00380         return i + 1;
00381     }
00382
00383     ipzr_old = ipzr;
00384     apply_pre_conditioner(M, z, r, m, &ipzr);
00385     ABIP(scale_array)(p, ipzr / ipzr_old, m);
00386     ABIP(add_scaled_array)(p, z, m, 1);
00387
00388 }
00389
00390 return i;
00391 }
00392
00393 abip_int ABIP(solve_lin_sys)
00394 {
00395     const ABIPMatrix *A,
00396     const ABIPSettings *stgs,
00397     ABIPLinSysWork *p,
00398     abip_float *b,
00399     const abip_float *s,
00400     abip_int iter
00401 }
00402 {
00403     abip_int cg_its;
00404     ABIP(timer) linsys_timer;
00405
00406     abip_float cg_tol = ABIP(norm)(b, A->m) * (iter < 0 ? CG_BEST_TOL
00407                                         : CG_MIN_TOL / POWF((abip_float)iter + 1,
00408 stgs->cg_rate));
00409
00410     cg_tol = MAX(cg_tol, 1e-07);
00411
00412     ABIP(tic)(&linsys_timer);
00413
00414     /* solves Mx = b, for x but stores result in b */
00415     /* s contains warm-start (if available) */
00416     ABIP(accum_by_A)(A, p, &(b[A->m]), b);
00417
00418     /* solves (I+AA')x = b, s warm start, solution stored in b */
00419     cg_its = pcg(A, stgs, p, s, b, A->m, MAX(cg_tol, CG_BEST_TOL));
00420     ABIP(scale_array)(&(b[A->m]), -1, A->n);
00421     ABIP(accum_by_Atrans)(A, p, b, &(b[A->m]));
00422
00423     if (iter >= 0)
00424     {
00425         p->tot_cg_its += cg_its;
00426     }
00427
00428     p->total_solve_time += ABIP(tocq)(&linsys_timer);
00429
00430 #if EXTRA_VERBOSE > 0
00431     abip_printf("linsys solve time: %1.2es\n", ABIP(tocq)(&linsys_timer) / 1e3);
00432 #endif
00433
00434     return 0;
00435 }

```

5.84 linsys/indirect.h File Reference

```

#include <math.h>
#include "common.h"
#include "glbopts.h"
#include "linalg.h"
#include "abip.h"

```

Data Structures

- struct [ABIP_LIN_SYS_WORK](#)

5.85 indirect.h

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

```

00001 #ifndef PRIV_H_GUARD
00002 #define PRIV_H_GUARD
00003
00004 #ifdef __cplusplus
00005 extern "C" {
00006 #endif
00007
00008 #include <math.h>
00009 #include "common.h"
00010 #include "glbopts.h"
00011 #include "linalg.h"
00012 #include "abip.h"
00013
00014 struct ABIP_LIN_SYS_WORK
00015 {
00016     abip_float *p; /* cg iterate */
00017     abip_float *r; /* cg residual */
00018     abip_float *Gp;
00019     abip_float *tmp;
00020     ABIPMatrix *At;
00021
00022     /* preconditioning */
00023     abip_float *z;
00024     abip_float *M;
00025
00026     /* reporting */
00027     abip_int tot_cg_its;
00028     abip_float total_solve_time;
00029 };
00030
00031 #ifdef __cplusplus
00032 }
00033 #endif
00034 #endif

```

5.86 make_abip.m File Reference

Typedefs

- using `int` = false

Functions

- `if (~isempty(strfind(computer, '64')))` `flags.arr`
- `end if (isunix && ~ismac)` `flags.link`
- `elseif (ismac)` `flags.link`
- `end if (float)` `flags.LCFLAG`
- `end if (int)` `flags.INT`
- `end if (flags.COMPILE_WITH_OPENMP)` `flags.link`
- `end delete (fullfile(".", "interface", "*. "+mexext))`
- `compile_direct (flags, common_abip)`
- `compile_indirect (flags, common_abip)`
- `movefile (fullfile(".", "*. "+mexext), fullfile(".", "interface"))`
- `addpath (fullfile("mexfile"))`
- `addpath (fullfile("interface"))`
- `addpath (fullfile("test"))`

Variables

- parameter setting for compiling the [ABIP](#) solver `gpu = false`
- compile the `gpu version` of [ABIP](#) `float = false`
- use bit integers for indexing [WARNING](#)
- use bit integers for indexing use with caution openmp parallelizes the matrix multiply for the indirect solver(using CG) flags [EXTRA_VERBOSE](#) = 0
- flags [BLASLIB](#) = ""
- MATLAB_MEX_FILE env variable sets blasint to ptrdiff_t flags [LCFLAG](#) = '-DMATLAB_MEX_FILE -DUSE↵_LAPACK -DCTRLC=1 -DCOPYMATRIX'
- flags [INCS](#) = ""
- flags [LOCS](#) = ""
- Common source files [abip_common_src](#) = ["linalg.c"
- [adaptive c](#)
- [abip_common_linsys](#) = ["common.c"]
- [abip_mexfile](#) = ["abip_mex.c"]
- [common_abip](#) = strcat([abip_common_src](#), " ", [abip_common_linsys](#), " ", [abip_mexfile](#))
- else flags [arr](#) = ""
- else flags [link](#) = '-lut'
- else flags [INT](#) = '-DDLONG'

5.86.1 Typedef Documentation

5.86.1.1 int

using `int` = false

Definition at line 4 of file [make_abip.m](#).

5.86.2 Function Documentation

5.86.2.1 addpath() [1/3]

```
addpath (
    fullfile("interface") )
```

5.86.2.2 addpath() [2/3]

```
addpath (
    fullfile("mexfile") )
```


5.86.2.3 addpath() [3/3]

```
addpath (
    fullfile("test") )
```

5.86.2.4 compile_direct()

```
compile_direct (
    flags ,
    common_abip )
```

5.86.2.5 compile_indirect()

```
compile_indirect (
    flags ,
    common_abip )
```

5.86.2.6 delete()

```
end delete (
    fullfile(".", "interface", "*.+mexext") )
```

5.86.2.7 elseif()

```
elseif (
    ismac )
```

5.86.2.8 if() [1/5]

```
end if (
    flags. COMPILER_WITH_OPENMP )
```

5.86.2.9 if() [2/5]

```
end if (
    float )
```

5.86.2.10 `if()` [3/5]

```
end if (
    int )
```

5.86.2.11 `if()` [4/5]

```
end if (
    isunix &&~ ismac )
```

5.86.2.12 `if()` [5/5]

```
if (
    ~ isemptystrfind(computer, '64') )
```

5.86.2.13 `movefile()`

```
movefile (
    fullfile(".", ".*"+mexext) ,
    fullfile(".", "interface") )
```

5.86.3 Variable Documentation

5.86.3.1 `abip_common_linsys`

```
abip_common_linsys = ["common.c"]
```

Definition at line 21 of file [make_abip.m](#).

5.86.3.2 `abip_common_src`

```
abip_common_src = ["linalg.c"]
```

Definition at line 19 of file [make_abip.m](#).

5.86.3.3 abip_mexfile

```
abip_mexfile = ["abip_mex.c"]
```

Definition at line 22 of file [make_abip.m](#).

5.86.3.4 arr

```
else flags arr = ''
```

Definition at line 34 of file [make_abip.m](#).

5.86.3.5 BLASLIB

```
flags BLASLIB = ''
```

Definition at line 11 of file [make_abip.m](#).

5.86.3.6 c

```
abip_version c
```

Definition at line 19 of file [make_abip.m](#).

5.86.3.7 common_abip

```
common_abip = strcat(abip_common_src, " ", abip_common_linsys, " ", abip_mexfile)
```

Definition at line 29 of file [make_abip.m](#).

5.86.3.8 EXTRA_VERBOSE

```
use bit integers for indexing use with caution openmp parallelizes the matrix multiply for the  
indirect solver (using CG) flags EXTRA_VERBOSE = 0
```

Definition at line 9 of file [make_abip.m](#).

5.86.3.9 float

compile the `gpu version` of `ABIP` `float = false`

Definition at line 3 of file [make_abip.m](#).

5.86.3.10 gpu

parameter setting for compiling the `ABIP` solver `gpu = false`

Definition at line 2 of file [make_abip.m](#).

5.86.3.11 INCS

`flags INCS = ''`

Definition at line 14 of file [make_abip.m](#).

5.86.3.12 INT

`else flags INT = '-DDLONG'`

Definition at line 53 of file [make_abip.m](#).

5.86.3.13 LCFLAG

`flags LCFLAG = '-DMATLAB_MEX_FILE -DUSE_LAPACK -DCTRLC=1 -DCOPYMATRIX'`

Definition at line 13 of file [make_abip.m](#).

5.86.3.14 link

`else flags link = '-lut'`

Definition at line 42 of file [make_abip.m](#).

5.86.3.15 LOCS

```
flags LOCS = ''
```

Definition at line 15 of file [make_abip.m](#).

5.86.3.16 WARNING

```
use bit integers for indexing WARNING
```

Definition at line 6 of file [make_abip.m](#).

5.87 make_abip.m

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

```
00001 %% parameter setting for compiling the ABIP solver.
00002 gpu = false;           % compile the gpu version of ABIP.
00003 float = false;         % using single precision (rather than double) floating points
00004 int = false;           % use 32 bit integers for indexing
00005
00006 % WARNING: openmp used in MATLAB can cause errors and crashes, use with caution.
00007 % openmp parallelizes the matrix multiply for the indirect solver (using CG):
00008 flags.COMPILE_WITH_OPENMP = false;
00009 flags.EXTRA_VERBOSE = 0;
00010
00011 flags.BLASLIB = "";
00012 % MATLAB_MEX_FILE env variable sets blasint to ptrdiff_t.
00013 flags.LCFLAG = '-DMATLAB_MEX_FILE -DUSE_LAPACK -DCTRLC=1 -DCOPYMATRIX';
00014 flags.INCS = "";
00015 flags.LOCS = "";
00016
00017
00018 % Common source files
00019 abip_common_src = ["linalg.c"; "adaptive.c"; "cs.c"; "util.c"; "abip.c";
00020                   "ctrlc.c"; "normalize.c"; "abip_version.c"];
00021 abip_common_linsys = ["common.c"];
00022 abip_mexfile = ["abip_mex.c"];
00023
00024 abip_common_src = fullfile('src', abip_common_src);
00025 abip_common_src = strjoin(abip_common_src);
00026 abip_common_linsys = fullfile('linsys', abip_common_linsys);
00027 abip_mexfile = fullfile('mexfile', abip_mexfile);
00028
00029 common_abip = strcat(abip_common_src, " ", abip_common_linsys, " ", abip_mexfile);
00030
00031 if (~isempty (strfind (computer, '64')))
00032     flags.arr = '-largeArrayDims';
00033 else
00034     flags.arr = "";
00035 end
00036
00037 if (isunix && ~ismac)
00038     flags.link = '-lm -lut -lrt';
00039 elseif (ismac)
00040     flags.link = '-lm -lut';
00041 else
00042     flags.link = '-lut';
00043     flags.LCFLAG = sprintf('-DNOBLASSUFFIX %s', flags.LCFLAG);
00044 end
00045
00046 if (float)
00047     flags.LCFLAG = sprintf('-DSFLOAT %s', flags.LCFLAG);
00048 end
00049
00050 if (int)
00051     flags.INT = "";
00052 else
00053     flags.INT = '-DDLONG';
00054 end
00055
```

```

00056 if (flags.COMPILE_WITH_OPENMP)
00057     flags.link = strcat(flags.link, ' -lgomp');
00058 end
00059
00060 if (flags.EXTRA_VERBOSE)
00061     flags.LCFLAG = sprintf('-DEXTRA_VERBOSE %s', flags.LCFLAG);
00062 end
00063
00064 delete(fullfile(".", "interface", "*" + mexext));
00065 compile_direct(flags, common_abip);
00066 compile_indirect(flags, common_abip);
00067 movefile(fullfile(".", "*" + mexext), fullfile(".", "interface"));
00068 addpath(fullfile("mexfile"));
00069 addpath(fullfile("interface"));
00070 % addpath(fullfile("test"));
00071 savepath

```

5.88 mexfile/abip_mex.c File Reference

```

#include "glbopts.h"
#include "linalg.h"
#include "amatrix.h"
#include "matrix.h"
#include "mex.h"
#include "abip.h"
#include "util.h"

```

Functions

- void [free_mex](#) ([ABIPData](#) *d)
set memory free
- [abip_int](#) [parse_warm_start](#) (const mxArray *p_mex, [abip_float](#) **p, [abip_int](#) len)
- [abip_int](#) * [cast_to_abip_int_arr](#) (mwIndex *arr, [abip_int](#) len)
- void [set_output_field](#) (mxArray **pout, [abip_float](#) *out, [abip_int](#) len)
- void [mexFunction](#) (int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[])
obtain data from mex function

5.88.1 Function Documentation

5.88.1.1 cast_to_abip_int_arr()

```

abip_int * cast_to_abip_int_arr (
    mwIndex * arr,
    abip_int len )

```

Definition at line 31 of file [abip_mex.c](#).

5.88.1.2 free_mex()

```
void free_mex (
    ABIPData * d )
```

set memory free

Definition at line 429 of file [abip_mex.c](#).

5.88.1.3 mexFunction()

```
void mexFunction (
    int nlhs,
    mxArray * plhs[],
    int nrhs,
    const mxArray * prhs[] )
```

obtain data from mex function

Definition at line 83 of file [abip_mex.c](#).

5.88.1.4 parse_warm_start()

```
abip_int parse_warm_start (
    const mxArray * p_mex,
    abip_float ** p,
    abip_int len )
```

Definition at line 11 of file [abip_mex.c](#).

5.88.1.5 set_output_field()

```
void set_output_field (
    mxArray ** pout,
    abip_float * out,
    abip_int len )
```

Definition at line 67 of file [abip_mex.c](#).

5.89 abip_mex.c

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

```

00001 #include "glbopts.h"
00002 #include "linalg.h"
00003 #include "amatrix.h"
00004 #include "matrix.h"
00005 #include "mex.h"
00006 #include "abip.h"
00007 #include "util.h"
00008
00009 void free_mex(ABIPData *d);
00010
00011 abip_int parse_warm_start(const mxArray *p_mex, abip_float **p, abip_int len)
00012 {
00013     *p = (abip_float *)abip_calloc(len, sizeof(abip_float));
00014     if (p_mex == ABIP_NULL)
00015     {
00016         return 0;
00017     }
00018     else if (mxIsSparse(p_mex) || (abip_int) *mxGetDimensions(p_mex) != len)
00019     {
00020         abip_printf("Error in warm-start (the input vectors should be dense and of correct size),
00021 running without full warm-start");
00022         return 0;
00023     }
00024     else
00025     {
00026         memcpy(*p, mxGetPr(p_mex), len * sizeof(abip_float));
00027         return 1;
00028     }
00029 }
00030 #if !(DLONG > 0)
00031 abip_int *cast_to_abip_int_arr(mwIndex *arr, abip_int len)
00032 {
00033     abip_int i;
00034     abip_int *arr_out = (abip_int *)abip_malloc(sizeof(abip_int) * len);
00035     for (i = 0; i < len; i++)
00036     {
00037         arr_out[i] = (abip_int) arr[i];
00038     }
00039     return arr_out;
00040 }
00041 #endif
00042
00043 #if SFLOAT > 0
00044 abip_float *cast_to_abip_float_arr(double *arr, abip_int len)
00045 {
00046     abip_int i;
00047     abip_float *arr_out = (abip_float *)abip_malloc(sizeof(abip_float) * len);
00048     for (i = 0; i < len; i++)
00049     {
00050         arr_out[i] = (abip_float) arr[i];
00051     }
00052     return arr_out;
00053 }
00054
00055 double *cast_to_double_arr(abip_float *arr, abip_int len)
00056 {
00057     abip_int i;
00058     double *arr_out = (double *)abip_malloc(sizeof(double) * len);
00059     for (i = 0; i < len; i++)
00060     {
00061         arr_out[i] = (double) arr[i];
00062     }
00063     return arr_out;
00064 }
00065 #endif
00066
00067 void set_output_field(mxArray **pout, abip_float *out, abip_int len)
00068 {
00069     *pout = mxCreateDoubleMatrix(0, 0, mxREAL);
00070     #if SFLOAT > 0
00071         mxSetPr(*pout, cast_to_double_arr(out, len));
00072         abip_free(out);
00073     #else
00074         mxSetPr(*pout, out);
00075     #endif
00076     mxSetM(*pout, len);
00077     mxSetN(*pout, 1);
00078 }
00079
00083 void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[])
00084 {

```



```

00085     abip_int i;
00086     abip_int status;
00087
00088     ABIPData *d;
00089     ABIPSolution sol = {0};
00090     ABIPInfo info;
00091     ABIPMatrix *A;
00092
00093     const mxArray *data;
00094     const mxArray *A_mex;
00095     const mxArray *b_mex;
00096     const mxArray *c_mex;
00097
00098     const mxArray *settings;
00099
00100     const mwSize one[1] = {1};
00101     const int num_info_fields = 13;
00102     const char *info_fields[] = {"status", "ipm_iter", "admm_iter", "mu", "pobj", "dobj", "resPri",
"resDual", "relGap", "resInfeas", "resUnbdd", "setupTime", "solveTime"};
00103
00104     mxArray *tmp;
00105
00106     #if EXTRA_VERBOSE > 0
00107         abip_printf("size of mwSize = %i\n", (int) sizeof(mwSize));
00108         abip_printf("size of mwIndex = %i\n", (int) sizeof(mwIndex));
00109     #endif
00110
00111     if (nrhs != 2)
00112     {
00113         mexErrMsgTxt("Inputs are required in this order: data struct, settings struct");
00114     }
00115
00116     if (nlhs > 4)
00117     {
00118         mexErrMsgTxt("ABIP returns up to 4 output arguments only.");
00119     }
00120
00121     d = (ABIPData *)mxMalloc(sizeof(ABIPData));
00122     d->stgs = (ABIPSettings *)mxMalloc(sizeof(ABIPSettings));
00123     data = prhs[0];
00124
00125     A_mex = (mxArray *)mxGetField(data, 0, "A");
00126
00127     if (A_mex == ABIP_NULL)
00128     {
00129         abip_free(d);
00130         mexErrMsgTxt("ABIPData struct must contain a matrix 'A'.");
00131     }
00132
00133     if (!mxIsSparse(A_mex))
00134     {
00135         abip_free(d);
00136         mexErrMsgTxt("Input matrix A must be in sparse format.");
00137     }
00138
00139     b_mex = (mxArray *)mxGetField(data, 0, "b");
00140
00141     if (b_mex == ABIP_NULL)
00142     {
00143         abip_free(d);
00144         mexErrMsgTxt("ABIPData struct must contain a vector 'b'.");
00145     }
00146
00147     if (mxIsSparse(b_mex))
00148     {
00149         abip_free(d);
00150         mexErrMsgTxt("Input vector b must be in dense format.");
00151     }
00152
00153     c_mex = (mxArray *)mxGetField(data, 0, "c");
00154
00155     if (c_mex == ABIP_NULL)
00156     {
00157         abip_free(d);
00158         mexErrMsgTxt("ABIPData struct must contain a vector 'c'.");
00159     }
00160
00161     if (mxIsSparse(c_mex))
00162     {
00163         abip_free(d);
00164         mexErrMsgTxt("Input vector c must be in dense format.");
00165     }
00166
00167     settings = prhs[1];
00168
00169     d->n = (abip_int) * (mxGetDimensions(c_mex));
00170     d->m = (abip_int) * (mxGetDimensions(b_mex));

```

```

00171
00172     #if SFLOAT > 0
00173         d->b = castTo_abip_float_arr(mxGetPr(b_mex), d->m);
00174         d->c = cast_to_abip_float_arr(mxGetPr(c_mex), d->n);
00175     #else
00176         d->b = (abip_float *)mxGetPr(b_mex);
00177         d->c = (abip_float *)mxGetPr(c_mex);
00178     #endif
00179
00180     // set default parameters
00181     ABIP(set_default_settings)(d);
00182
00183     tmp = mxGetField(settings, 0, "max_ipm_iters");
00184     if (tmp != ABIP_NULL)
00185     {
00186         d->stgs->max_ipm_iters = (abip_int)*mxGetPr(tmp);
00187     }
00188
00189     tmp = mxGetField(settings, 0, "max_admm_iters");
00190     if (tmp != ABIP_NULL)
00191     {
00192         d->stgs->max_admm_iters = (abip_int)*mxGetPr(tmp);
00193     }
00194
00195     tmp = mxGetField(settings, 0, "eps");
00196     if (tmp != ABIP_NULL)
00197     {
00198         d->stgs->eps = (abip_float)*mxGetPr(tmp);
00199     }
00200
00201     tmp = mxGetField(settings, 0, "cg_rate");
00202     if (tmp != ABIP_NULL)
00203     {
00204         d->stgs->cg_rate = (abip_float)*mxGetPr(tmp);
00205     }
00206
00207     tmp = mxGetField(settings, 0, "alpha");
00208     if (tmp != ABIP_NULL)
00209     {
00210         d->stgs->alpha = (abip_float)*mxGetPr(tmp);
00211     }
00212
00213     tmp = mxGetField(settings, 0, "rho_y");
00214     if (tmp != ABIP_NULL)
00215     {
00216         d->stgs->rho_y = (abip_float)*mxGetPr(tmp);
00217     }
00218
00219     tmp = mxGetField(settings, 0, "normalize");
00220     if (tmp != ABIP_NULL)
00221     {
00222         d->stgs->normalize = (abip_int)*mxGetPr(tmp);
00223     }
00224
00225     tmp = mxGetField(settings, 0, "scale");
00226     if (tmp != ABIP_NULL)
00227     {
00228         d->stgs->scale = (abip_float)*mxGetPr(tmp);
00229     }
00230
00231     tmp = mxGetField(settings, 0, "sparsity_ratio");
00232     if (tmp != ABIP_NULL)
00233     {
00234         d->stgs->sparsity_ratio = (abip_float)*mxGetPr(tmp);
00235     }
00236
00237     tmp = mxGetField(settings, 0, "adaptive");
00238     if (tmp != ABIP_NULL)
00239     {
00240         d->stgs->adaptive = (abip_int)*mxGetPr(tmp);
00241     }
00242
00243     tmp = mxGetField(settings, 0, "adaptive_lookback");
00244     if (tmp != ABIP_NULL)
00245     {
00246         d->stgs->adaptive_lookback = (abip_int)*mxGetPr(tmp);
00247     }
00248
00249     tmp = mxGetField(settings, 0, "dynamic_sigma");
00250     if (tmp != ABIP_NULL) {
00251         d->stgs->dynamic_sigma = (abip_float) *mxGetPr(tmp);
00252     }
00253
00254     tmp = mxGetField(settings, 0, "dynamic_x");
00255     if (tmp != ABIP_NULL) {
00256         d->stgs->dynamic_x = (abip_float) *mxGetPr(tmp);
00257     }

```

```

00258
00259     tmp = mxGetField(settings, 0, "dynamic_eta");
00260     if (tmp != ABIP_NULL) {
00261         d->stgs->dynamic_eta = (abip_float) *mxGetPr(tmp);
00262     }
00263
00264     tmp = mxGetField(settings, 0, "restart_thresh");
00265     if (tmp != ABIP_NULL) {
00266         d->stgs->restart_thresh = (abip_int) *mxGetPr(tmp);
00267     }
00268
00269     tmp = mxGetField(settings, 0, "restart_fre");
00270     if (tmp != ABIP_NULL) {
00271         d->stgs->restart_fre = (abip_float) *mxGetPr(tmp);
00272     }
00273
00274     // add by Kurt. 22.05.03
00275     tmp = mxGetField(settings, 0, "origin_rescale");
00276     if (tmp != ABIP_NULL) {
00277         d->stgs->origin_rescale = (abip_int) *mxGetPr(tmp);
00278     }
00279
00280     tmp = mxGetField(settings, 0, "pc_ruiz_rescale");
00281     if (tmp != ABIP_NULL) {
00282         d->stgs->pc_ruiz_rescale = (abip_int) *mxGetPr(tmp);
00283     }
00284
00285     tmp = mxGetField(settings, 0, "qp_rescale");
00286     if (tmp != ABIP_NULL) {
00287         d->stgs->qp_rescale = (abip_int) *mxGetPr(tmp);
00288     }
00289
00290     tmp = mxGetField(settings, 0, "ruiz_iter");
00291     if (tmp != ABIP_NULL) {
00292         d->stgs->ruiz_iter = (abip_int) *mxGetPr(tmp);
00293     }
00294
00295     tmp = mxGetField(settings, 0, "hybrid_mu");
00296     if (tmp != ABIP_NULL) {
00297         d->stgs->hybrid_mu = (abip_int) *mxGetPr(tmp);
00298     }
00299
00300     tmp = mxGetField(settings, 0, "half_update");
00301     if (tmp != ABIP_NULL) {
00302         d->stgs->half_update = (abip_int) *mxGetPr(tmp);
00303     }
00304
00305     tmp = mxGetField(settings, 0, "avg_criterion");
00306     if (tmp != ABIP_NULL) {
00307         d->stgs->avg_criterion = (abip_int) *mxGetPr(tmp);
00308     }
00309
00310     tmp = mxGetField(settings, 0, "hybrid_thresh");
00311     if (tmp != ABIP_NULL) {
00312         d->stgs->hybrid_thresh = (abip_float) *mxGetPr(tmp);
00313     }
00314
00315     tmp = mxGetField(settings, 0, "dynamic_sigma_second");
00316     if (tmp != ABIP_NULL) {
00317         d->stgs->dynamic_sigma_second = (abip_float) *mxGetPr(tmp);
00318     }
00319
00320     tmp = mxGetField(settings, 0, "timelimit");
00321     if (tmp != ABIP_NULL) {
00322         d->stgs->max_time = (abip_float) *mxGetPr(tmp);
00323     }
00324     else {
00325         d->stgs->max_time = 3600;
00326     }
00327     // -----
00328
00329     tmp = mxGetField(settings, 0, "verbose");
00330     if (tmp != ABIP_NULL)
00331     {
00332         d->stgs->verbose = (abip_int) *mxGetPr(tmp);
00333     }
00334
00335     tmp = mxGetField(settings, 0, "feasopt");
00336     if (tmp != ABIP_NULL)
00337     {
00338         d->stgs->pfeasopt = (abip_int) *mxGetPr(tmp);
00339     } else {
00340         d->stgs->pfeasopt = 0;
00341     }
00342
00343     A = (ABIPMatrix *)abip_malloc(sizeof(ABIPMatrix));
00344     A->n = d->n;

```

```

00345     A->m = d->m;
00346
00347     #if DLONG > 0
00348         A->p = (abip_int *)mxGetJc(A_mex);
00349         A->i = (abip_int *)mxGetIr(A_mex);
00350     #else
00351         A->p = cast_to_abip_int_arr(mxGetJc(A_mex), A->n + 1);
00352         A->i = cast_to_abip_int_arr(mxGetIr(A_mex), A->p[A->n]);
00353     #endif
00354
00355     #if SFLOAT > 0
00356         A->x = cast_to_abip_float_arr(mxGetPr(A_mex), A->p[A->n]);
00357     #else
00358         A->x = (abip_float *)mxGetPr(A_mex);
00359     #endif
00360
00361     d->A = A;
00362     d->sp = (abip_float)A->p[A->n]/(A->m*A->n);
00363
00364     d->stgs->warm_start = parse_warm_start((mxArray *) mxGetField(data, 0, "x"), &(sol.x), d->n);
00365     d->stgs->warm_start |= parse_warm_start((mxArray *) mxGetField(data, 0, "y"), &(sol.y), d->m);
00366     d->stgs->warm_start |= parse_warm_start((mxArray *) mxGetField(data, 0, "s"), &(sol.s), d->n);
00367
00368     status = ABIP(main)(d, &sol, &info);
00369
00370     set_output_field(&plhs[0], sol.x, d->n);
00371     set_output_field(&plhs[1], sol.y, d->m);
00372     set_output_field(&plhs[2], sol.s, d->n);
00373
00374     plhs[3] = mxCreateStructArray(1, one, num_info_fields, info_fields);
00375
00376     mxSetField(plhs[3], 0, "status", mxCreateString(info.status));
00377
00378     tmp = mxCreateDoubleMatrix(1, 1, mxREAL);
00379     mxSetField(plhs[3], 0, "ipm_iter", tmp);
00380     *mxGetPr(tmp) = (abip_float)info.ipm_iter;
00381
00382     tmp = mxCreateDoubleMatrix(1, 1, mxREAL);
00383     mxSetField(plhs[3], 0, "admm_iter", tmp);
00384     *mxGetPr(tmp) = (abip_float)info.admm_iter;
00385
00386     tmp = mxCreateDoubleMatrix(1, 1, mxREAL);
00387     mxSetField(plhs[3], 0, "pobj", tmp);
00388     *mxGetPr(tmp) = info.pobj;
00389
00390     tmp = mxCreateDoubleMatrix(1, 1, mxREAL);
00391     mxSetField(plhs[3], 0, "dobj", tmp);
00392     *mxGetPr(tmp) = info.dobj;
00393
00394     tmp = mxCreateDoubleMatrix(1, 1, mxREAL);
00395     mxSetField(plhs[3], 0, "resPri", tmp);
00396     *mxGetPr(tmp) = info.res_pri;
00397
00398     tmp = mxCreateDoubleMatrix(1, 1, mxREAL);
00399     mxSetField(plhs[3], 0, "resDual", tmp);
00400     *mxGetPr(tmp) = info.res_dual;
00401
00402     tmp = mxCreateDoubleMatrix(1, 1, mxREAL);
00403     mxSetField(plhs[3], 0, "relGap", tmp);
00404     *mxGetPr(tmp) = info.rel_gap;
00405
00406     tmp = mxCreateDoubleMatrix(1, 1, mxREAL);
00407     mxSetField(plhs[3], 0, "resInfeas", tmp);
00408     *mxGetPr(tmp) = info.res_infeas;
00409
00410     tmp = mxCreateDoubleMatrix(1, 1, mxREAL);
00411     mxSetField(plhs[3], 0, "resUnbdd", tmp);
00412     *mxGetPr(tmp) = info.res_unbdd;
00413
00414     tmp = mxCreateDoubleMatrix(1, 1, mxREAL);
00415     mxSetField(plhs[3], 0, "setupTime", tmp);
00416     *mxGetPr(tmp) = info.setup_time;
00417
00418     tmp = mxCreateDoubleMatrix(1, 1, mxREAL);
00419     mxSetField(plhs[3], 0, "solveTime", tmp);
00420     *mxGetPr(tmp) = info.solve_time;
00421
00422     free_mex(d);
00423     return;
00424 }
00425
00429 void free_mex(ABIPData *d)
00430 {
00431     if (d)
00432     {
00433         #if SFLOAT > 0
00434         if (d->b)

```

```

00435         {
00436             abip_free(d->b);
00437         }
00438         if (d->c)
00439         {
00440             abip_free(d->c);
00441         }
00442         #endif
00443
00444         if (d->A)
00445         {
00446             #if !(DLONG > 0)
00447                 if (d->A->p)
00448                 {
00449                     abip_free(d->A->p);
00450                 }
00451                 if (d->A->i)
00452                 {
00453                     abip_free(d->A->i);
00454                 }
00455                 #endif
00456
00457                 #if SFLOAT > 0
00458                 if (d->A->x)
00459                 {
00460                     abip_free(d->A->x);
00461                 }
00462                 #endif
00463
00464                 abip_free(d->A);
00465             }
00466
00467             if (d->stgs)
00468             {
00469                 abip_free(d->stgs);
00470             }
00471
00472             abip_free(d);
00473         }
00474     }

```

5.90 mexfile/abip_version_mex.c File Reference

```

#include "mex.h"
#include "abip.h"

```

Functions

- void [mexFunction](#) (int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[])

5.90.1 Function Documentation

5.90.1.1 mexFunction()

```

void mexFunction (
    int nlhs,
    mxArray * plhs[],
    int nrhs,
    const mxArray * prhs[] )

```

Definition at line 4 of file [abip_version_mex.c](#).

5.91 abip_version_mex.c

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

```
00001 #include "mex.h"
00002 #include "abip.h"
00003
00004 void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[])
00005 {
00006     if (nrhs != 0) {
00007         mexErrMsgTxt("Too many input arguments.");
00008     }
00009     if (nlhs > 1) {
00010         mexErrMsgTxt("Too many output arguments.");
00011     }
00012     plhs[0] = mxCreateString(abip_version());
00013     return;
00014 }
```

5.92 src/abip.c File Reference

```
#include <assert.h>
#include <time.h>
#include "abip.h"
#include "glbopts.h"
#include "adaptive.h"
#include "ctrlc.h"
#include "linalg.h"
#include "linsys.h"
#include "normalize.h"
#include "util.h"
```

Functions

- [ABIP](#) (timer)
- [abip_int ABIP\(\)](#) [solve](#) ([ABIPWork](#) *w, const [ABIPData](#) *d, [ABIPSolution](#) *sol, [ABIPInfo](#) *info)
detailed update rule of ABIP
- void [ABIP\(\)](#) [finish](#) ([ABIPWork](#) *w)
- [ABIPWork](#) *[ABIP\(\)](#) [init](#) (const [ABIPData](#) *d, [ABIPInfo](#) *info)
- [abip_int ABIP\(\)](#) [main](#) (const [ABIPData](#) *d, [ABIPSolution](#) *sol, [ABIPInfo](#) *info)
the main function

5.92.1 Function Documentation

5.92.1.1 ABIP()

```
ABIP (
    timer )
```

Definition at line 12 of file [abip.c](#).

5.92.1.2 finish()

```
void ABIP() finish (
    ABIPWork * w )
```

Definition at line 2301 of file [abip.c](#).

5.92.1.3 init()

```
ABIPWork *ABIP() init (
    const ABIPData * d,
    ABIPInfo * info )
```

Definition at line 2341 of file [abip.c](#).

5.92.1.4 main()

```
abip_int ABIP() main (
    const ABIPData * d,
    ABIPSolution * sol,
    ABIPInfo * info )
```

the main function

Definition at line 2393 of file [abip.c](#).

5.92.1.5 solve()

```
abip_int ABIP() solve (
    ABIPWork * w,
    const ABIPData * d,
    ABIPSolution * sol,
    ABIPInfo * info )
```

detailed update rule of ABIP

Definition at line 2056 of file [abip.c](#).

5.93 abip.c

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

```

00001 #include <assert.h>
00002 #include <time.h>
00003 #include "abip.h"
00004 #include "glbopts.h"
00005 #include "adaptive.h"
00006 #include "ctrlc.h"
00007 #include "linalg.h"
00008 #include "linsys.h"
00009 #include "normalize.h"
00010 #include "util.h"
00011
00012 ABIP(timer) global_timer;
00013
00014 /*
00015 @brief printing header
00016 */
00017 static const char *HEADER[] = {
00018     " ipm iter ", " admm iter ", "      mu ",
00019     " pri res ", " dua res ", " rel gap ",
00020     " pri obj ", " dua obj ", " kap/tau ", " time (s)",
00021 };
00022
00023 static const abip_int HSPACE = 9;
00024 static const abip_int HEADER_LEN = 10;
00025 static const abip_int LINE_LEN = 150;
00026
00030 static abip_int abip_isnan
00031 (
00032     abip_float x
00033 )
00034 {
00035     DEBUG_FUNC
00036     RETURN(x == NAN || x != x);
00037 }
00041 static void free_work
00042 (
00043     ABIPWork *w
00044 )
00045 {
00046     DEBUG_FUNC
00047
00048     if (!w) {
00049         RETURN;
00050     }
00051
00052     if (w->u)
00053     {
00054         abip_free(w->u);
00055     }
00056
00057     if (w->v)
00058     {
00059         abip_free(w->v);
00060     }
00061
00062     if (w->u_t)
00063     {
00064         abip_free(w->u_t);
00065     }
00066
00067     if (w->u_avg)
00068     {
00069         abip_free(w->u_avg);
00070     }
00071
00072     if (w->v_avg)
00073     {
00074         abip_free(w->v_avg);
00075     }
00076
00077     if (w->u_avgcon)
00078     {
00079         abip_free(w->u_avgcon);
00080     }
00081
00082     if (w->v_avgcon)
00083     {
00084         abip_free(w->v_avgcon);
00085     }
00086
00087     if (w->u_sumcon)
00088     {

```



```

00089     abip_free(w->u_sumcon);
00090 }
00091
00092 if (w->v_sumcon)
00093 {
00094     abip_free(w->v_sumcon);
00095 }
00096
00097 if (w->u_prev)
00098 {
00099     abip_free(w->u_prev);
00100 }
00101
00102 if (w->v_prev)
00103 {
00104     abip_free(w->v_prev);
00105 }
00106
00107 if (w->h)
00108 {
00109     abip_free(w->h);
00110 }
00111
00112 if (w->g)
00113 {
00114     abip_free(w->g);
00115 }
00116
00117 if (w->pr)
00118 {
00119     abip_free(w->pr);
00120 }
00121
00122 if (w->dr)
00123 {
00124     abip_free(w->dr);
00125 }
00126
00127 if (w->b)
00128 {
00129     abip_free(w->b);
00130 }
00131
00132 if (w->c)
00133 {
00134     abip_free(w->c);
00135 }
00136
00137 if (w->scal)
00138 {
00139     if (w->scal->D)
00140     {
00141         abip_free(w->scal->D);
00142     }
00143
00144     if (w->scal->E)
00145     {
00146         abip_free(w->scal->E);
00147     }
00148
00149     abip_free(w->scal);
00150 }
00151
00152 abip_free(w);
00153
00154 RETURN;
00155 }
00159 static void print_init_header
00160 {
00161     const ABIPData *d
00162 }
00163 {
00164     DEBUG_FUNC
00165
00166     abip_int i;
00167     ABIPSettings *stgs = d->stgs;
00168     char *lin_sys_method = ABIP(get_lin_sys_method)(d->A, d->stgs);
00169
00170     for (i = 0; i < LINE_LEN; ++i)
00171     {
00172         abip_printf("-");
00173     }
00174
00175     abip_printf("\n\tABIP v%s - First-Order Interior-Point Solver\n\t(c) Tianyi Lin, UC Berkeley,
2018-2020\n\t(c) LEAVES Group, 2021-2024\n",
ABIP(version)());
00176
00177

```

```

00178     for (i = 0; i < LINE_LEN; ++i)
00179     {
00180         abip_printf("-");
00181     }
00182
00183     abip_printf("\n");
00184
00185     if (lin_sys_method)
00186     {
00187         abip_printf("Lin-sys: %s\n", lin_sys_method);
00188         abip_free(lin_sys_method);
00189     }
00190
00191     if (stgs->normalize)
00192     {
00193         abip_printf("eps = %.2e, alpha = %.2f, max_ipm_iters = %i, max_admm_iters = %i, normalize =
00194         %i\n"
00195         "scale = %.2f, adaptive = %i, adaptive_lookback = %i, rho_y = %.2e\n",
00196         stgs->eps, stgs->alpha, (int)stgs->max_ipm_iters, (int)stgs->max_admm_iters,
00197         (int)stgs->normalize, stgs->scale, (int)stgs->adaptive, stgs->adaptive_lookback,
00198         stgs->rho_y);
00199     }
00200     else
00201     {
00202         abip_printf("eps = %.2e, alpha = %.2f, max_ipm_iters = %i, max_admm_iters = %i, normalize =
00203         %i\n"
00204         "adaptive = %i, adaptive_lookback = %i, rho_y = %.2e\n",
00205         stgs->eps, stgs->alpha, (int)stgs->max_ipm_iters, (int)stgs->max_admm_iters,
00206         (int)stgs->normalize, (int)stgs->adaptive, stgs->adaptive_lookback, stgs->rho_y);
00207     }
00208     abip_printf("Variables n = %i, constraints m = %i\n", (int)d->n, (int)d->m);
00209
00210 #ifdef MATLAB_MEX_FILE
00211     mexEvalString("drawnow;");
00212 #endif
00213
00214     RETURN;
00215 }
00216 static void populate_on_failure
00217 (
00218     abip_int m,
00219     abip_int n,
00220     ABIPSolution *sol,
00221     ABIPInfo *info,
00222     abip_int status_val,
00223     const char *msg
00224 )
00225 {
00226     DEBUG_FUNC
00227
00228     if (info)
00229     {
00230         info->res_pri = NAN;
00231         info->res_dual = NAN;
00232         info->rel_gap = NAN;
00233         info->res_infeas = NAN;
00234         info->res_unbdd = NAN;
00235
00236         info->pobj = NAN;
00237         info->dobj = NAN;
00238
00239         info->ipm_iter = -1;
00240         info->admm_iter = -1;
00241         info->status_val = status_val;
00242         info->solve_time = NAN;
00243         strcpy(info->status, msg);
00244     }
00245
00246     if (sol)
00247     {
00248         if (n > 0)
00249         {
00250             if (!sol->x)
00251             {
00252                 sol->x = (abip_float *)abip_malloc(sizeof(abip_float) * n);
00253             }
00254             ABIP(scale_array)(sol->x, NAN, n);
00255
00256             if (!sol->s)
00257             {
00258                 sol->s = (abip_float *)abip_malloc(sizeof(abip_float) * n);
00259             }
00260             ABIP(scale_array)(sol->s, NAN, m);
00261         }
00262     }
00263 }

```

```

00265
00266     if (m > 0)
00267     {
00268         if (!sol->y)
00269         {
00270             sol->y = (abip_float *)abip_malloc(sizeof(abip_float) * m);
00271         }
00272         ABIP(scale_array)(sol->y, NAN, m);
00273     }
00274 }
00275
00276 RETURN;
00277 }
00278
00282 static abip_int failure
00283 (
00284     ABIPWork *w,
00285     abip_int m,
00286     abip_int n,
00287     ABIPSolution *sol,
00288     ABIPInfo *info,
00289     abip_int stint,
00290     const char *msg,
00291     const char *ststr
00292 )
00293 {
00294     DEBUG_FUNC
00295
00296     abip_int status = stint;
00297     populate_on_failure(m, n, sol, info, status, ststr);
00298
00299     abip_printf("Failure:%s\n", msg);
00300     abip_end_interrupt_listener();
00301
00302     RETURN status;
00303 }
00307 static void warm_start_vars
00308 (
00309     ABIPWork *w,
00310     const ABIPSolution *sol
00311 )
00312 {
00313     DEBUG_FUNC
00314
00315     abip_int i;
00316     abip_int n = w->n;
00317     abip_int m = w->m;
00318
00319     memset(w->v, 0, m * sizeof(abip_float));
00320     memcpy(w->u, sol->y, m * sizeof(abip_float));
00321     memcpy(&(w->u[m]), sol->x, n * sizeof(abip_float));
00322     memcpy(&(w->v[m]), sol->s, n * sizeof(abip_float));
00323     w->u[n + m] = 1.0;
00324     w->v[n + m] = 0.0;
00325
00326 #ifndef NOVALIDATE
00327
00328     for (i = 0; i < n + m + 1; ++i)
00329     {
00330         if (abip_isnan(w->u[i]) && i < m)
00331         {
00332             w->u[i] = 0;
00333         }
00334         else
00335         {
00336             w->u[i] = SQRTF(w->mu/w->beta);
00337         }
00338
00339         if (abip_isnan(w->v[i]))
00340         {
00341             w->v[i] = 0;
00342         }
00343         else
00344         {
00345             w->v[i] = SQRTF(w->mu/w->beta);
00346         }
00347     }
00348 #endif
00349
00350 if (w->stgs->normalize)
00351 {
00352     ABIP(normalize_warm_start)(w);
00353 }
00354
00355 RETURN;
00356 }
00357 }

```

```

00361 static void cold_start_vars
00362 (
00363     ABIPWork *w
00364 )
00365 {
00366     DEBUG_FUNC
00367     abip_int l = w->m + w->n + 1;
00368     abip_int i;
00369
00370     memset(w->u, 0, w->m * sizeof(abip_float));
00371     memset(w->v, 0, w->m * sizeof(abip_float));
00372
00373     for (i = w->m; i < l; ++i)
00374     {
00375         w->u[i] = SQRTF(w->mu/w->beta);
00376         w->v[i] = SQRTF(w->mu/w->beta);
00377     }
00378
00379     RETURN;
00380 }
00381
00382 static abip_float calc_primal_resid
00383 (
00384     ABIPWork *w,
00385     const abip_float *x,
00386     const abip_float tau,
00387     abip_float *nm_A_x
00388 )
00389 {
00390     DEBUG_FUNC
00391     abip_int i;
00392
00393     abip_float pres = 0;
00394     abip_float scale;
00395     abip_float *pr = w->pr;
00396
00397     *nm_A_x = 0;
00398
00399     // compute w->pr again
00400     memset(pr, 0, w->m * sizeof(abip_float));
00401     ABIP(accum_by_A)(w->A, w->p, x, pr);
00402
00403     for (i = 0; i < w->m; ++i)
00404     {
00405         scale = w->stgs->normalize ? w->scal->D[i] / (w->sc_b * w->stgs->scale) : 1;
00406         scale = scale * scale;
00407         *nm_A_x += (pr[i] * pr[i]) * scale;
00408         pres += (pr[i] - w->b[i] * tau) * (pr[i] - w->b[i] * tau) * scale;
00409     }
00410
00411     *nm_A_x = SQRTF(*nm_A_x);
00412     RETURN SQRTF(pres);
00413 }
00414
00415 static abip_float calc_dual_resid
00416 (
00417     ABIPWork *w,
00418     const abip_float *y,
00419     const abip_float *s,
00420     const abip_float tau,
00421     abip_float *nm_At_ys
00422 )
00423 {
00424     DEBUG_FUNC
00425     abip_int i;
00426     abip_float dres = 0;
00427     abip_float scale;
00428     abip_float *dr = w->dr;
00429
00430     *nm_At_ys = 0;
00431
00432     memset(dr, 0, w->n * sizeof(abip_float));
00433     ABIP(accum_by_Atrans)(w->A, w->p, y, dr);
00434     ABIP(add_scaled_array)(dr, s, w->n, 1.0);
00435
00436     for (i = 0; i < w->n; ++i)
00437     {
00438         scale = w->stgs->normalize ? w->scal->E[i] / (w->sc_c * w->stgs->scale) : 1;
00439         scale = scale * scale;
00440         *nm_At_ys += (dr[i] * dr[i]) * scale;
00441         dres += (dr[i] - w->c[i] * tau) * (dr[i] - w->c[i] * tau) * scale;
00442     }
00443
00444     *nm_At_ys = SQRTF(*nm_At_ys);
00445     RETURN SQRTF(dres);
00446 }
00447
00448 }

```

```

00454
00458 static void calc_residuals
00459 (
00460     ABIPWork *w,
00461     ABIPResiduals *r,
00462     abip_int ipm_iter,
00463     abip_int admm_iter
00464 )
00465 {
00466     DEBUG_FUNC
00467
00468     abip_float *y;
00469     abip_float *x;
00470     abip_float *s;
00471
00472     // find y, x, s
00473     if (w->stgs->avg_criterion)
00474     {
00475         y = w->u_avgcon;
00476         x = &(w->u_avgcon[w->m]);
00477         s = &(w->v_avgcon[w->m]);
00478     }
00479     else
00480     {
00481         y = w->u;
00482         x = &(w->u[w->m]);
00483         s = &(w->v[w->m]);
00484     }
00485
00486     abip_float nmpr_tau;
00487     abip_float nmldr_tau;
00488     abip_float nm_A_x_tau;
00489     abip_float nm_At_ys_tau;
00490     abip_float ct_x;
00491     abip_float bt_y;
00492
00493     abip_int n = w->n;
00494     abip_int m = w->m;
00495
00496     if (admm_iter && r->last_admm_iter == admm_iter)
00497     {
00498         RETURN;
00499     }
00500
00501     r->last_ipm_iter = ipm_iter;
00502     r->last_admm_iter = admm_iter;
00503
00504
00505     if (w->stgs->avg_criterion)
00506     {
00507         r->tau = ABS(w->u_avgcon[n + m]);
00508         r->kap = ABS(w->v_avgcon[n + m]) / (w->stgs->normalize ? (w->stgs->scale * w->sc_c * w->sc_b)
: 1);
00509     }
00510     else
00511     {
00512         r->tau = ABS(w->u[n + m]);
00513         r->kap = ABS(w->v[n + m]) / (w->stgs->normalize ? (w->stgs->scale * w->sc_c * w->sc_b) : 1);
00514     }
00515
00516
00517     // compute primal resid. and dual resid. memset pr and dr, then compute.
00518     nmpr_tau = calc_primal_resid(w, x, r->tau, &nm_A_x_tau);
00519     nmldr_tau = calc_dual_resid(w, y, s, r->tau, &nm_At_ys_tau);
00520
00521     r->bt_y_by_tau = ABIP(dot)(y, w->b, m) / (w->stgs->normalize ? (w->stgs->scale * w->sc_c *
w->sc_b) : 1);
00522     r->ct_x_by_tau = ABIP(dot)(x, w->c, n) / (w->stgs->normalize ? (w->stgs->scale * w->sc_c *
w->sc_b) : 1);
00523
00524     r->res_infeas = r->bt_y_by_tau > 0 ? w->nm_b * nm_At_ys_tau / r->bt_y_by_tau : NAN;
00525     r->res_unbdd = r->ct_x_by_tau < 0 ? w->nm_c * nm_A_x_tau / -r->ct_x_by_tau : NAN;
00526
00527     bt_y = SAFEDIV_POS(r->bt_y_by_tau, r->tau);
00528     ct_x = SAFEDIV_POS(r->ct_x_by_tau, r->tau);
00529
00530     r->res_pri = SAFEDIV_POS(nmpr_tau / (1 + w->nm_b), r->tau);
00531     r->res_dual = SAFEDIV_POS(nmldr_tau / (1 + w->nm_c), r->tau);
00532     r->rel_gap = ABS(ct_x - bt_y) / (1 + ABS(ct_x) + ABS(bt_y));
00533
00534     RETURN;
00535 }
00539 static abip_int project_lin_sys
00540 (
00541     ABIPWork *w,
00542     abip_int iter
00543 )

```

```

00544 {
00545     DEBUG_FUNC
00546
00547     abip_int n = w->n;
00548     abip_int m = w->m;
00549     abip_int l = n + m + 1;
00550
00551     abip_int status;
00552     memcpy(w->u_t, w->u, l * sizeof(abip_float));
00553     ABIP(add_scaled_array)(w->u_t, w->v, l, 1.0);
00554
00555     ABIP(scale_array)(w->u_t, w->stgs->rho_y, m);
00556     ABIP(add_scaled_array)(w->u_t, w->h, l - 1, -w->u_t[l - 1]);
00557     ABIP(add_scaled_array)(w->u_t, w->h, l - 1, -ABIP(dot)(w->u_t, w->g, l - 1) / (w->g_th + 1));
00558     ABIP(scale_array)(w->u_t[m], -1, n);
00559     status = ABIP(solve_lin_sys)(w->A, w->stgs, w->p, w->u_t, w->u, iter);
00560     w->u_t[l - 1] += ABIP(dot)(w->u_t, w->h, l - 1);
00561     RETURN status;
00562 }
00563
00564 static void update_dual_vars
00565 (
00566     ABIPWork *w
00567 )
00568 {
00569     DEBUG_FUNC
00570
00571     abip_int i;
00572     abip_int m = w->m;
00573     abip_int l = m + w->n + 1;
00574
00575     for (i = m; i < l; ++i)
00576     {
00577         w->v[i] += (w->u[i] - w->stgs->alpha * w->u_t[i] - (1.0 - w->stgs->alpha) * w->u_prev[i]);
00578     }
00579
00580     RETURN;
00581 }
00582
00583 /* @brief use restart strategy
00584 */
00585 static void restart_vars
00586 (
00587     ABIPWork* w,
00588     abip_int admm_iter,
00589     abip_int total_admm_iter
00590 )
00591 {
00592     DEBUG_FUNC
00593
00594     abip_int i;
00595     abip_int fre = w->stgs->restart_fre;
00596     abip_int m = w->m;
00597     abip_int l = m + w->n + 1;
00598
00599     for (i = 0; i < l; ++i)
00600     {
00601         w->u_avg[i] += w->u[i];
00602         w->v_avg[i] += w->v[i];
00603     }
00604
00605     if (total_admm_iter < w->stgs->restart_thresh ||
00606         (admm_iter + 1 - w->fre_old) % fre != 0)
00607     {
00608         return;
00609     }
00610
00611     for (i = 0; i < l; ++i)
00612     {
00613         w->u_avg[i] /= fre;
00614         w->v_avg[i] /= fre;
00615     }
00616
00617     memcpy(w->u, w->u_avg, sizeof(abip_float) * l);
00618     memcpy(w->v, w->v_avg, sizeof(abip_float) * l);
00619     memset(w->u_avg, 0, sizeof(abip_float) * l);
00620     memset(w->v_avg, 0, sizeof(abip_float) * l);
00621
00622     // memset(w->u_sumcon, 0, sizeof(abip_float) * l);
00623     // memset(w->v_sumcon, 0, sizeof(abip_float) * l);
00624
00625     w->fre_old = fre;
00626
00627 }
00628
00629 static void compute_avg
00630 (

```

```

00637     ABIPWork* w,
00638     abip_int admm_iter
00639 )
00640 {
00641     DEBUG_FUNC
00642
00643     abip_int i;
00644     abip_int m = w->m;
00645     abip_int l = m + w->n + 1;
00646
00647     abip_int dom = admm_iter + 1;
00648
00649     for (i = 0; i < l; ++i)
00650     {
00651         w->u_sumcon[i] += w->u[i] ;
00652         w->v_sumcon[i] += w->v[i] ;
00653
00654         w->u_avgcon[i] = w->u_sumcon[i]/dom;
00655         w->v_avgcon[i] = w->v_sumcon[i]/dom;
00656     }
00657
00658
00659 }
00660
00661
00662 // add by Kurt. 22.04.11
00663 static void half_update_dual_vars
00664 (
00665     ABIPWork *w
00666 )
00667 {
00668     DEBUG_FUNC
00669
00670     abip_int i;
00671     abip_int l = w->m + w->n + 1;
00672
00673     for (i = 0; i < l; ++i)
00674     {
00675         w->v[i] += 0.5 * (w->u[i] - w->u_t[i] );
00676     }
00677
00678     RETURN;
00679 }
00680
00681 static void project_barrier_dual
00682 (
00683     ABIPWork *w
00684 )
00685 {
00686     DEBUG_FUNC
00687
00688     abip_int i;
00689     abip_int m = w->m;
00690     abip_int l = m + w->n + 1;
00691     abip_float tmp;
00692     // update u
00693     for (i = 0; i < l; ++i)
00694     {
00695         w->u[i] = w->u_t[i] - w->v[i];
00696     }
00697
00698     for(i = m; i < l; ++i)
00699     {
00700         tmp = w->u[i] / 2;
00701         w->u[i] = tmp + SQRTF(tmp * tmp + w->mu / w->beta);
00702     }
00703
00704     // dual update
00705     for (i = 0; i < l; ++i)
00706     {
00707         w->v[i] += (w->u[i] - w->u_t[i]);
00708     }
00709
00710     RETURN;
00711 }
00712
00713
00717 static void project_barrier
00718 (
00719     ABIPWork *w
00720 )
00721 {
00722     DEBUG_FUNC
00723
00724     abip_int i;
00725     abip_int m = w->m;
00726     abip_int l = m + w->n + 1;

```

```

00727     abip_int status;
00728
00729     abip_float tmp;
00730
00731     for (i = 0; i < m; ++i)
00732     {
00733         w->u[i] = w->u_t[i] - w->v[i];
00734     }
00735
00736     for (i = m; i < l; ++i)
00737     {
00738         w->u[i] = w->stgs->alpha * w->u_t[i] + (1 - w->stgs->alpha) * w->u_prev[i] - w->v[i];
00739     }
00740
00741     for(i = m; i < l; ++i)
00742     {
00743         tmp = w->u[i] / 2;
00744         w->u[i] = tmp + SQRTF(tmp * tmp + w->mu / w->beta);
00745     }
00746
00747     RETURN;
00748 }
00749
00753 static void update_barrier
00754 (
00755     ABIPWork* w,
00756     ABIPResiduals* r
00757 )
00758 {
00759     abip_float sigma, gamma, mu = w->mu;
00760
00761     abip_float ratio = w->mu / w->stgs->eps;
00762     abip_float err_ratio = MAX(MAX(r->res_pri, r->res_dual), r->rel_gap) / w->stgs->eps;
00763
00764     if (MAX(w->sp, w->stgs->sparsity_ratio) > 0.4 || MIN(w->sp, w->stgs->sparsity_ratio) > 0.1)
00765     {
00766         if (ratio > 10.0)
00767         {
00768             gamma = 2.0;
00769         }
00770         else if (ratio > 1.0 && ratio <= 10.0)
00771         {
00772             gamma = 1.0;
00773         }
00774         else if (ratio > 0.5 && ratio <= 1.0)
00775         {
00776             gamma = 0.9;
00777         }
00778         else if (ratio > 0.1 && ratio <= 0.5)
00779         {
00780             gamma = 0.8;
00781         }
00782         else if (ratio > 0.05 && ratio <= 0.1)
00783         {
00784             gamma = 0.7;
00785         }
00786         else if (ratio > 0.01 && ratio <= 0.05)
00787         {
00788             gamma = 0.6;
00789         }
00790         else if (ratio > 0.005 && ratio <= 0.01)
00791         {
00792             gamma = 0.5;
00793         }
00794         else if (ratio > 0.001 && ratio <= 0.005)
00795         {
00796             gamma = 0.4;
00797         }
00798         else
00799         {
00800             gamma = 0.3;
00801         }
00802
00803         if (err_ratio > 6 && err_ratio <= 10)
00804         {
00805             sigma = 0.5;
00806         }
00807         else if (err_ratio > 3 && err_ratio <= 6)
00808         {
00809             sigma = 0.6;
00810             gamma = gamma * 0.8;
00811         }
00812         else if (err_ratio > 1 && err_ratio <= 3)
00813         {
00814             w->final_check = 1;
00815             gamma = gamma * 0.4;
00816             if (ratio < 0.1)

```



```
00817         {
00818             sigma = 0.8;
00819         }
00820         else
00821         {
00822             sigma = 0.7;
00823         }
00824     }
00825 }
00826 else
00827 {
00828     sigma = w->sigma;
00829 }
00830 }
00831 else
00832 {
00833     if (ratio > 10.0)
00834     {
00835         gamma = 3.0;
00836     }
00837     else if (ratio > 1.0 && ratio <= 10.0)
00838     {
00839         gamma = 1.0;
00840     }
00841     else if (ratio > 0.5 && ratio <= 1.0)
00842     {
00843         gamma = 0.9;
00844     }
00845     else if (ratio > 0.1 && ratio <= 0.5)
00846     {
00847         gamma = 0.8;
00848     }
00849     else if (ratio > 0.05 && ratio <= 0.1)
00850     {
00851         gamma = 0.7;
00852     }
00853     else if (ratio > 0.01 && ratio <= 0.05)
00854     {
00855         gamma = 0.6;
00856     }
00857     else if (ratio > 0.005 && ratio <= 0.01)
00858     {
00859         gamma = 0.5;
00860     }
00861     else if (ratio > 0.001 && ratio <= 0.005)
00862     {
00863         gamma = 0.4;
00864     }
00865     else
00866     {
00867         gamma = 0.3;
00868     }
00869 }
00870 if (err_ratio > 6 && err_ratio <= 10)
00871 {
00872     sigma = 0.82;
00873     gamma = gamma * 0.8;
00874 }
00875 else if (err_ratio > 4 && err_ratio <= 6)
00876 {
00877     sigma = 0.84;
00878     gamma = gamma * 0.6;
00879 }
00880 else if (err_ratio > 3 && err_ratio <= 4)
00881 {
00882     sigma = 0.85;
00883     gamma = gamma * 0.5;
00884     w->final_check = 1;
00885 }
00886 else if (err_ratio > 1 && err_ratio <= 3)
00887 {
00888     w->final_check = 1;
00889     if (ratio < 0.1)
00890     {
00891         if (w->double_check)
00892         {
00893             sigma = 0.9;
00894             gamma = gamma * 0.4;
00895             w->double_check = 0;
00896         }
00897         else
00898         {
00899             sigma = 1.0;
00900             gamma = gamma * 0.1;
00901             w->double_check = 1;
00902         }
00903     }
00904 }
```

```

00904         else
00905         {
00906             sigma = 0.88;
00907             gamma = gamma * 0.4;
00908         }
00909     }
00910     else
00911     {
00912         sigma = w->sigma;
00913     }
00914 }
00915
00916 mu = mu * sigma;
00917
00918 w->mu = mu;
00919 w->sigma = sigma;
00920 w->gamma = gamma;
00921 }
00922
00923 // #ifndef MYDEBUG
00924 // #define MYDEBUG
00925 // #endif
00926
00930 static void update_barrier_dynamic
00931 (
00932     ABIPWork *w,
00933     ABIPResiduals *r
00934 )
00935 {
00936     /*
00937      * Implementation of a more delicate mu adjustment strategy
00938      */
00939     ksi = min_i {x_i * z_i} / (x' * z / n)
00940     sigma = 0.1 * min(0.05 * (1 - ksi) / ksi, 2)^3
00941
00942     */
00943
00944     abip_int m = w->m, n = w->n, l = m + n + 1, i;
00945
00946     double *u;
00947     double *v;
00948     double shrink = w->stgs->dynamic_sigma;
00949
00950     if (w->stgs->avg_criterion) {
00951         u = w->u_avgcon;
00952         v = w->v_avgcon;
00953     }
00954     else
00955     {
00956         u = w->u;
00957         v = w->v;
00958     }
00959
00960     double ksi = 0.0, sigma = 0.0, xisi = 0.0, xs = 0.0, minxs = 1e+10;
00961
00962     for (i = m; i < l; ++i) {
00963         xisi = u[i] * v[i];
00964         xs += xisi; minxs = MIN(xisi, minxs);
00965     }
00966
00967     if (minxs <= 0.0) {
00968         abip_printf("Invalid xisi < 0 \n");
00969         assert(0);
00970     }
00971
00972     xs /= (n + 1); ksi = minxs / xs;
00973     sigma = MIN(0.05 * (1 - ksi) / ksi, 2.0);
00974     sigma = MAX(0.1 * sigma * sigma * sigma, shrink);
00975     w->mu *= sigma;
00976     // w->gamma = gamma;
00977 }
00978
00982 static void update_barrier_dynamic_2
00983 (
00984     ABIPWork* w
00985 ) {
00986
00987     abip_float x = w->stgs->dynamic_x;
00988     abip_float eta = w->stgs->dynamic_sigma;
00989     w->mu *= MIN(x * w->mu, pow(w->mu, eta));
00990     return;
00991 }
00992
00996 static void reinitialize_vars
00997 (
00998     ABIPWork *w,
00999     abip_int indx

```

```

01000     )
01001 {
01002     DEBUG_FUNC
01003
01004     abip_int i;
01005     abip_int m = w->m;
01006     abip_int l = m + w->n + 1;
01007
01008     if (w->stgs->avg_criterion) // restart related
01009     {
01010         if (indx == 0)
01011         {
01012             for (i = m; i < l; ++i)
01013             {
01014                 if (w->u_avgcon[i] > w->v_avgcon[i])
01015                 {
01016                     w->v_avgcon[i] = w->sigma * w->v_avgcon[i];
01017                 }
01018                 else
01019                 {
01020                     w->u_avgcon[i] = w->sigma * w->u_avgcon[i];
01021                 }
01022             }
01023         }
01024         else if (indx == 1)
01025         {
01026             for (i = m; i < l; ++i)
01027             {
01028                 w->u_avgcon[i] = SQRTF(w->sigma) * w->u_avgcon[i];
01029                 w->v_avgcon[i] = SQRTF(w->sigma) * w->v_avgcon[i];
01030             }
01031         }
01032         else
01033         {
01034             for (i = m; i < l; ++i)
01035             {
01036                 w->u_avgcon[i] = SQRTF(1.0/w->sigma) * w->u_avgcon[i];
01037                 w->v_avgcon[i] = SQRTF(1.0/w->sigma) * w->v_avgcon[i];
01038             }
01039         }
01040     }
01041     else
01042     {
01043         if (indx == 0)
01044         {
01045             for (i = m; i < l; ++i)
01046             {
01047                 if (w->u[i] > w->v[i])
01048                 {
01049                     w->v[i] = w->sigma * w->v[i];
01050                 }
01051                 else
01052                 {
01053                     w->u[i] = w->sigma * w->u[i];
01054                 }
01055             }
01056         }
01057         else if (indx == 1)
01058         {
01059             for (i = m; i < l; ++i)
01060             {
01061                 w->u[i] = SQRTF(w->sigma) * w->u[i];
01062                 w->v[i] = SQRTF(w->sigma) * w->v[i];
01063             }
01064         }
01065         else
01066         {
01067             for (i = m; i < l; ++i)
01068             {
01069                 w->u[i] = SQRTF(1.0/w->sigma) * w->u[i];
01070                 w->v[i] = SQRTF(1.0/w->sigma) * w->v[i];
01071             }
01072         }
01073     }
01074     RETURN;
01075 }
01076 static abip_int indeterminate
01077 {
01078     ABIPWork *w,
01079     ABIPSolution *sol,
01080     ABIPInfo *info
01081 }
01082 {
01083     DEBUG_FUNC
01084
01085     strcpy(info->status, "Indeterminate");
01086 }

```

```

01090     ABIP(scale_array)(sol->x, NAN, w->n);
01091     ABIP(scale_array)(sol->y, NAN, w->m);
01092     ABIP(scale_array)(sol->s, NAN, w->n);
01093
01094     RETURN ABIP_INDETERMINATE;
01095 }
01096
01097 static abip_int solved
01098 (
01099     ABIPWork *w,
01100     ABIPSolution *sol,
01101     ABIPInfo *info,
01102     abip_float tau
01103 )
01104 {
01105     DEBUG_FUNC
01106
01107     ABIP(scale_array)(sol->x, SAFEDIV_POS(1.0, tau), w->n);
01108     ABIP(scale_array)(sol->y, SAFEDIV_POS(1.0, tau), w->m);
01109     ABIP(scale_array)(sol->s, SAFEDIV_POS(1.0, tau), w->n);
01110
01111     if (info->status_val == 0)
01112     {
01113         strcpy(info->status, "Solved/Inaccurate");
01114         RETURN ABIP_SOLVED_INACCURATE;
01115     }
01116
01117     strcpy(info->status, "Solved");
01118
01119     RETURN ABIP_SOLVED;
01120 }
01121
01122 static void sety
01123 (
01124     ABIPWork *w,
01125     ABIPSolution *sol
01126 )
01127 {
01128     DEBUG_FUNC
01129
01130     if (!sol->y)
01131     {
01132         sol->y = (abip_float *) abip_malloc(sizeof(abip_float) * w->m);
01133     }
01134
01135     if (w->stgs->avg_criterion)
01136     {
01137         memcpy(sol->y, w->u_avgcon, w->m * sizeof(abip_float));
01138     }
01139     else
01140     {
01141         memcpy(sol->y, w->u, w->m * sizeof(abip_float));
01142     }
01143
01144     RETURN;
01145 }
01146
01147 static void setx
01148 (
01149     ABIPWork *w,
01150     ABIPSolution *sol
01151 )
01152 {
01153     DEBUG_FUNC
01154
01155     if (!sol->x)
01156     {
01157         sol->x = (abip_float *) abip_malloc(sizeof(abip_float) * w->n);
01158     }
01159
01160     if (w->stgs->avg_criterion)
01161     {
01162         memcpy(sol->x, &(w->u_avgcon[w->m]), w->n * sizeof(abip_float));
01163     }
01164     else
01165     {
01166         memcpy(sol->x, &(w->u[w->m]), w->n * sizeof(abip_float));
01167     }
01168
01169     RETURN;
01170 }
01171
01172 static void sets
01173 (
01174     ABIPWork *w,
01175     ABIPSolution *sol

```

```

01189     )
01190 {
01191     DEBUG_FUNC
01192
01193     if (!sol->s)
01194     {
01195         sol->s = (abip_float *) abip_malloc(sizeof(abip_float) * w->n);
01196     }
01197
01198     if (w->stgs->avg_criterion)
01199     {
01200         memcpy(sol->s, &(w->v_avgcon[w->m]), w->n * sizeof(abip_float));
01201     }
01202     else
01203     {
01204         memcpy(sol->s, &(w->v[w->m]), w->n * sizeof(abip_float));
01205     }
01206 }
01207
01208     RETURN;
01209 }
01210
01211 static abip_int infeasible
01212 (
01213     ABIPWork *w,
01214     ABIPSolution *sol,
01215     ABIPInfo *info,
01216     abip_float bt_y
01217 )
01218 {
01219     DEBUG_FUNC
01220
01221     ABIP(scale_array)(sol->y, 1 / bt_y, w->m);
01222     ABIP(scale_array)(sol->s, 1 / bt_y, w->n);
01223     ABIP(scale_array)(sol->x, NAN, w->n);
01224
01225     if (info->status_val == 0)
01226     {
01227         strcpy(info->status, "Infeasible/Inaccurate");
01228         RETURN ABIP_INFEASIBLE_INACCURATE;
01229     }
01230
01231     strcpy(info->status, "Infeasible");
01232     RETURN ABIP_INFEASIBLE;
01233 }
01234
01235 static abip_int unbounded
01236 (
01237     ABIPWork *w,
01238     ABIPSolution *sol,
01239     ABIPInfo *info,
01240     abip_float ct_x
01241 )
01242 {
01243     DEBUG_FUNC
01244
01245     ABIP(scale_array)(sol->x, -1 / ct_x, w->n);
01246     ABIP(scale_array)(sol->y, NAN, w->m);
01247     ABIP(scale_array)(sol->s, NAN, w->n);
01248
01249     if (info->status_val == 0)
01250     {
01251         strcpy(info->status, "Unbounded/Inaccurate");
01252         RETURN ABIP_UNBOUNDED_INACCURATE;
01253     }
01254
01255     strcpy(info->status, "Unbounded");
01256     RETURN ABIP_UNBOUNDED;
01257 }
01258
01259 static abip_int is_solved_status
01260 (
01261     abip_int status
01262 )
01263 {
01264     RETURN status == ABIP_SOLVED || status == ABIP_SOLVED_INACCURATE;
01265 }
01266
01267 static abip_int is_infeasible_status
01268 (
01269     abip_int status
01270 )
01271 {
01272     RETURN status == ABIP_INFEASIBLE || status == ABIP_INFEASIBLE_INACCURATE;
01273 }
01274
01275 static abip_int is_unbounded_status
01276 (
01277     abip_int status
01278 )
01279 {
01280 }
01281
01282 static abip_int is_unbounded_status
01283 (
01284     abip_int status
01285 )
01286 {
01287 }

```

```

01291     RETURN status == ABIP_UNBOUNDED || status == ABIP_UNBOUNDED_INACCURATE;
01292 }
01296 static void get_info
01297 (
01298     ABIPWork *w,
01299     ABIPSolution *sol,
01300     ABIPInfo *info,
01301     ABIPResiduals *r,
01302     abip_int ipm_iter,
01303     abip_int admm_iter
01304 )
01305 {
01306     DEBUG_FUNC
01307
01308     info->ipm_iter = ipm_iter + 1;
01309     info->admm_iter = admm_iter + 1;
01310
01311     info->res_infeas = r->res_infeas;
01312     info->res_unbdd = r->res_unbdd;
01313
01314     if (is_solved_status(info->status_val))
01315     {
01316         info->rel_gap = r->rel_gap;
01317         info->res_pri = r->res_pri;
01318         info->res_dual = r->res_dual;
01319         info->pobj = r->ct_x_by_tau / r->tau;
01320         info->dobj = r->bt_y_by_tau / r->tau;
01321     }
01322     else if (is_unbounded_status(info->status_val))
01323     {
01324         info->rel_gap = NAN;
01325         info->res_pri = NAN;
01326         info->res_dual = NAN;
01327         info->pobj = -INFINITY;
01328         info->dobj = -INFINITY;
01329     }
01330     else if (is_infeasible_status(info->status_val))
01331     {
01332         info->rel_gap = NAN;
01333         info->res_pri = NAN;
01334         info->res_dual = NAN;
01335         info->pobj = INFINITY;
01336         info->dobj = INFINITY;
01337     }
01338
01339     RETURN;
01340 }
01344 static void get_solution
01345 (
01346     ABIPWork *w,
01347     ABIPSolution *sol,
01348     ABIPInfo *info,
01349     ABIPResiduals *r,
01350     abip_int ipm_iter,
01351     abip_int admm_iter
01352 )
01353 {
01354     DEBUG_FUNC
01355
01356     abip_int l = w->m + w->n + 1;
01357
01358     calc_residuals(w, r, ipm_iter, admm_iter);
01359     setx(w, sol);
01360     sety(w, sol);
01361     sets(w, sol);
01362
01363     abip_float *tmp;
01364
01365     if (w->stgs->avg_criterion)
01366     {
01367         tmp = w->u_avgcon;
01368     }
01369     else
01370     {
01371         tmp = w->u;
01372     }
01373
01374     if (info->status_val == ABIP_UNFINISHED)
01375     {
01376         if (r->tau > INDETERMINATE_TOL && r->tau > r->kap)
01377         {
01378             info->status_val = solved(w, sol, info, r->tau);
01379         }
01380         else if (ABIP(norm)(tmp, l) < INDETERMINATE_TOL * SQRTF((abip_float)l))
01381         {
01382             info->status_val = indeterminate(w, sol, info);
01383         }
01384     }

```

```

01384         else if (-r->bt_y_by_tau < r->ct_x_by_tau)
01385         {
01386             info->status_val = infeasible(w, sol, info, r->bt_y_by_tau);
01387         }
01388         else
01389         {
01390             info->status_val = unbounded(w, sol, info, r->ct_x_by_tau);
01391         }
01392     }
01393     else if (is_solved_status(info->status_val))
01394     {
01395         info->status_val = solved(w, sol, info, r->tau);
01396     }
01397     else if (is_infeasible_status(info->status_val))
01398     {
01399         info->status_val = infeasible(w, sol, info, r->bt_y_by_tau);
01400     }
01401     else
01402     {
01403         info->status_val = unbounded(w, sol, info, r->ct_x_by_tau);
01404     }
01405
01406     if (w->stgs->normalize)
01407     {
01408         ABIP(un_normalize_sol)(w, sol);
01409     }
01410
01411     get_info(w, sol, info, r, ipm_iter, admm_iter);
01412
01413     RETURN;
01414 }
01418 static void print_summary
01419 (
01420     ABIPWork *w,
01421     abip_int i,
01422     abip_int j,
01423     ABIPResiduals *r,
01424     ABIP(timer) *solve_timer
01425 )
01426 {
01427     DEBUG_FUNC
01428
01429     abip_printf("%i|", (int) strlen(HEADER[0]), (int) i);
01430     abip_printf("%i|", (int) strlen(HEADER[1]), (int) j);
01431     abip_printf("%*.2e|", (int) strlen(HEADER[2]), w->mu);
01432
01433     abip_printf("%*.2e|", (int) HSPACE, r->res_pri);
01434     abip_printf("%*.2e|", (int) HSPACE, r->res_dual);
01435     abip_printf("%*.2e|", (int) HSPACE, r->rel_gap);
01436     abip_printf("%*.2e|", (int) HSPACE, SAFEDIV_POS(r->ct_x_by_tau, r->tau));
01437     abip_printf("%*.2e|", (int) HSPACE, SAFEDIV_POS(r->bt_y_by_tau, r->tau));
01438     abip_printf("%*.2e|", (int) HSPACE, SAFEDIV_POS(r->kap, r->tau));
01439     abip_printf("%*.2e ", (int) HSPACE, ABIP(tocq)(solve_timer) / 1e3);
01440     abip_printf("\n");
01441
01442     #if EXTRA_VERBOSE > 0
01443
01444     abip_printf("Norm u = %4f, ", ABIP(norm)(w->u, w->n + w->m + 1));
01445     abip_printf("Norm u_t = %4f, ", ABIP(norm)(w->u_t, w->n + w->m + 1));
01446     abip_printf("Norm v = %4f, ", ABIP(norm)(w->v, w->n + w->m + 1));
01447     abip_printf("tau = %4f, ", r->tau);
01448     abip_printf("kappa = %4f, ", r->kap);
01449     abip_printf("|u - u_prev| = %1.2e, ", ABIP(norm_diff)(w->u, w->u_prev, w->n + w->m + 1));
01450     abip_printf("|u - u_t| = %1.2e, ", ABIP(norm_diff)(w->u, w->u_t, w->n + w->m + 1));
01451     abip_printf("res_infeas = %1.2e, ", r->res_infeas);
01452     abip_printf("res_unbdd = %1.2e\n", r->res_unbdd);
01453
01454     #endif
01455
01456     #ifdef MATLAB_MEX_FILE
01457
01458     mexEvalString("drawnow;");
01459
01460     #endif
01461
01462     RETURN;
01463 }
01467 static void print_header
01468 (
01469     ABIPWork *w
01470 )
01471 {
01472     DEBUG_FUNC
01473
01474     abip_int i;
01475
01476     if (w->stgs->warm_start)

```

```

01477     {
01478         abip_printf("ABIP using variable warm-starting\n");
01479     }
01480
01481     for (i = 0; i < LINE_LEN; ++i)
01482     {
01483         abip_printf("-");
01484     }
01485     abip_printf("\n");
01486
01487     for (i = 0; i < HEADER_LEN - 1; ++i)
01488     {
01489         abip_printf("%s|", HEADER[i]);
01490     }
01491     abip_printf("%s\n", HEADER[HEADER_LEN - 1]);
01492
01493     for (i = 0; i < LINE_LEN; ++i)
01494     {
01495         abip_printf("-");
01496     }
01497     abip_printf("\n");
01498
01499 #ifdef MATLAB_MEX_FILE
01500     mexEvalString("drawnow;");
01501
01502 #endif
01503
01504     RETURN;
01505 }
01506
01507 static void print_footer
01508 (
01509     const ABIPData *d,
01510     ABIPSolution *sol,
01511     ABIPWork *w,
01512     ABIPInfo *info
01513 )
01514 {
01515     DEBUG_FUNC
01516
01517     abip_int i;
01518
01519     char *lin_sys_str = ABIP(get_lin_sys_summary)(w->p, info);
01520     char *adapt_str = ABIP(get_adapt_summary)(info, w->adapt);
01521
01522     for (i = 0; i < LINE_LEN; ++i)
01523     {
01524         abip_printf("-");
01525     }
01526
01527     abip_printf("\n");
01528
01529     abip_printf("Status: %s\n", info->status);
01530
01531     if (info->ipm_iter+1 == w->stgs->max_ipm_iters)
01532     {
01533         abip_printf("Hit max_ipm_iters, solution may be inaccurate\n");
01534     }
01535
01536     if (info->admm_iter+1 >= w->stgs->max_admm_iters)
01537     {
01538         abip_printf("Hit max_admm_iters, solution may be inaccurate\n");
01539     }
01540
01541     abip_printf("Timing: Solve time: %1.2es\n", info->solve_time / 1e3);
01542
01543     if (lin_sys_str)
01544     {
01545         abip_printf("%s", lin_sys_str);
01546         abip_free(lin_sys_str);
01547     }
01548
01549     if (adapt_str)
01550     {
01551         abip_printf("%s", adapt_str);
01552         abip_free(adapt_str);
01553     }
01554
01555     for (i = 0; i < LINE_LEN; ++i)
01556     {
01557         abip_printf("-");
01558     }
01559
01560     abip_printf("\n");
01561
01562     if (is_infeasible_status(info->status_val))
01563     {

```



```

01567         abip_printf("Certificate of primal infeasibility:\n");
01568         abip_printf("||A'y + s||_2 * ||b||_2 = %.4e\n", info->res_infeas);
01569         abip_printf("b'y = %.4f\n", ABIP(dot)(d->b, sol->y, d->m));
01570     }
01571     else if (is_unbounded_status(info->status_val))
01572     {
01573         abip_printf("Certificate of dual infeasibility:\n");
01574         abip_printf("||Ax||_2 * ||c||_2 = %.4e\n", info->res_unbdd);
01575         abip_printf("c'x = %.4f\n", ABIP(dot)(d->c, sol->x, d->n));
01576     }
01577     else
01578     {
01579         abip_printf("Error metrics:\n");
01580         abip_printf("primal res: ||Ax - b||_2 / (1 + ||b||_2) = %.4e\n", info->res_pri);
01581         abip_printf("dual res: ||A'y + s - c||_2 / (1 + ||c||_2) = %.4e\n", info->res_dual);
01582         abip_printf("rel gap: |c'x - b'y| / (1 + |c'x| + |b'y|) = %.4e\n", info->rel_gap);
01583
01584         for (i = 0; i < LINE_LEN; ++i)
01585         {
01586             abip_printf("-");
01587         }
01588
01589         abip_printf("\n");
01590         abip_printf("c'x = %.4e, b'y = %.4e\n", info->pobj, info->dobj);
01591     }
01592
01593     for (i = 0; i < LINE_LEN; ++i)
01594     {
01595         abip_printf("=");
01596     }
01597
01598     abip_printf("\n");
01599
01600 #ifdef MATLAB_MEX_FILE
01601     mexEvalString("drawnow;");
01602
01603 #endif
01604     RETURN;
01605 }
01606
01607 static abip_int has_converged
01608 (
01609     ABIPWork *w,
01610     ABIPResiduals *r,
01611     abip_int ipm_iter,
01612     abip_int admm_iter
01613 )
01614 {
01615     DEBUG_FUNC
01616
01617     abip_float eps = w->stgs->eps;
01618
01619     if (r->res_pri < eps && (r->res_dual < eps || w->stgs->pfeasopt) && r->rel_gap < eps)
01620     {
01621         RETURN ABIP_SOLVED;
01622     }
01623
01624     if (r->res_unbdd < eps && ipm_iter > 0 && admm_iter > 0)
01625     {
01626         RETURN ABIP_UNBOUNDED;
01627     }
01628
01629     if (r->res_infeas < eps && ipm_iter > 0 && admm_iter > 0)
01630     {
01631         RETURN ABIP_INFEASIBLE;
01632     }
01633
01634     RETURN 0;
01635 }
01636
01637 static abip_int validate
01638 (
01639     const ABIPData *d
01640 )
01641 {
01642     DEBUG_FUNC
01643
01644     ABIPSettings *stgs = d->stgs;
01645
01646     if (d->m <= 0 || d->n <= 0)
01647     {
01648         abip_printf("m and n must both be greater than 0; m = %li, n = %li\n", (long) d->m, (long)
01649             d->n);
01650         RETURN - 1;
01651     }
01652 }

```

```

01660
01661     if (d->m > d->n)
01662     {
01663         abip_printf("WARN: m larger than n, problem likely degenerate\n");
01664         RETURN - 1;
01665     }
01666
01667     if (ABIP(validate_lin_sys)(d->A) < 0)
01668     {
01669         abip_printf("invalid linear system input data\n");
01670         RETURN - 1;
01671     }
01672
01673     if (stgs->max_ipm_iters <= 0)
01674     {
01675         abip_printf("max_ipm_iters must be positive\n");
01676         RETURN - 1;
01677     }
01678
01679     if (stgs->max_admm_iters <= 0)
01680     {
01681         abip_printf("max_admm_iters must be positive\n");
01682         RETURN - 1;
01683     }
01684
01685     if (stgs->eps <= 0)
01686     {
01687         abip_printf("eps tolerance must be positive\n");
01688         RETURN - 1;
01689     }
01690
01691     if (stgs->alpha <= 0 || stgs->alpha >= 2)
01692     {
01693         abip_printf("alpha must be in (0,2)\n");
01694         RETURN - 1;
01695     }
01696
01697     if (stgs->rho_y <= 0)
01698     {
01699         abip_printf("rho_y must be positive (1e-3 works well).\n");
01700         RETURN - 1;
01701     }
01702
01703     if (stgs->scale <= 0)
01704     {
01705         abip_printf("scale must be positive (1 works well).\n");
01706         RETURN - 1;
01707     }
01708
01709     if (stgs->eps_cor <= 0)
01710     {
01711         abip_printf("eps_cor tolerance must be positive.\n");
01712         RETURN - 1;
01713     }
01714
01715     if (stgs->eps_pen <= 0)
01716     {
01717         abip_printf("eps_pen tolerance must be positive.\n");
01718         RETURN - 1;
01719     }
01720
01721     if (stgs->adaptive_lookback <= 0)
01722     {
01723         abip_printf("adaptive_lookback must be positive.\n");
01724         RETURN - 1;
01725     }
01726
01727     if (stgs->hybrid_mu > 0 && stgs->dynamic_sigma >= 0 )
01728     {
01729         abip_printf("when use hybrid mu strategy, dynamic_sigma must be negative.\n");
01730         RETURN - 1;
01731     }
01732
01733     RETURN 0;
01734 }
01735
01736 static ABIPWork *init_work
01737 (
01738     const ABIPData *d
01739 )
01740 {
01741     DEBUG_FUNC
01742
01743     ABIPWork *w = (ABIPWork *) abip_calloc(1, sizeof(ABIPWork));
01744     abip_int l = d->n + d->m + 1;
01745
01746     if (d->stgs->verbose)

```

```

01750     {
01751         print_init_header(d);
01752     }
01753
01754     if (!w)
01755     {
01756         abip_printf("ERROR: allocating work failure\n");
01757         RETURN ABIP_NULL;
01758     }
01759
01760     w->stgs = d->stgs;
01761     w->m = d->m;
01762     w->n = d->n;
01763
01764     w->u = (abip_float *) abip_malloc(1 * sizeof(abip_float));
01765     w->v = (abip_float *) abip_malloc(1 * sizeof(abip_float));
01766     w->u_t = (abip_float *) abip_malloc(1 * sizeof(abip_float));
01767     w->u_prev = (abip_float *) abip_malloc(1 * sizeof(abip_float));
01768     w->v_prev = (abip_float *) abip_malloc(1 * sizeof(abip_float));
01769     w->u_avg = (abip_float*) abip_malloc(1 * sizeof(abip_float));
01770     w->v_avg = (abip_float*)abip_malloc(1 * sizeof(abip_float));
01771
01772     w->u_avgcon = (abip_float*) abip_malloc(1 * sizeof(abip_float));
01773     w->v_avgcon = (abip_float*)abip_malloc(1 * sizeof(abip_float));
01774     w->u_sumcon = (abip_float*) abip_malloc(1 * sizeof(abip_float));
01775     w->v_sumcon = (abip_float*)abip_malloc(1 * sizeof(abip_float));
01776
01777
01778     w->h = (abip_float *) abip_malloc((1 - 1) * sizeof(abip_float));
01779     w->g = (abip_float *) abip_malloc((1 - 1) * sizeof(abip_float));
01780     w->pr = (abip_float *) abip_malloc(d->m * sizeof(abip_float));
01781     w->dr = (abip_float *) abip_malloc(d->n * sizeof(abip_float));
01782     w->b = (abip_float *) abip_malloc(d->m * sizeof(abip_float));
01783     w->c = (abip_float *) abip_malloc(d->n * sizeof(abip_float));
01784
01785     if (!w->u || !w->v || !w->u_t || !w->u_prev || !w->h || !w->g || !w->pr || !w->dr || !w->b ||
!w->c)
01786     {
01787         abip_free(w->u); abip_free(w->v); abip_free(w->u_t); abip_free(w->u_prev);
01788         abip_free(w->h); abip_free(w->g); abip_free(w->pr); abip_free(w->dr);
01789         abip_free(w->b); abip_free(w->c);
01790         abip_printf("ERROR: work memory allocation failure\n");
01791         RETURN ABIP_NULL;
01792     }
01793
01794     w->A = d->A;
01795     w->sp = d->sp;
01796
01797     if (w->stgs->normalize)
01798     {
01799 #ifdef COPYAMATRIX
01800
01801         if (!ABIP(copy_A_matrix) (&(w->A), d->A))
01802         {
01803             abip_printf("ERROR: copy A matrix failed\n");
01804             RETURN ABIP_NULL;
01805         }
01806
01807 #endif
01808
01809         w->scal = (ABIPScaling *)abip_malloc(sizeof(ABIPScaling));
01810         ABIP(normalize_A) (w->A, w->stgs, w->scal);
01811
01812 #if EXTRA_VERBOSE > 0
01813
01814         ABIP(print_array) (w->scal->D, d->m, "D");
01815         abip_printf("ABIP (norm) D = %4f\n", ABIP(norm) (w->scal->D, d->m));
01816         ABIP(print_array) (w->scal->E, d->n, "E");
01817         abip_printf("ABIP (norm) E = %4f\n", ABIP(norm) (w->scal->E, d->n));
01818
01819 #endif
01820     }
01821     else
01822     {
01823         w->scal = ABIP_NULL;
01824     }
01825
01826     if (!(w->p = ABIP(init_lin_sys_work) (w->A, w->stgs)))
01827     {
01828         abip_printf("ERROR: init_lin_sys_work failure\n");
01829         RETURN ABIP_NULL;
01830     }
01831
01832     if (!(w->adapt = ABIP(init_adapt) (w)))
01833     {
01834         abip_printf("ERROR: init_adapt failure\n");
01835         RETURN ABIP_NULL;

```

```

01836     }
01837
01838     RETURN w;
01839 }
01843 static abip_int update_work
01844 (
01845     const ABIPData *d,
01846     ABIPWork *w,
01847     const ABIPSolution *sol
01848 )
01849 {
01850     DEBUG_FUNC
01851
01852     abip_int n = d->n;
01853     abip_int m = d->m;
01854
01855     w->nm_b = ABIP(norm) (d->b, m);
01856     w->nm_c = ABIP(norm) (d->c, n);
01857     memcpy(w->b, d->b, d->m * sizeof(abip_float));
01858     memcpy(w->c, d->c, d->n * sizeof(abip_float));
01859
01860     #if EXTRA_VERBOSE > 0
01861
01862     ABIP(print_array) (w->b, m, "b");
01863     abip_printf("pre-normalized norm b = %4f\n", ABIP(norm) (w->b, m));
01864     ABIP(print_array) (w->c, n, "c");
01865     abip_printf("pre-normalized norm c = %4f\n", ABIP(norm) (w->c, n));
01866
01867     #endif
01868
01869     if (w->stgs->normalize)
01870     {
01871         ABIP(normalize_b_c) (w);
01872
01873     #if EXTRA_VERBOSE > 0
01874
01875     ABIP(print_array) (w->b, m, "bn");
01876     abip_printf("sc_b = %4f\n", w->sc_b);
01877     abip_printf("post-normalized norm b = %4f\n", ABIP(norm) (w->b, m));
01878
01879     ABIP(print_array) (w->c, n, "cn");
01880     abip_printf("sc_c = %4f\n", w->sc_c);
01881     abip_printf("post-normalized norm c = %4f\n", ABIP(norm) (w->c, n));
01882
01883     #endif
01884     }
01885
01886     if (MAX(w->sp, w->stgs->sparsity_ratio) > 0.4 || (MIN(w->sp, w->stgs->sparsity_ratio) > 0.1 &&
MIN(w->sp, w->stgs->sparsity_ratio) < 0.2))
01887     {
01888         w->sigma = 0.3;
01889         w->gamma = 2.0;
01890     }
01891     else if (MIN(w->sp, w->stgs->sparsity_ratio) > 0.2)
01892     {
01893         w->sigma = 0.5;
01894         w->gamma = 3.0;
01895     }
01896     else
01897     {
01898         w->sigma = 0.8;
01899         w->gamma = 3.0;
01900     }
01901
01902     w->final_check = 0;
01903     w->double_check = 0;
01904
01905     w->mu = 1.0;
01906     w->beta = 1.0;
01907
01908     if (w->stgs->warm_start)
01909     {
01910         warm_start_vars(w, sol);
01911     }
01912     else
01913     {
01914         cold_start_vars(w);
01915     }
01916
01917     memcpy(w->h, w->b, m * sizeof(abip_float));
01918     memcpy(&(w->h[m]), w->c, n * sizeof(abip_float));
01919     ABIP(scale_array) (w->h, -1, m);
01920     memcpy(w->g, w->h, (n + m) * sizeof(abip_float));
01921
01922     ABIP(solve_lin_sys) (w->A, w->stgs, w->p, w->g, ABIP_NULL, -1); // solve the linear system
01923     ABIP(scale_array) (&(w->g[m]), -1, n);
01924     w->g_th = ABIP(dot) (w->h, w->g, n + m);

```

```

01925
01926     RETURN 0;
01927 }
01928
01932 static abip_float iterate_norm_diff
01933 (
01934     ABIPWork *w
01935 )
01936 {
01937     DEBUG_FUNC
01938
01939     abip_int l = w->m + w->n + 1;
01940
01941     abip_float u_norm_difference = ABIP(norm_diff)(w->u, w->u_prev, l);
01942     abip_float v_norm_difference = ABIP(norm_diff)(w->v, w->v_prev, l);
01943     abip_float norm = 1 + SQRTF(ABIP(norm_sq)(w->u, l) + ABIP(norm_sq)(w->v, l)) +
01944     SQRTF(ABIP(norm_sq)(w->u_prev, l) + ABIP(norm_sq)(w->v_prev, l));
01944     abip_float norm_diff = SQRTF(u_norm_difference * u_norm_difference + v_norm_difference *
01945     v_norm_difference);
01946
01947     RETURN norm_diff / norm;
01948 }
01951 static abip_float iterate_Q_norm_resd
01952 (
01953     ABIPWork *w,
01954     abip_int j
01955 )
01956 {
01957     DEBUG_FUNC
01958
01959     abip_int i;
01960     abip_int l = w->m + w->n + 1;
01961
01962     abip_float *y = w->u;
01963     abip_float *x = &(w->u[w->m]);
01964     abip_float *s = &(w->v[w->m]);
01965     abip_float tau = w->u[w->m + w->n];
01966     abip_float kap = w->v[w->m + w->n];
01967
01968     abip_float Qres = 0;
01969     abip_float Qres_avg = w->stgs->max_admm_iters;
01970     abip_float norm_avg = 1;
01971
01972     // initialize pr and dr
01973     abip_float *pr = w->pr;
01974     abip_float *dr = w->dr;
01975
01976     memset(pr, 0, w->m * sizeof(abip_float));
01977     memset(dr, 0, w->n * sizeof(abip_float));
01978
01979     // compute pr and dr
01980     ABIP(accum_by_A)(w->A, w->p, x, pr);
01981     ABIP(accum_by_Atrans)(w->A, w->p, y, dr);
01982     ABIP(add_scaled_array)(dr, s, w->n, 1.0);
01983
01984     for (i = 0; i < w->m; ++i)
01985     {
01986         Qres += (pr[i] - w->b[i] * tau) * (pr[i] - w->b[i] * tau);
01987     }
01988
01989     for (i = 0; i < w->n; ++i)
01990     {
01991         Qres += (dr[i] - w->c[i] * tau) * (dr[i] - w->c[i] * tau);
01992     }
01993     abip_float cTx = ABIP(dot)(x, w->c, w->n);
01994     abip_float bTy = ABIP(dot)(y, w->b, w->m);
01995     Qres += (bTy - cTx - kap) * (bTy - cTx - kap);
01996     abip_float norm = 1 + SQRTF(ABIP(norm_sq)(w->u, l) + ABIP(norm_sq)(w->v, l)) ;
01997
01998
01999     // check inner loop termination criterion via the average solution returned by restart strategy
02000     if ( (j+1) % 10 == 0)
02001     {
02002         // initialize pr_avg and dr_avg
02003         Qres_avg = 0;
02004         norm_avg = 0;
02005         abip_float *y_avg = w->u_avgcon;
02006         abip_float *x_avg = &(w->u_avgcon[w->m]);
02007         abip_float *s_avg = &(w->v_avgcon[w->m]);
02008         abip_float tau_avg = w->u_avgcon[w->m + w->n];
02009         abip_float kap_avg = w->v_avgcon[w->m + w->n];
02010         abip_float *pr_avg;
02011         abip_float *dr_avg;
02012         pr_avg = (abip_float *) abip_malloc(w->m * sizeof(abip_float));
02013         dr_avg = (abip_float *) abip_malloc(w->n * sizeof(abip_float));
02014         memset(pr_avg, 0, w->m * sizeof(abip_float));
02015         memset(dr_avg, 0, w->n * sizeof(abip_float));

```

```

02016
02017 // compute pr_avg and dr_avg
02018 ABIP(accum_by_A)(w->A, w->p, x_avg, pr_avg);
02019 ABIP(accum_by_Atrans)(w->A, w->p, y_avg, dr_avg);
02020 ABIP(add_scaled_array)(dr_avg, s_avg, w->n, 1.0);
02021 for (i = 0; i < w->m; ++i)
02022 {
02023     Qres_avg += (pr_avg[i] - w->b[i] * tau_avg) * (pr_avg[i] - w->b[i] * tau_avg);
02024 }
02025
02026 for (i = 0; i < w->n; ++i)
02027 {
02028     Qres_avg += (dr_avg[i] - w->c[i] * tau_avg) * (dr_avg[i] - w->c[i] * tau_avg);
02029 }
02030
02031 abip_float cTx_avg = ABIP(dot)(x_avg, w->c, w->n);
02032 abip_float bTy_avg = ABIP(dot)(y_avg, w->b, w->m);
02033
02034 Qres_avg += (bTy_avg - cTx_avg - kap_avg) * (bTy_avg - cTx_avg - kap_avg);
02035
02036 norm_avg = 1 + SQRTF(ABIP(norm_sq)(w->u_avgcon, 1) + ABIP(norm_sq)(w->v_avgcon, 1));
02037 abip_free(pr_avg); abip_free(dr_avg);
02038 }
02039
02040 if (SQRTF(Qres_avg) / norm_avg < SQRTF(Qres) / norm)
02041 {
02042     w->stgs->avg_criterion = 1;
02043
02044     RETURN SQRTF(Qres_avg) / norm_avg;
02045 }
02046 else
02047 {
02048     w->stgs->avg_criterion = 0;
02049     RETURN SQRTF(Qres) / norm;
02050 }
02051 }
02052
02056 abip_int ABIP(solve)
02057 {
02058     ABIPWork *w,
02059     const ABIPData *d,
02060     ABIPSolution *sol,
02061     ABIPInfo *info
02062 )
02063 {
02064     DEBUG_FUNC
02065
02066     abip_int i;
02067     abip_int j;
02068     abip_int k;
02069     abip_int ii;
02070     abip_int inner_stopper;
02071     ABIP(timer) solve_timer;
02072
02073     abip_int jj;
02074
02075     ABIPResiduals r;
02076     abip_int l = w->m + w->n + 1;
02077
02078     if (!d || !sol || !info || !w || !d->b || !d->c)
02079     {
02080         abip_printf("ERROR: ABIP_NULL input\n");
02081         RETURN ABIP_FAILED;
02082     }
02083
02084     clock_t start_time = clock();
02085     double elapsedT = 0.0, maxTime = w->stgs->max_time;
02086
02087     abip_start_interrupt_listener();
02088     ABIP(tic)(&solve_timer);
02089
02090     info->status_val = ABIP_UNFINISHED;
02091     r.last_ipm_iter = -1;
02092     r.last_admm_iter = -1;
02093     update_work(d, w, sol);
02094
02095     if (w->stgs->verbose)
02096     {
02097         print_header(w);
02098     }
02099
02100     k = 0;
02101
02102     for (i = 0; i < w->stgs->max_ipm_iters; ++i) // the outer loop
02103     {
02104         if (MIN(w->sp, w->stgs->sparsity_ratio) > 0.5) // determine the # iteratin of inner loop
02105         {

```

```

02106         inner_stopper = (int)round(POWF(w->mu, -0.35));
02107     }
02108     else if (MIN(w->sp, w->stgs->sparsity_ratio) > 0.2)
02109     {
02110         inner_stopper = (int)round(POWF(w->mu, -1));
02111     }
02112     else
02113     {
02114         inner_stopper = w->stgs->max_admm_iters;
02115     }
02116     w->fre_old = 0;
02117     memset(w->u_avg, 0, sizeof(abip_float) * 1);
02118     memset(w->v_avg, 0, sizeof(abip_float) * 1);
02119
02120
02121     memset(w->u_sumcon, 0, sizeof(abip_float) * 1);
02122     memset(w->v_sumcon, 0, sizeof(abip_float) * 1);
02123
02124
02125     if (w->stgs->avg_criterion)
02126     {
02127         memcpy(w->u, w->u_avgcon, sizeof(abip_float) * 1);
02128         memcpy(w->v, w->v_avgcon, sizeof(abip_float) * 1);
02129     }
02130
02131     for (j = 0; j < inner_stopper; ++j) // the inner loop
02132     {
02133         memcpy(w->u_prev, w->u, 1 * sizeof(abip_float));
02134         memcpy(w->v_prev, w->v, 1 * sizeof(abip_float));
02135
02136         // update variables
02137         if (project_lin_sys(w, k) < 0)
02138         {
02139             RETURN failure(w, w->m, w->n, sol, info, ABIP_FAILED, "error in project_lin_sys",
02140 "Failure");
02141         }
02142
02143         if (w->stgs->half_update) // half update
02144         {
02145             half_update_dual_vars(w);
02146
02147             project_barrier_dual(w);
02148         }
02149
02150         else // normal update
02151         {
02152             project_barrier(w);
02153
02154             update_dual_vars(w);
02155         }
02156
02157         // restart
02158         restart_vars(w, j, k);
02159
02160
02161         if (abip_is_interrupted())
02162         {
02163             RETURN failure(w, w->m, w->n, sol, info, ABIP_SIGINT, "Interrupted", "Interrupted");
02164         }
02165
02166         // compute average solution
02167         compute_avg(w, j);
02168
02169
02170         k += 1;
02171
02172
02173         if (iterate_Q_norm_resd(w, j) < w->gamma*w->mu) // inner loop termination criterion
02174         {
02175             if (w->stgs->half_update)
02176             {
02177                 for (jj=0; jj<1; ++jj)
02178                 {
02179                     if( w->v[jj] < 0 )
02180                     {
02181                         w->v[jj] = 1e-6;
02182                         // abip_printf("find a negative element\n");
02183                     }
02184                 }
02185             }
02186
02187             break;
02188         }
02189         // check whether the inner loop iterate has satisfied the tolerance level
02190         if (w-> final_check && (j+1) % CONVERGED_INTERVAL == 0)
02191         {

```

```

02192         calc_residuals(w, &r, i, k); // compute residuals
02193
02194         if ((info->status_val = has_converged(w, &r, i, k)) != 0 || k+1 >=
w->stgs->max_admm_iters || i+1 >= w->stgs->max_ipm_iters)
02195         {
02196             if (w->stgs->verbose && k>0)
02197             {
02198                 print_summary(w, i, k, &r, &solve_timer); // converge
02199             }
02200
02201             get_solution(w, sol, info, &r, i, k);
02202             info->solve_time = ABIP(tocq)(&solve_timer);
02203
02204             if (w->stgs->verbose)
02205             {
02206                 print_footer(d, sol, w, info);
02207             }
02208
02209             abip_end_interrupt_listener();
02210
02211             RETURN info->status_val;
02212         }
02213     }
02214 }
02215
02216 elapsedT = ((abip_float)clock() - start_time) / CLOCKS_PER_SEC; // record runtime
02217 if (elapsedT > maxTime) {
02218     abip_printf("Timelimit reached. \n");
02219     w->stgs->max_admm_iters = k * 1.05;
02220 }
02221
02222 // early stopping strategy
02223 if (w->mu < w->stgs->eps)
02224 {
02225     w->final_check = 1;
02226 }
02227
02228 // check whether the iterate satisfies the tolerance level in the outer loop
02229 calc_residuals(w, &r, i, k);
02230 if (w->stgs->verbose)
02231 {
02232     print_summary(w, i, k, &r, &solve_timer);
02233 }
02234
02235 if ((info->status_val = has_converged(w, &r, i, k)) != 0 || k+1 >= w->stgs->max_admm_iters)
02236 {
02237     get_solution(w, sol, info, &r, i, k);
02238     info->solve_time = ABIP(tocq)(&solve_timer);
02239
02240     if (w->stgs->verbose)
02241     {
02242         print_footer(d, sol, w, info);
02243     }
02244
02245     abip_end_interrupt_listener();
02246
02247     RETURN info->status_val;
02248 }
02249
02250 // update mu
02251 if (w->stgs->hybrid_mu)
02252 {
02253     if (w->stgs->dynamic_sigma_second > 0.0 && w->mu < w->stgs->hybrid_thresh * w->stgs->eps)
02254     {
02255         w->stgs->dynamic_sigma = w->stgs->dynamic_sigma_second;
02256         update_barrier_dynamic(w, &r);
02257     }
02258     else if (w->stgs->dynamic_sigma_second == 0.0 && w->mu < w->stgs->hybrid_thresh *
w->stgs->eps) {
02259         w->stgs->dynamic_sigma = w->stgs->dynamic_sigma_second;
02260         update_barrier(w, &r);
02261     }
02262     else if (w->stgs->dynamic_sigma < 0.0) {
02263         update_barrier_dynamic_2(w);
02264     }
02265 }
02266 else
02267 {
02268     if (w->stgs->dynamic_sigma == 0.0) {
02269         update_barrier(w, &r);
02270     }
02271     else if (w->stgs->dynamic_sigma < 0.0) {
02272         update_barrier_dynamic_2(w);
02273     }
02274     else {
02275         update_barrier_dynamic(w, &r);

```



```

02276         }
02277     }
02278     // prepare the next outer loop
02279     reinitialize_vars(w, 0);
02280
02281     if (w->stgs->adaptive)
02282     {
02283         reinitialize_vars(w, 1);
02284
02285         w->beta = 1;
02286
02287         if (ABIP(adaptive)(w, k) < 0)
02288         {
02289             RETURN failure(w, w->m, w->n, sol, info, ABIP_FAILED, "error in adaptive", "Failure");
02290         }
02291
02292         reinitialize_vars(w, 2);
02293     }
02294 }
02295
02296 RETURN info->status_val; // return status
02297 }
02298
02299 /* @brief recover the optimal solution and set memory free
02300 */
02301 void ABIP(finish)
02302 {
02303     ABIPWork *w
02304 }
02305 {
02306     DEBUG_FUNC
02307
02308     if (w)
02309     {
02310         if (w->stgs && w->stgs->normalize)
02311         {
02312             #ifndef COPYAMATRIX
02313                 ABIP(un_normalize_A)(w->A, w->stgs, w->scal);
02314             #else
02315                 ABIP(free_A_matrix)(w->A);
02316             #endif
02317         }
02318
02319         if (w->p)
02320         {
02321             #ifdef ABIP_PARDISO
02322                 ABIP(free_lin_sys_work_pds)(w->p, w->A);
02323             #else
02324                 ABIP(free_lin_sys_work)(w->p);
02325             #endif
02326         }
02327
02328         if (w->adapt)
02329         {
02330             ABIP(free_adapt)(w->adapt);
02331         }
02332
02333         free_work(w);
02334     }
02335
02336     RETURN;
02337 }
02338
02339 /* @brief initialization
02340 */
02341 ABIPWork *ABIP(init)
02342 {
02343     const ABIPData *d,
02344     ABIPInfo *info
02345 }
02346 {
02347     DEBUG_FUNC
02348
02349     #if EXTRA_VERBOSE > 1
02350         ABIP(tic)(&global_timer);
02351     #endif
02352
02353     ABIPWork *w;
02354     ABIP(timer) init_timer;
02355     abip_start_interrupt_listener();
02356
02357     if (!d || !info)
02358     {
02359         abip_printf("ERROR: Missing ABIPData or ABIPInfo input\n");
02360         RETURN ABIP_NULL;
02361     }
02362

```

```

02363 #if EXTRA_VERBOSE > 0
02364     ABIP(print_data) (d);
02365 #endif
02366
02367 #ifndef NOVALIDATE
02368     if (validate(d) < 0)
02369     {
02370         abip_printf("ERROR: Validation returned failure\n");
02371         RETURN ABIP_NULL;
02372     }
02373 #endif
02374
02375     ABIP(tic) (&init_timer);
02376
02377     w = init_work(d); // initialization
02378     info->setup_time = ABIP(tocq) (&init_timer);
02379
02380     if (d->stgs->verbose)
02381     {
02382         abip_printf("Setup time: %1.2es\n", info->setup_time / 1e3); // printf information
02383     }
02384
02385     abip_end_interrupt_listener();
02386
02387     RETURN w;
02388 }
02389
02393 abip_int ABIP(main)
02394 {
02395     const ABIPData *d,
02396     ABIPSolution *sol,
02397     ABIPInfo *info
02398 }
02399 {
02400     DEBUG_FUNC
02401
02402     abip_int status;
02403     ABIPWork *w = ABIP(init) (d, info);
02404
02405     #if EXTRA_VERBOSE > 0
02406         abip_printf("size of abip_int = %lu, size of abip_float = %lu\n", sizeof(abip_int),
02407             sizeof(abip_float));
02408     #endif
02409
02409     if (w)
02410     {
02411         ABIP(solve) (w, d, sol, info); // solve the problem via abip
02412         status = info->status_val;
02413     }
02414     else
02415     {
02416         status = failure(ABIP_NULL, d ? d->m : -1, d ? d->n : -1, sol, info, ABIP_FAILED, "could not
02417             initialize work", "Failure");
02418     }
02419
02419     ABIP(finish) (w);
02420
02421     RETURN status;
02422 }

```

5.94 src/abip_version.c File Reference

```
#include "glbopts.h"
```

Functions

- const char *[ABIP\(\) version](#) (void)
return the abip version

5.94.1 Function Documentation

5.94.1.1 version()

```
const char *ABIP() version (
    void )
```

return the abip version

Definition at line 5 of file [abip_version.c](#).

5.95 abip_version.c

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

```
00001 #include "glbopts.h"
00005 const char *ABIP(version) (void)
00006 {
00007     return ABIP_VERSION;
00008 }
```

5.96 src/adaptive.c File Reference

```
#include "adaptive.h"
#include "linalg.h"
#include "linsys.h"
#include "abip.h"
#include "abip_blas.h"
#include "util.h"
```

Data Structures

- struct [ABIP_ADAPTIVE_WORK](#)

Functions

- [ABIPAdaptWork](#) *ABIP() [init_adapt](#) ([ABIPWork](#) *w)
- [abip_int](#) ABIP() [adaptive](#) ([ABIPWork](#) *w, [abip_int](#) iter)
- void ABIP() [free_adapt](#) ([ABIPAdaptWork](#) *a)
- char *ABIP() [get_adapt_summary](#) (const [ABIPInfo](#) *info, [ABIPAdaptWork](#) *a)

5.96.1 Function Documentation

5.96.1.1 adaptive()

```
abip_int ABIP() adaptive (
    ABIPWork * w,
    abip_int iter )
```

Definition at line 305 of file [adaptive.c](#).

5.96.1.2 free_adapt()

```
void ABIP() free_adapt (
    ABIPAdaptWork * a )
```

Definition at line 336 of file [adaptive.c](#).

5.96.1.3 get_adapt_summary()

```
char *ABIP() get_adapt_summary (
    const ABIPInfo * info,
    ABIPAdaptWork * a )
```

Definition at line 406 of file [adaptive.c](#).

5.96.1.4 init_adapt()

```
ABIPAdaptWork *ABIP() init_adapt (
    ABIPWork * w )
```

Definition at line 258 of file [adaptive.c](#).

5.97 adaptive.c

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

```
00001 #include "adaptive.h"
00002 #include "linalg.h"
00003 #include "linsys.h"
00004 #include "abip.h"
00005 #include "abip_blas.h"
00006 #include "util.h"
00007
00008 /* This file uses adaption to improve the convergence rate of the ADMM in each inner loop.
00009  * At each iteration we need to select a nearly-optimal penalty parameter beta, we do this using
00010  * Barzilai-Borwein spectral method.
00011  * Adaptive_lookback is the number of lookback iterations.
00012  */
00013 struct ABIP_ADAPTIVE_WORK
00014 {
00015     abip_float *u_prev;
00016     abip_float *v_prev;
00017     abip_float *ut;
00018     abip_float *u;
00019     abip_float *v;
00020     abip_float *ut_next;
00021     abip_float *u_next;
00022     abip_float *v_next;
00023
00024     abip_float *delta_ut;
00025     abip_float *delta_u;
00026     abip_float *delta_v;
00027
00028     abip_int l;
00029     abip_int k;
00030
00031     abip_float total_adapt_time;
00032 };
00033
```

```

00034 static abip_int update_adapt_params
00035 {
00036     ABIPWork *w,
00037     abip_int iter
00038 }
00039 {
00040     DEBUG_FUNC
00041
00042     abip_float *u_prev = w->adapt->u_prev;
00043     abip_float *v_prev = w->adapt->v_prev;
00044     abip_float *ut = w->adapt->ut;
00045     abip_float *u = w->adapt->u;
00046     abip_float *v = w->adapt->v;
00047     abip_float *ut_next = w->adapt->ut_next;
00048     abip_float *u_next = w->adapt->u_next;
00049     abip_float *v_next = w->adapt->v_next;
00050
00051     abip_float *delta_ut = w->adapt->delta_ut;
00052     abip_float *delta_u = w->adapt->delta_u;
00053     abip_float *delta_v = w->adapt->delta_v;
00054
00055     abip_int l = w->adapt->l;
00056     abip_int n = w->n;
00057     abip_int m = w->m;
00058     abip_int k = w->adapt->k;
00059     abip_int i;
00060     abip_int j;
00061
00062     abip_int status_1 = 0;
00063     abip_int status_2 = 0;
00064
00065     abip_float tmp;
00066     abip_float beta_prev = 1.0;
00067     abip_float beta = 0.0;
00068
00069     abip_float uu;
00070     abip_float uv;
00071     abip_float vv;
00072     abip_float utut;
00073     abip_float utv;
00074     abip_float norm_ut;
00075     abip_float norm_u;
00076     abip_float norm_v;
00077
00078     abip_float alpha_SD;
00079     abip_float alpha_MG;
00080     abip_float gamma_SD;
00081     abip_float gamma_MG;
00082     abip_float alpha_cor;
00083     abip_float gamma_cor;
00084     abip_float alpha_ss;
00085     abip_float gamma_ss;
00086
00087     memcpy(u_prev, w->u, sizeof(abip_float) * l);
00088     memcpy(v_prev, w->v, sizeof(abip_float) * l);
00089
00090     for (i = 0; i < k; ++i)
00091     {
00092         memcpy(ut, u_prev, l * sizeof(abip_float));
00093         ABIP(add_scaled_array)(ut, v_prev, l, 1.0);
00094         ABIP(scale_array)(ut, w->stgs->rho_y, m);
00095         ABIP(add_scaled_array)(ut, w->h, l - 1, -ut[l - 1]);
00096         ABIP(add_scaled_array)(ut, w->h, l - 1, -ABIP(dot)(ut, w->q, l - 1) / (w->q_th + 1));
00097         ABIP(scale_array)(ut, w->h, l - 1, n);
00098         status_1 = ABIP(solve_lin_sys)(w->A, w->stgs, w->p, ut, u_prev, iter);
00099         ut[l - 1] += ABIP(dot)(ut, w->h, l - 1);
00100
00101         for (j = 0; j < m; ++j)
00102         {
00103             u[j] = ut[j] - v_prev[j];
00104         }
00105
00106         for (j = m; j < l; ++j)
00107         {
00108             u[j] = w->stgs->alpha * ut[j] + (1 - w->stgs->alpha) * u_prev[j] - v_prev[j];
00109         }
00110
00111         for (j = m; j < l; ++j)
00112         {
00113             tmp = u[j] / 2;
00114             u[j] = tmp + SQRTF(tmp * tmp + w->mu / beta_prev);
00115         }
00116
00117         for (j = m; j < l; ++j)
00118         {
00119             v[j] = v_prev[j] + (u[j] - w->stgs->alpha * ut[j] - (1 - w->stgs->alpha) * u_prev[j]);
00120         }
00121     }

```

```

00121     }
00122
00123
00124     memcpy(ut_next, u, 1 * sizeof(abip_float));
00125     ABIP(add_scaled_array)(ut_next, v, 1, 1.0);
00126     ABIP(scale_array)(ut_next, w->stgs->rho_y, m);
00127     ABIP(add_scaled_array)(ut_next, w->h, 1 - 1, -ut_next[1 - 1]);
00128     ABIP(add_scaled_array)(ut_next, w->h, 1 - 1, -ABIP(dot)(ut_next, w->g, 1 - 1) / (w->g_th +
1));
00129     ABIP(scale_array)(&(ut_next[m]), -1, n);
00130     status_2 = ABIP(solve_lin_sys)(w->A, w->stgs, w->p, ut_next, u, iter);
00131     ut_next[1 - 1] += ABIP(dot)(ut_next, w->h, 1 - 1);
00132
00133     for (j = 0; j < m; ++j)
00134     {
00135         u_next[j] = ut_next[j] - v[j];
00136     }
00137
00138     for (j = m; j < 1; ++j)
00139     {
00140         u_next[j] = w->stgs->alpha * ut_next[j] + (1 - w->stgs->alpha) * u[j] - v[j];
00141     }
00142
00143     for(j = m; j < 1; ++j)
00144     {
00145         tmp = u_next[j] / 2;
00146         u_next[j] = tmp + SQRTF(tmp * tmp + w->mu / beta_prev);
00147     }
00148
00149     for (j = m; j < 1; ++j)
00150     {
00151         v_next[j] = v[j] + (u_next[j] - w->stgs->alpha * ut_next[j] - (1 - w->stgs->alpha) *
u[j]);
00152     }
00153
00154     memcpy(delta_ut, v, 1 * sizeof(abip_float));
00155     ABIP(scale_array)(delta_ut, 2.0, 1);
00156     ABIP(add_scaled_array)(delta_ut, u_next, 1, 1.0);
00157     ABIP(add_scaled_array)(delta_ut, u, 1, -1.0);
00158     ABIP(add_scaled_array)(delta_ut, v_next, 1, -1.0);
00159     ABIP(add_scaled_array)(delta_ut, v_prev, 1, -1.0);
00160
00161     memcpy(delta_u, u, 1 * sizeof(abip_float));
00162     ABIP(add_scaled_array)(delta_u, u_next, 1, -1.0);
00163
00164     memcpy(delta_v, u_next, 1 * sizeof(abip_float));
00165     ABIP(add_scaled_array)(delta_v, u, 1, -1.0);
00166     ABIP(scale_array)(delta_v, w->stgs->alpha-1.0, 1);
00167     ABIP(add_scaled_array)(delta_v, v_next, 1, 1.0);
00168     ABIP(add_scaled_array)(delta_v, v, 1, -1.0);
00169
00170     utut = ABIP(dot)(delta_ut, delta_ut, 1);
00171     utv = ABIP(dot)(delta_ut, delta_v, 1);
00172     uu = ABIP(dot)(delta_u, delta_u, 1);
00173     vv = ABIP(dot)(delta_v, delta_v, 1);
00174     uv = ABIP(dot)(delta_u, delta_v, 1);
00175
00176     norm_ut = ABIP(norm)(delta_ut, 1);
00177     norm_u = ABIP(norm)(delta_u, 1);
00178     norm_v = ABIP(norm)(delta_v, 1);
00179
00180     alpha_SD = vv/utv;
00181     alpha_MG = utv/utut;
00182     gamma_SD = vv/uv;
00183     gamma_MG = uv/uu;
00184     /*
00185     abip_printf("alpha_SD = %3.6f, alpha_MG = %3.6f, gamma_SD = %3.6f, gamma_MG = %3.6f\n",
alpha_SD, alpha_MG, gamma_SD, gamma_MG);
00186     */
00187     if (2*alpha_MG > alpha_SD)
00188     {
00189         alpha_ss = alpha_MG;
00190     }
00191     else
00192     {
00193         alpha_ss = alpha_SD - 0.5*alpha_MG;
00194     }
00195
00196     if (2*gamma_MG > gamma_SD)
00197     {
00198         gamma_ss = gamma_MG;
00199     }
00200     else
00201     {
00202         gamma_ss = gamma_SD - 0.5*gamma_MG;
00203     }
00204

```

```

00205     alpha_cor = utv / (norm_v*norm_ut);
00206     gamma_cor = uv / (norm_v*norm_u);
00207
00208     if (alpha_cor > w->stgs->eps_cor && gamma_cor > w->stgs->eps_cor)
00209     {
00210         beta = SQRTE(alpha_ss*gamma_ss);
00211     }
00212     else if (alpha_cor > w->stgs->eps_cor && gamma_cor <= w->stgs->eps_cor)
00213     {
00214         beta = alpha_ss;
00215     }
00216     else if (alpha_cor <= w->stgs->eps_cor && gamma_cor > w->stgs->eps_cor)
00217     {
00218         beta = gamma_ss;
00219     }
00220     else
00221     {
00222         beta = beta_prev;
00223     }
00224
00225     if (ABS(beta-beta_prev) > 0 && ABS(beta-beta_prev) <= w->stgs->eps_pen)
00226     {
00227         beta = (beta+beta_prev)/2;
00228         break;
00229     }
00230     else if (ABS(beta-beta_prev) > w->stgs->eps_pen)
00231     {
00232         beta_prev = beta;
00233         memcpy(u_prev, u, 1 * sizeof(abip_float));
00234         for (j = 0; j < m; ++j)
00235         {
00236             v_prev[j] = v[j];
00237         }
00238         for (j = m; j < l; ++j)
00239         {
00240             v_prev[j] = (w->mu / beta_prev) / u_prev[j];
00241         }
00242     }
00243     else
00244     {
00245         memcpy(u_prev, u, 1 * sizeof(abip_float));
00246         memcpy(v_prev, v, 1 * sizeof(abip_float));
00247     }
00248     /*
00249     abip_printf("beta = %3.7e, vv = %3.7e, uu = %3.7e, utv = %3.7e, uv = %3.7e, utut = %3.7e\n",
00250     beta, vv, uu, utv, uv, utut);
00251     */
00252 }
00253 w->beta = beta;
00254
00255 RETURN MIN(status_1, status_2) ;
00256 }
00257
00258 ABIPAdaptWork *ABIP(init_adapt)
00259 {
00260     ABIPWork *w
00261 }
00262 {
00263     DEBUG_FUNC
00264
00265     ABIPAdaptWork *a = (ABIPAdaptWork *) abip_malloc(1, sizeof(ABIPAdaptWork));
00266
00267     if (!a)
00268     {
00269         RETURN ABIP_NULL;
00270     }
00271
00272     a->l = w->m + w->n + 1;
00273
00274     a->k = w->stgs->adaptive_lookback;
00275
00276     if (a->k <= 0)
00277     {
00278         RETURN a;
00279     }
00280
00281     a->u_prev = (abip_float *) abip_malloc(a->l, sizeof(abip_float));
00282     a->v_prev = (abip_float *) abip_malloc(a->l, sizeof(abip_float));
00283     a->ut = (abip_float *) abip_malloc(a->l, sizeof(abip_float));
00284     a->u = (abip_float *) abip_malloc(a->l, sizeof(abip_float));
00285     a->v = (abip_float *) abip_malloc(a->l, sizeof(abip_float));
00286     a->ut_next = (abip_float *) abip_malloc(a->l, sizeof(abip_float));
00287     a->u_next = (abip_float *) abip_malloc(a->l, sizeof(abip_float));
00288     a->v_next = (abip_float *) abip_malloc(a->l, sizeof(abip_float));
00289
00290     a->delta_ut = (abip_float *) abip_malloc(a->l, sizeof(abip_float));

```

```

00291     a->delta_u = (abip_float *) abip_malloc(a->l, sizeof(abip_float));
00292     a->delta_v = (abip_float *) abip_malloc(a->l, sizeof(abip_float));
00293
00294     a->total_adapt_time = 0.0;
00295
00296     if (!a->u_prev || !a->v_prev || !a->ut || !a->u || !a->v || !a->ut_next || !a->u_next ||
!a->v_next || !a->delta_ut || !a->delta_u || !a->delta_v)
00297     {
00298         ABIP(free_adapt)(a);
00299         a = ABIP_NULL;
00300     }
00301
00302     RETURN a;
00303 }
00304
00305 abip_int ABIP(adaptive)
00306 {
00307     ABIPWork *w,
00308     abip_int iter
00309 )
00310 {
00311     DEBUG_FUNC
00312
00313     abip_int k = w->adapt->k;
00314     abip_int info;
00315
00316     ABIP(timer) adapt_timer;
00317     if (k <= 0)
00318     {
00319         RETURN -1;
00320     }
00321
00322     ABIP(tic) (&adapt_timer);
00323
00324     info = update_adapt_params(w, iter);
00325
00326     if (iter == -1)
00327     {
00328         RETURN -1;
00329     }
00330
00331     w->adapt->total_adapt_time += ABIP(tocq) (&adapt_timer);
00332
00333     RETURN info;
00334 }
00335
00336 void ABIP(free_adapt)
00337 {
00338     ABIPAdaptWork *a
00339 )
00340 {
00341     DEBUG_FUNC
00342
00343     if (a)
00344     {
00345         if (a->u_prev)
00346         {
00347             abip_free(a->u_prev);
00348         }
00349
00350         if (a->v_prev)
00351         {
00352             abip_free(a->v_prev);
00353         }
00354
00355         if (a->ut)
00356         {
00357             abip_free(a->ut);
00358         }
00359
00360         if (a->u)
00361         {
00362             abip_free(a->u);
00363         }
00364
00365         if (a->v)
00366         {
00367             abip_free(a->v);
00368         }
00369
00370         if (a->ut_next)
00371         {
00372             abip_free(a->ut_next);
00373         }
00374
00375         if (a->u_next)
00376         {

```



```

00377         abip_free(a->u_next);
00378     }
00379
00380     if (a->v_next)
00381     {
00382         abip_free(a->v_next);
00383     }
00384
00385     if (a->delta_ut)
00386     {
00387         abip_free(a->delta_ut);
00388     }
00389
00390     if (a->delta_u)
00391     {
00392         abip_free(a->delta_u);
00393     }
00394
00395     if (a->delta_v)
00396     {
00397         abip_free(a->delta_v);
00398     }
00399
00400     abip_free(a);
00401 }
00402
00403 RETURN;
00404 }
00405
00406 char *ABIP(get_adapt_summary)
00407 (
00408     const ABIPInfo *info,
00409     ABIPAdaptWork *a
00410 )
00411 {
00412     DEBUG_FUNC
00413
00414     char *str = (char *) abip_malloc(sizeof(char) * 64);
00415     sprintf(str, "\tBarzilai-Borwein spectral method: avg step time: %1.2es\n", a->total_adapt_time /
00416         (info->admm_iter + 1) / 1e3);
00417
00418     a->total_adapt_time = 0.0;
00419     RETURN str;
00420 }

```

5.98 src/cs.c File Reference

```

#include "cs.h"
#include "abip.h"

```

Functions

- `cs *ABIP() cs_transpose` (const `cs *A`, `abip_int` values)
- `cs *ABIP() cs_compress` (const `cs *T`)
- `cs *ABIP() cs_spalloc` (`abip_int` m, `abip_int` n, `abip_int` nnzmax, `abip_int` values, `abip_int` triplet)
- `cs *ABIP() cs_spfree` (`cs *A`)
- `abip_float ABIP() cs_cumsum` (`abip_int` *p, `abip_int` *c, `abip_int` n)
- `abip_int *ABIP() cs_pinv` (`abip_int` const *p, `abip_int` n)
- `cs *ABIP() cs_symperm` (const `cs *A`, const `abip_int` *pinv, `abip_int` values)

5.98.1 Function Documentation

5.98.1.1 cs_compress()

```
cs *ABIP() cs_compress (
    const cs * T )
```

Definition at line 57 of file [cs.c](#).

5.98.1.2 cs_cumsum()

```
abip_float ABIP() cs_cumsum (
    abip_int * p,
    abip_int * c,
    abip_int n )
```

Definition at line 162 of file [cs.c](#).

5.98.1.3 cs_pinv()

```
abip_int *ABIP() cs_pinv (
    abip_int const * p,
    abip_int n )
```

Definition at line 190 of file [cs.c](#).

5.98.1.4 cs_spalloc()

```
cs *ABIP() cs_spalloc (
    abip_int m,
    abip_int n,
    abip_int nnzmax,
    abip_int values,
    abip_int triplet )
```

Definition at line 117 of file [cs.c](#).

5.98.1.5 cs_spfree()

```
cs *ABIP() cs_spfree (
    cs * A )
```

Definition at line 145 of file [cs.c](#).

5.98.1.6 cs_symperm()

```
cs *ABIP() cs_symperm (
    const cs * A,
    const abip_int * pinv,
    abip_int values )
```

Definition at line 218 of file cs.c.

5.98.1.7 cs_transpose()

```
cs *ABIP() cs_transpose (
    const cs * A,
    abip_int values )
```

Definition at line 33 of file cs.c.

5.99 cs.c

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

```
00001 #include "cs.h"
00002 #include "abip.h"
00003
00004 /* NB: this is a subset of the routines in the CSPARSE package by Tim Davis et. al., for the full
    package please visit
    * http://www.cise.ufl.edu/research/sparse/CSparse/ */
00005
00006
00007 /* wrapper for free */
00008 static void *cs_free
00009 (
00010     void *p
00011 )
00012 {
00013     if (p)
00014     {
00015         abip_free(p);
00016     }
00017     return (ABIP_NULL);
00018 }
00019
00020 static cs *cs_done
00021 (
00022     cs *C,
00023     void *w,
00024     void *x,
00025     abip_int ok
00026 )
00027 {
00028     cs_free(w);
00029     cs_free(x);
00030     return (ok ? C : ABIP(cs_spfree)(C));
00031 }
00032
00033 cs *ABIP(cs_transpose) (const cs *A, abip_int values)
00034 {
00035     abip_int p, q, j, *Cp, *Ci, n, m, *Ap, *Ai, *w ;
00036     abip_float *Cx, *Ax ;
00037     cs *C ;
00038     m = A->m ; n = A->n ; Ap = A->p ; Ai = A->i ; Ax = A->x ;
00039     C = ABIP(cs_spalloc)(n, m, Ap [n], values && Ax, 0) ; /* allocate result */
00040     w = abip_calloc(m, sizeof (abip_int)) ; /* get workspace */
00041     if (!C || !w) return (cs_done (C, w, NULL, 0)) ; /* out of memory */
00042     Cp = C->p ; Ci = C->i ; Cx = C->x ;
00043     for (p = 0 ; p < Ap [n] ; p++) w [Ai [p]]++ ; /* row counts */
00044     ABIP(cs_cumsum) (Cp, w, m) ; /* row pointers */
00045     for (j = 0 ; j < n ; j++)
00046     {
```

```

00047         for (p = Ap [j] ; p < Ap [j+1] ; p++)
00048         {
00049             Ci [q = w [Ai [p]]++] = j ; /* place A(i,j) as entry C(j,i) */
00050             if (Cx [q] = Ax [p];
00051         }
00052     }
00053     return (cs_done (C, w, NULL, 1)) ; /* success; free w and return C */
00054 }
00055
00056
00057 cs *ABIP(cs_compress)
00058 (
00059     const cs *T
00060 )
00061 {
00062     abip_int m;
00063     abip_int n;
00064     abip_int nnz;
00065     abip_int p;
00066     abip_int k;
00067
00068     abip_int *Cp;
00069     abip_int *Ci;
00070     abip_int *w;
00071     abip_int *Ti;
00072     abip_int *Tj;
00073
00074     abip_float *Cx;
00075     abip_float *Tx;
00076
00077     cs *C;
00078
00079     m = T->m;
00080     n = T->n;
00081     Ti = T->i;
00082     Tj = T->p;
00083     Tx = T->x;
00084     nnz = T->nnz;
00085
00086     C = ABIP(cs_spalloc)(m, n, nnz, Tx != ABIP_NULL, 0);
00087     w = (abip_int *) abip_calloc(n, sizeof(abip_int));
00088
00089     if (!C || !w)
00090     {
00091         return (cs_done(C, w, ABIP_NULL, 0));
00092     }
00093
00094     Cp = C->p;
00095     Ci = C->i;
00096     Cx = C->x;
00097
00098     for (k = 0; k < nnz; k++)
00099     {
00100         w[Tj[k]]++;
00101     }
00102
00103     ABIP(cs_cumsum)(Cp, w, n);
00104
00105     for (k = 0; k < nnz; k++)
00106     {
00107         Ci[p = w[Tj[k]]++] = Ti[k];
00108         if (Cx)
00109         {
00110             Cx[p] = Tx[k];
00111         }
00112     }
00113
00114     return (cs_done(C, w, ABIP_NULL, 1));
00115 }
00116
00117 cs *ABIP(cs_spalloc)
00118 (
00119     abip_int m,
00120     abip_int n,
00121     abip_int nnzmax,
00122     abip_int values,
00123     abip_int triplet
00124 )
00125 {
00126     cs *A = (cs *) abip_calloc(1, sizeof(cs));
00127     if (!A)
00128     {
00129         return (ABIP_NULL);
00130     }
00131
00132     A->m = m;
00133     A->n = n;

```

```

00134
00135     A->nnzmax = nnzmax = MAX(nnzmax, 1);
00136     A->nnz = triplet ? 0 : -1;
00137
00138     A->p = (abip_int *) abip_malloc((triplet ? nnzmax : n + 1) * sizeof(abip_int));
00139     A->i = (abip_int *) abip_malloc(nnzmax * sizeof(abip_int));
00140     A->x = values ? (abip_float *) abip_malloc(nnzmax * sizeof(abip_float)) : ABIP_NULL;
00141
00142     return ((!A->p || !A->i || (values && !A->x)) ? ABIP(cs_spfree)(A) : A);
00143 }
00144
00145 cs *ABIP(cs_spfree)
00146 (
00147     cs *A
00148 )
00149 {
00150     if (!A)
00151     {
00152         return (ABIP_NULL);
00153     }
00154
00155     cs_free(A->p);
00156     cs_free(A->i);
00157     cs_free(A->x);
00158
00159     return ((cs *)cs_free(A));
00160 }
00161
00162 abip_float ABIP(cs_cumsum)
00163 (
00164     abip_int *p,
00165     abip_int *c,
00166     abip_int n
00167 )
00168 {
00169     abip_int i;
00170     abip_int nnz = 0;
00171     abip_float nnz2 = 0;
00172
00173     if (!p || !c)
00174     {
00175         return (-1);
00176     }
00177
00178     for (i = 0; i < n; i++)
00179     {
00180         p[i] = nnz;
00181         nnz += c[i];
00182         nnz2 += c[i];
00183         c[i] = p[i];
00184     }
00185
00186     p[n] = nnz;
00187     return (nnz2);
00188 }
00189
00190 abip_int *ABIP(cs_pinv)
00191 (
00192     abip_int const *p,
00193     abip_int n
00194 )
00195 {
00196     abip_int k;
00197     abip_int *pinv;
00198
00199     if (!p)
00200     {
00201         return (ABIP_NULL);
00202     }
00203
00204     pinv = (abip_int *)abip_malloc(n * sizeof(abip_int));
00205     if (!pinv)
00206     {
00207         return (ABIP_NULL);
00208     }
00209
00210     for (k = 0; k < n; k++)
00211     {
00212         pinv[p[k]] = k;
00213     }
00214
00215     return (pinv);
00216 }
00217
00218 cs *ABIP(cs_symperm)
00219 (
00220     const cs *A,

```

```

00221     const abip_int *pinv,
00222     abip_int values
00223 )
00224 {
00225     abip_int i;
00226     abip_int j;
00227     abip_int p;
00228     abip_int q;
00229     abip_int i2;
00230     abip_int j2;
00231     abip_int n;
00232     abip_int *Ap;
00233     abip_int *Ai;
00234     abip_int *Cp;
00235     abip_int *Ci;
00236     abip_int *w;
00237
00238     abip_float *Cx;
00239     abip_float *Ax;
00240
00241     cs *C;
00242
00243     n = A->n;
00244     Ap = A->p;
00245     Ai = A->i;
00246     Ax = A->x;
00247
00248     C = ABIP(cs_spalloc)(n, n, Ap[n], values && (Ax != ABIP_NULL), 0);
00249     w = (abip_int *) abip_calloc(n, sizeof(abip_int));
00250
00251     if (!C || !w)
00252     {
00253         return (cs_done(C, w, ABIP_NULL, 0));
00254     }
00255
00256     Cp = C->p;
00257     Ci = C->i;
00258     Cx = C->x;
00259
00260     for (j = 0; j < n; j++)
00261     {
00262         j2 = pinv ? pinv[j] : j;
00263
00264         for (p = Ap[j]; p < Ap[j + 1]; p++)
00265         {
00266             i = Ai[p];
00267             if (i > j)
00268             {
00269                 continue;
00270             }
00271             i2 = pinv ? pinv[i] : i;
00272             w[MAX(i2, j2)]++;
00273         }
00274     }
00275
00276     ABIP(cs_cumsum)(Cp, w, n);
00277
00278     for (j = 0; j < n; j++)
00279     {
00280         j2 = pinv ? pinv[j] : j;
00281
00282         for (p = Ap[j]; p < Ap[j + 1]; p++)
00283         {
00284             i = Ai[p];
00285
00286             if (i > j)
00287             {
00288                 continue;
00289             }
00290
00291             i2 = pinv ? pinv[i] : i;
00292             Ci[q = w[MAX(i2, j2)]++] = MIN(i2, j2);
00293
00294             if (Cx)
00295             {
00296                 Cx[q] = Ax[p];
00297             }
00298         }
00299     }
00300
00301     return (cs_done(C, w, ABIP_NULL, 1));
00302 }

```

5.100 src/ctrlc.c File Reference

```
#include "ctrlc.h"
```

5.101 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 #ifdef MATLAB_MEX_FILE
00015
00016 static int istrate;
00017 void abip_start_interrupt_listener(void)
00018 {
00019     istrate = 0; //tSetInterruptEnabled(1);
00020 }
00021
00022 void abip_end_interrupt_listener(void)
00023 {
00024     utSetInterruptEnabled(istrate);
00025 }
00026
00027 int abip_is_interrupted(void)
00028 {
00029     return 0; // 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     int_detected = 1;
00042     return TRUE;
00043 }
00044
00045 void abip_start_interrupt_listener(void)
00046 {
00047     int_detected = 0;
00048     SetConsoleCtrlHandler(abip_handle_ctrlc, TRUE);
00049 }
00050
00051 void abip_end_interrupt_listener(void)
00052 {
00053     SetConsoleCtrlHandler(abip_handle_ctrlc, FALSE);
00054 }
00055
00056 int abip_is_interrupted(void)
00057 {
00058     return int_detected;
00059 }
00060
00061 #else /* Unix */
00062
00063 #include <signal.h>
00064 static int int_detected;
00065 struct sigaction oact;
00066 static void abip_handle_ctrlc(int dummy)
00067 {
00068     int_detected = dummy ? dummy : -1;
00069 }
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 */

```

5.102 src/linalg.c File Reference

```

#include "linalg.h"
#include <math.h>

```

Functions

- void **ABIP()** **set_as_scaled_array** (abip_float *x, const abip_float *a, const abip_float b, abip_int len)
*compute $x = b * a$*
- void **ABIP()** **set_as_sqrt** (abip_float *x, const abip_float *v, abip_int len)
compute $x = \sqrt{v}$
- void **ABIP()** **set_as_sq** (abip_float *x, const abip_float *v, abip_int len)
compute $x = v.^2$
- void **ABIP()** **scale_array** (abip_float *a, const abip_float b, abip_int len)
*compute $a *= b$*
- abip_float **ABIP()** **dot** (const abip_float *x, const abip_float *y, abip_int len)
*compute $x * y$*
- abip_float **ABIP()** **norm_sq** (const abip_float *v, abip_int len)
compute $\|v\|_2^2$
- abip_float **ABIP()** **norm** (const abip_float *v, abip_int len)
compute $\|v\|_2$
- abip_float **ABIP()** **min_abs_sqrt** (const abip_float *a, abip_int len, abip_float ref)
compute square root of the minimal absolute value
- abip_float **ABIP()** **norm_one** (const abip_float *v, abip_int len)
compute L1 norm
- abip_float **ABIP()** **norm_one_sqrt** (const abip_float *v, abip_int len)
compute square root L1 norm
- abip_float **ABIP()** **norm_inf** (const abip_float *a, abip_int len)
compute the infinity norm
- abip_float **ABIP()** **norm_inf_sqrt** (const abip_float *v, abip_int len)

- compute square root infinity norm*
- void `ABIP()` `add_array` (`abip_float` *a, const `abip_float` b, `abip_int` len)
- compute $a \mathrel{.+=} b$*
- void `ABIP()` `add_scaled_array` (`abip_float` *a, const `abip_float` *b, `abip_int` len, const `abip_float` sc)
- compute $a \mathrel{+=} sc*b$*
- `abip_float` `ABIP()` `norm_diff` (const `abip_float` *a, const `abip_float` *b, `abip_int` len)
- compute $\|a-b\|_2^2$*
- `abip_float` `ABIP()` `norm_inf_diff` (const `abip_float` *a, const `abip_float` *b, `abip_int` len)
- compute $\max(|a-b|)$*

5.102.1 Function Documentation

5.102.1.1 `add_array()`

```
void ABIP() add_array (
    abip_float * a,
    const abip_float b,
    abip_int len )
```

compute $a \mathrel{.+=} b$

Definition at line 218 of file `linalg.c`.

5.102.1.2 `add_scaled_array()`

```
void ABIP() add_scaled_array (
    abip_float * a,
    const abip_float * b,
    abip_int len,
    const abip_float sc )
```

compute $a \mathrel{+=} sc*b$

Definition at line 235 of file `linalg.c`.

5.102.1.3 `dot()`

```
abip_float ABIP() dot (
    const abip_float * x,
    const abip_float * y,
    abip_int len )
```

compute $x'y$

Definition at line 78 of file `linalg.c`.

5.102.1.4 min_abs_sqrt()

```
abip_float ABIP() min_abs_sqrt (
    const abip_float * a,
    abip_int len,
    abip_float ref )
```

compute square root of the minimal absolute value

Definition at line 126 of file [linalg.c](#).

5.102.1.5 norm()

```
abip_float ABIP() norm (
    const abip_float * v,
    abip_int len )
```

compute $\|v\|_2$

Definition at line 115 of file [linalg.c](#).

5.102.1.6 norm_diff()

```
abip_float ABIP() norm_diff (
    const abip_float * a,
    const abip_float * b,
    abip_int len )
```

compute $\|a-b\|_2^2$

Definition at line 253 of file [linalg.c](#).

5.102.1.7 norm_inf()

```
abip_float ABIP() norm_inf (
    const abip_float * a,
    abip_int len )
```

compute the infinity norm

Definition at line 182 of file [linalg.c](#).

5.102.1.8 norm_inf_diff()

```
abip_float ABIP() norm_inf_diff (
    const abip_float * a,
    const abip_float * b,
    abip_int len )
```

compute $\max(|a-b|)$

Definition at line 274 of file [linalg.c](#).

5.102.1.9 norm_inf_sqrt()

```
abip_float ABIP() norm_inf_sqrt (
    const abip_float * v,
    abip_int len )
```

compute square root infinity norm

Definition at line 205 of file [linalg.c](#).

5.102.1.10 norm_one()

```
abip_float ABIP() norm_one (
    const abip_float * v,
    abip_int len )
```

compute L1 norm

Definition at line 149 of file [linalg.c](#).

5.102.1.11 norm_one_sqrt()

```
abip_float ABIP() norm_one_sqrt (
    const abip_float * v,
    abip_int len )
```

compute square root L1 norm

Definition at line 167 of file [linalg.c](#).

5.102.1.12 norm_sq()

```
abip_float ABIP() norm_sq (
    const abip_float * v,
    abip_int len )
```

compute $\|v\|_2^2$

Definition at line 97 of file [linalg.c](#).

5.102.1.13 scale_array()

```
void ABIP() scale_array (
    abip_float * a,
    const abip_float b,
    abip_int len )
```

compute $a *= b$

Definition at line 61 of file [linalg.c](#).

5.102.1.14 set_as_scaled_array()

```
void ABIP() set_as_scaled_array (
    abip_float * x,
    const abip_float * a,
    const abip_float b,
    abip_int len )
```

compute $x = b*a$

Definition at line 9 of file [linalg.c](#).

5.102.1.15 set_as_sq()

```
void ABIP() set_as_sq (
    abip_float * x,
    const abip_float * v,
    abip_int len )
```

compute $x = v.^2$

Definition at line 44 of file [linalg.c](#).

5.102.1.16 set_as_sqrt()

```
void ABIP() set_as_sqrt (
    abip_float * x,
    const abip_float * v,
    abip_int len )
```

compute $x = \sqrt{v}$

Definition at line 27 of file [linalg.c](#).

5.103 linalg.c

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

```
00001 #include "linalg.h"
00002 #include <math.h>
00003
00004 // basic algebra operators
00005
00009 void ABIP(set_as_scaled_array)
00010 (
00011     abip_float *x,
00012     const abip_float *a,
00013     const abip_float b,
00014     abip_int len
00015 )
00016 {
00017     abip_int i;
00018     for (i = 0; i < len; ++i)
00019     {
00020         x[i] = b * a[i];
00021     }
00022 }
00023
00027 void ABIP(set_as_sqrt)
00028 (
00029     abip_float *x,
00030     const abip_float *v,
00031     abip_int len
00032 )
00033 {
00034     abip_int i;
00035     for (i = 0; i < len; ++i)
00036     {
00037         x[i] = SQRTF(v[i]);
00038     }
00039 }
00040
00044 void ABIP(set_as_sq)
00045 (
00046     abip_float *x,
00047     const abip_float *v,
00048     abip_int len
00049 )
00050 {
00051     abip_int i;
00052     for (i = 0; i < len; ++i)
00053     {
00054         x[i] = v[i]*v[i];
00055     }
00056 }
00057
00061 void ABIP(scale_array)
00062 (
00063     abip_float *a,
00064     const abip_float b,
00065     abip_int len
00066 )
00067 {
00068     abip_int i;
00069     for (i = 0; i < len; ++i)
00070     {
00071         a[i] *= b;
00072     }
00073 }
00074
```

```

00078 abip_float ABIP(dot)
00079 (
00080     const abip_float *x,
00081     const abip_float *y,
00082     abip_int len
00083 )
00084 {
00085     abip_int i;
00086     abip_float ip = 0.0;
00087     for (i = 0; i < len; ++i)
00088     {
00089         ip += x[i] * y[i];
00090     }
00091     return ip;
00092 }
00093
00097 abip_float ABIP(norm_sq)
00098 (
00099     const abip_float *v,
00100     abip_int len
00101 )
00102 {
00103     abip_int i;
00104     abip_float nmsq = 0.0;
00105     for (i = 0; i < len; ++i)
00106     {
00107         nmsq += v[i] * v[i];
00108     }
00109     return nmsq;
00110 }
00111
00115 abip_float ABIP(norm)
00116 (
00117     const abip_float *v,
00118     abip_int len
00119 )
00120 {
00121     return SQRTF(ABIP(norm_sq)(v, len));
00122 }
00126 abip_float ABIP(min_abs_sqrt)
00127 (
00128     const abip_float *a,
00129     abip_int len,
00130     abip_float ref
00131 )
00132 {
00133     abip_int i;
00134     abip_float tmp;
00135     for (i = 0; i < len; ++i)
00136     {
00137         tmp = ABS(a[i]);
00138         if (tmp <= ref && tmp > 0)
00139         {
00140             ref = tmp;
00141         }
00142     }
00143     return SQRTF(ref);
00144 }
00145
00149 abip_float ABIP(norm_one)
00150 (
00151     const abip_float *v,
00152     abip_int len
00153 )
00154 {
00155     abip_int i;
00156     abip_float nmone = 0.0;
00157     for (i = 0; i < len; ++i)
00158     {
00159         nmone += ABS(v[i]);
00160     }
00161     return nmone;
00162 }
00163
00167 abip_float ABIP(norm_one_sqrt)
00168 (
00169     const abip_float *v,
00170     abip_int len
00171 )
00172 {
00173     return SQRTF(ABIP(norm_one)(v, len));
00174 }
00175
00176
00177
00178
00182 abip_float ABIP(norm_inf)

```

```

00183 (
00184     const abip_float *a,
00185     abip_int len
00186 )
00187 {
00188     abip_int i;
00189     abip_float tmp;
00190     abip_float max = 0.0;
00191     for (i = 0; i < len; ++i)
00192     {
00193         tmp = ABS(a[i]);
00194         if (tmp >= max)
00195         {
00196             max = tmp;
00197         }
00198     }
00199     return max;
00200 }
00201
00205 abip_float ABIP(norm_inf_sqrt)
00206 (
00207     const abip_float *v,
00208     abip_int len
00209 )
00210 {
00211     return SQRTF(ABIP(norm_inf)(v, len));
00212 }
00213 // -----
00214
00218 void ABIP(add_array)
00219 (
00220     abip_float *a,
00221     const abip_float b,
00222     abip_int len
00223 )
00224 {
00225     abip_int i;
00226     for (i = 0; i < len; ++i)
00227     {
00228         a[i] += b;
00229     }
00230 }
00231
00235 void ABIP(add_scaled_array)
00236 (
00237     abip_float *a,
00238     const abip_float *b,
00239     abip_int len,
00240     const abip_float sc
00241 )
00242 {
00243     abip_int i;
00244     for (i = 0; i < len; ++i)
00245     {
00246         a[i] += sc * b[i];
00247     }
00248 }
00249
00253 abip_float ABIP(norm_diff)
00254 (
00255     const abip_float *a,
00256     const abip_float *b,
00257     abip_int len
00258 )
00259 {
00260     abip_int i;
00261     abip_float tmp;
00262     abip_float nm_diff = 0.0;
00263     for (i = 0; i < len; ++i)
00264     {
00265         tmp = (a[i] - b[i]);
00266         nm_diff += tmp * tmp;
00267     }
00268     return SQRTF(nm_diff);
00269 }
00270
00274 abip_float ABIP(norm_inf_diff)
00275 (
00276     const abip_float *a,
00277     const abip_float *b,
00278     abip_int len
00279 )
00280 {
00281     abip_int i;
00282     abip_float tmp;
00283     abip_float max = 0.0;
00284     for (i = 0; i < len; ++i)

```

```

00285     {
00286         tmp = ABS(a[i] - b[i]);
00287         if (tmp > max)
00288         {
00289             max = tmp;
00290         }
00291     }
00292     return max;
00293 }

```

5.104 src/normalize.c File Reference

```

#include "abip.h"
#include "normalize.h"
#include "linalg.h"

```

Macros

- `#define MIN_SCALE (1e-3)`
- `#define MAX_SCALE (1e3)`

Functions

- void `ABIP()` `normalize_b_c` (`ABIPWork *w`)
normalize b and c
- void `ABIP()` `calc_scaled_resids` (`ABIPWork *w`, `ABIPResiduals *r`)
calculate the scaled residuals
- void `ABIP()` `normalize_warm_start` (`ABIPWork *w`)
normalize the warm start solution
- void `ABIP()` `un_normalize_sol` (`ABIPWork *w`, `ABIPSolution *sol`)
recover the optimal solution

5.104.1 Macro Definition Documentation

5.104.1.1 MAX_SCALE

```
#define MAX_SCALE (1e3)
```

Definition at line 6 of file [normalize.c](#).

5.104.1.2 MIN_SCALE

```
#define MIN_SCALE (1e-3)
```

Definition at line 5 of file [normalize.c](#).

5.104.2 Function Documentation

5.104.2.1 `calc_scaled_resids()`

```
void ABIP() calc_scaled_resids (
    ABIPWork * w,
    ABIPResiduals * r )
```

calculate the scaled residuals

Definition at line 44 of file [normalize.c](#).

5.104.2.2 `normalize_b_c()`

```
void ABIP() normalize_b_c (
    ABIPWork * w )
```

normalize b and c

Definition at line 11 of file [normalize.c](#).

5.104.2.3 `normalize_warm_start()`

```
void ABIP() normalize_warm_start (
    ABIPWork * w )
```

normalize the warm start solution

Definition at line 100 of file [normalize.c](#).

5.104.2.4 `un_normalize_sol()`

```
void ABIP() un_normalize_sol (
    ABIPWork * w,
    ABIPSolution * sol )
```

recover the optimal solution

Definition at line 133 of file [normalize.c](#).

5.105 normalize.c

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

```

00001 #include "abip.h"
00002 #include "normalize.h"
00003 #include "linalg.h"
00004
00005 #define MIN_SCALE (1e-3)
00006 #define MAX_SCALE (1e3)
00007
00011 void ABIP(normalize_b_c)
00012 (
00013     ABIPWork *w
00014 )
00015 {
00016     abip_int i;
00017
00018     abip_float nm;
00019     abip_float *D = w->scal->D;
00020     abip_float *E = w->scal->E;
00021     abip_float *b = w->b;
00022     abip_float *c = w->c;
00023
00024     for (i = 0; i < w->n; ++i)
00025     {
00026         c[i] /= E[i];
00027     }
00028     nm = ABIP(norm)(c, w->n);
00029     w->sc_c = w->scal->mean_norm_row_A / MAX(nm, MIN_SCALE);
00030
00031     for (i = 0; i < w->m; ++i)
00032     {
00033         b[i] /= D[i];
00034     }
00035     nm = ABIP(norm)(b, w->m);
00036     w->sc_b = w->scal->mean_norm_col_A / MAX(nm, MIN_SCALE);
00037
00038     ABIP(scale_array)(c, w->sc_c * w->stgs->scale, w->n);
00039     ABIP(scale_array)(b, w->sc_b * w->stgs->scale, w->m);
00040 }
00044 void ABIP(calc_scaled_resids)
00045 (
00046     ABIPWork *w,
00047     ABIPResiduals *r
00048 )
00049 {
00050     abip_float *D = w->scal->D;
00051     abip_float *E = w->scal->E;
00052
00053     abip_float *u = w->u;
00054     abip_float *u_t = w->u_t;
00055     abip_float *u_prev = w->u_prev;
00056     abip_float tmp;
00057
00058     abip_int i;
00059     abip_int n = w->n;
00060     abip_int m = w->m;
00061
00062     r->res_pri = 0;
00063     for (i = 0; i < m; ++i)
00064     {
00065         tmp = (u[i] - u_t[i]) / (D[i] * w->sc_c);
00066         r->res_pri += tmp * tmp;
00067     }
00068
00069     for (i = 0; i < n; ++i)
00070     {
00071         tmp = (u[i + m] - u_t[i + m]) / (E[i] * w->sc_b);
00072         r->res_pri += tmp * tmp;
00073     }
00074
00075     tmp = u[n + m] - u_t[n + m];
00076     r->res_pri += tmp * tmp;
00077     r->res_pri = sqrt(r->res_pri);
00078
00079     r->res_dual = 0;
00080     for (i = 0; i < m; ++i)
00081     {
00082         tmp = (u[i] - u_prev[i]) * D[i] / w->sc_c;
00083         r->res_dual += tmp * tmp;
00084     }
00085
00086     for (i = 0; i < n; ++i)
00087     {
00088         tmp = (u[i + m] - u_prev[i + m]) * E[i] / w->sc_b;

```

```

00089         r->res_dual += tmp * tmp;
00090     }
00091
00092     tmp = u[n + m] - u_prev[n + m];
00093     r->res_dual += tmp * tmp;
00094     r->res_dual = sqrt(r->res_dual);
00095 }
00096
00100 void ABIP(normalize_warm_start)
00101 {
00102     ABIPWork *w
00103 }
00104 {
00105     abip_int i;
00106
00107     abip_float *D = w->scal->D;
00108     abip_float *E = w->scal->E;
00109
00110     abip_float *y = w->u;
00111     abip_float *x = &(w->u[w->m]);
00112     abip_float *s = &(w->v[w->m]);
00113
00114     for (i = 0; i < w->n; ++i)
00115     {
00116         x[i] *= (E[i] * w->sc_b);
00117     }
00118
00119     for (i = 0; i < w->m; ++i)
00120     {
00121         y[i] *= (D[i] * w->sc_c);
00122     }
00123
00124     for (i = 0; i < w->n; ++i)
00125     {
00126         s[i] /= (E[i] / (w->sc_c * w->stgs->scale));
00127     }
00128 }
00129
00133 void ABIP(un_normalize_sol)
00134 {
00135     ABIPWork *w,
00136     ABIPSolution *sol
00137 }
00138 {
00139     abip_int i;
00140
00141     abip_float *D = w->scal->D;
00142     abip_float *E = w->scal->E;
00143
00144     for (i = 0; i < w->n; ++i)
00145     {
00146         sol->x[i] /= (E[i] * w->sc_b);
00147     }
00148
00149     for (i = 0; i < w->m; ++i)
00150     {
00151         sol->y[i] /= (D[i] * w->sc_c);
00152     }
00153
00154     for (i = 0; i < w->n; ++i)
00155     {
00156         sol->s[i] *= E[i] / (w->sc_c * w->stgs->scale);
00157     }
00158 }

```

5.106 src/util.c File Reference

```

#include "glbopts.h"
#include "util.h"
#include "linsys.h"

```

Functions

- void [ABIP\(\)](#) [tic](#) ([ABIP](#)(timer) *t)

- `abip_float ABIP() tocq (ABIP(timer) *t)`
- `abip_float ABIP() toc (ABIP(timer) *t)`
define toc function
- `abip_float ABIP() str_toc (char *str, ABIP(timer) *t)`
store time consumed
- `void ABIP() print_work (const ABIPWork *w)`
print the iterates
- `void ABIP() print_data (const ABIPData *d)`
print some parameters
- `void ABIP() print_array (const abip_float *arr, abip_int n, const char *name)`
print array
- `void ABIP() free_data (ABIPData *d)`
set the memory of problem data free
- `void ABIP() free_sol (ABIPSolution *sol)`
set the memory of solution free
- `void ABIP() set_default_settings (ABIPData *d)`
set default setting

5.106.1 Function Documentation

5.106.1.1 free_data()

```
void ABIP() free_data (
    ABIPData * d )
```

set the memory of problem data free

Definition at line 226 of file [util.c](#).

5.106.1.2 free_sol()

```
void ABIP() free_sol (
    ABIPSolution * sol )
```

set the memory of solution free

Definition at line 259 of file [util.c](#).

5.106.1.3 print_array()

```
void ABIP() print_array (
    const abip_float * arr,
    abip_int n,
    const char * name )
```

print array

Definition at line 191 of file [util.c](#).

5.106.1.4 print_data()

```
void ABIP() print_data (
    const ABIPData * d )
```

print some parameters

Definition at line 162 of file [util.c](#).

5.106.1.5 print_work()

```
void ABIP() print_work (
    const ABIPWork * w )
```

print the iterates

Definition at line 133 of file [util.c](#).

5.106.1.6 set_default_settings()

```
void ABIP() set_default_settings (
    ABIPData * d )
```

set default setting

Definition at line 288 of file [util.c](#).

5.106.1.7 str_toc()

```
abip_float ABIP() str_toc (
    char * str,
    ABIP(timer) * t )
```

store time consumed

Definition at line 120 of file [util.c](#).

5.106.1.8 tic()

```
void ABIP() tic (
    ABIP(timer) * t )
```

Definition at line 73 of file [util.c](#).

5.106.1.9 toc()

```
abip_float ABIP() toc (
    ABIP(timer) * t )
```

define toc function

Definition at line 108 of file [util.c](#).

5.106.1.10 tocq()

```
abip_float ABIP() tocq (
    ABIP(timer) * t )
```

Definition at line 81 of file [util.c](#).

5.107 util.c

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

```

00001 #include "glbopts.h"
00002 #include "util.h"
00003 #include "linsys.h"
00004
00005 #if (defined NOTIMER)
00006
00007 void ABIP(tic)
00008 {
00009     ABIP(timer) *t
00010 } {}
00011
00012 abip_float ABIP(tocq)
00013 {
00014     ABIP(timer) *t
00015 }
00016 {
00017     return NAN;
00018 }
00019
00020 #elif (defined _WIN32 || _WIN64 || defined _WINDLL)
00021
00022 void ABIP(tic)
00023 {
00024     ABIP(timer) *t
00025 }
00026 {
00027     QueryPerformanceFrequency(&t->freq);
00028     QueryPerformanceCounter(&t->t看);
00029 }
00030
00031 abip_float ABIP(tocq)
00032 {
00033     ABIP(timer) *t
00034 }
00035 {
00036     QueryPerformanceCounter(&t->toc);
00037     return (1e3 * (t->toc.QuadPart - t->t看.QuadPart) / (abip_float)t->freq.QuadPart);
00038 }
00039
00040 #elif (defined __APPLE__)
00041
00042 void ABIP(tic)
00043 {
00044     ABIP(timer) *t
00045 }
00046 {
00047     /* read current clock cycles */
00048     t->t看 = mach_absolute_time();
00049 }
00050
00051 abip_float ABIP(tocq)
00052 {
00053     ABIP(timer) *t
00054 }
00055 {
00056     uint64_t duration;
00057
00058     t->toc = mach_absolute_time();
00059     duration = t->toc - t->t看;
00060
00061     mach_timebase_info(&(t->tinfo));
00062     duration *= t->tinfo.numer;
00063     duration /= t->tinfo.denom;
00064
00065     return (abip_float)duration / 1e6;
00066 }
00067
00068 #else
00069
00070 void ABIP(tic)
00071 {
00072     ABIP(timer) *t
00073 }
00074 {
00075     clock_gettime(CLOCK_MONOTONIC, &t->t看);
00076 }
00077
00078 abip_float ABIP(tocq)
00079 {
00080     ABIP(timer) *t
00081 }
00082 {
00083     clock_gettime(CLOCK_MONOTONIC, &t->toc);
00084     return (abip_float)(t->toc - t->t看) / 1e6;
00085 }

```

```

00086     struct timespec temp;
00087
00088     clock_gettime(CLOCK_MONOTONIC, &t->toc);
00089
00090     if ((t->toc.tv_nsec - t->tic.tv_nsec) < 0)
00091     {
00092         temp.tv_sec = t->toc.tv_sec - t->tic.tv_sec - 1;
00093         temp.tv_nsec = 1e9 + t->toc.tv_nsec - t->tic.tv_nsec;
00094     }
00095     else
00096     {
00097         temp.tv_sec = t->toc.tv_sec - t->tic.tv_sec;
00098         temp.tv_nsec = t->toc.tv_nsec - t->tic.tv_nsec;
00099     }
00100
00101     return (abip_float) temp.tv_sec * 1e3 + (abip_float) temp.tv_nsec / 1e6;
00102 }
00103
00104 #endif
00108 abip_float ABIP(toc)
00109 {
00110     ABIP(timer) *t
00111 }
00112 {
00113     abip_float time = ABIP(tocq)(t);
00114     abip_printf("time: %8.4f milli-seconds.\n", time);
00115     return time;
00116 }
00120 abip_float ABIP(str_toc)
00121 {
00122     char *str,
00123     ABIP(timer) *t
00124 }
00125 {
00126     abip_float time = ABIP(tocq)(t);
00127     abip_printf("%s - time: %8.4f milli-seconds.\n", str, time);
00128     return time;
00129 }
00133 void ABIP(print_work)
00134 {
00135     const ABIPWork *w
00136 }
00137 {
00138     abip_int i;
00139     abip_int l = w->n + w->m;
00140
00141     abip_printf("\n u_t is \n");
00142     for (i = 0; i < l; i++)
00143     {
00144         abip_printf("%f\n", w->u_t[i]);
00145     }
00146
00147     abip_printf("\n u is \n");
00148     for (i = 0; i < l; i++)
00149     {
00150         abip_printf("%f\n", w->u[i]);
00151     }
00152
00153     abip_printf("\n v is \n");
00154     for (i = 0; i < l; i++)
00155     {
00156         abip_printf("%f\n", w->v[i]);
00157     }
00158 }
00162 void ABIP(print_data)
00163 {
00164     const ABIPData *d
00165 }
00166 {
00167     abip_printf("m = %i\n", (int)d->m);
00168     abip_printf("n = %i\n", (int)d->n);
00169
00170     abip_printf("max_ipm_iters = %i\n", (int)d->stgs->max_ipm_iters);
00171     abip_printf("max_admm_iters = %i\n", (int)d->stgs->max_admm_iters);
00172
00173     abip_printf("verbose = %i\n", (int)d->stgs->verbose);
00174     abip_printf("normalize = %i\n", (int)d->stgs->normalize);
00175     abip_printf("warm_start = %i\n", (int)d->stgs->warm_start);
00176     abip_printf("adaptive = %i\n", (int)d->stgs->adaptive);
00177     abip_printf("adaptive_lookback = %i\n", (int)d->stgs->adaptive_lookback);
00178
00179     abip_printf("eps = %4f\n", d->stgs->eps);
00180     abip_printf("alpha = %4f\n", d->stgs->alpha);
00181     abip_printf("rho_y = %4f\n", d->stgs->rho_y);
00182     abip_printf("scale = %4f\n", d->stgs->scale);
00183
00184     abip_printf("eps_cor = %4f\n", d->stgs->eps_cor);

```



```

00185     abip_printf("eps_pen = %4f\n", d->stgs->eps_pen);
00186
00187 }
00191 void ABIP(print_array)
00192 (
00193     const abip_float *arr,
00194     abip_int n,
00195     const char *name
00196 )
00197 {
00198     abip_int i;
00199     abip_int j;
00200     abip_int k = 0;
00201
00202     abip_int num_on_one_line = 10;
00203
00204     abip_printf("\n");
00205     for (i = 0; i < n / num_on_one_line; ++i)
00206     {
00207         for (j = 0; j < num_on_one_line; ++j)
00208         {
00209             abip_printf("%s[%li] = %4f, ", name, (long)k, arr[k]);
00210             k++;
00211         }
00212         abip_printf("\n");
00213     }
00214
00215     for (j = k; j < n; ++j)
00216     {
00217         abip_printf("%s[%li] = %4f, ", name, (long)j, arr[j]);
00218     }
00219
00220     abip_printf("\n");
00221 }
00222
00226 void ABIP(free_data)
00227 (
00228     ABIPData *d
00229 )
00230 {
00231     if (d)
00232     {
00233         if (d->b)
00234         {
00235             abip_free(d->b);
00236         }
00237
00238         if (d->c)
00239         {
00240             abip_free(d->c);
00241         }
00242
00243         if (d->stgs)
00244         {
00245             abip_free(d->stgs);
00246         }
00247
00248         if (d->A)
00249         {
00250             ABIP(free_A_matrix) (d->A);
00251         }
00252
00253         abip_free(d);
00254     }
00255 }
00259 void ABIP(free_sol)
00260 (
00261     ABIPSolution *sol
00262 )
00263 {
00264     if (sol)
00265     {
00266         if (sol->x)
00267         {
00268             abip_free(sol->x);
00269         }
00270
00271         if (sol->y)
00272         {
00273             abip_free(sol->y);
00274         }
00275
00276         if (sol->s)
00277         {
00278             abip_free(sol->s);
00279         }
00280

```

```
00281         abip_free(sol);
00282     }
00283 }
00284
00288 void ABIP(set_default_settings)
00289 (
00290     ABIPData *d
00291 )
00292 {
00293     d->stgs->max_ipm_iters = MAX_IPM_ITERS;
00294     d->stgs->max_admm_iters = MAX_ADMM_ITERS;
00295     d->stgs->eps = EPS;
00296     d->stgs->alpha = ALPHA;
00297     d->stgs->cg_rate = CG_RATE;
00298
00299     d->stgs->normalize = NORMALIZE;
00300     d->stgs->scale = SCALE;
00301     d->stgs->rho_y = RHO_Y;
00302     d->stgs->sparsity_ratio = SPARSITY_RATIO;
00303
00304     d->stgs->adaptive = ADAPTIVE;
00305     d->stgs->eps_cor = EPS_COR;
00306     d->stgs->eps_pen = EPS_PEN;
00307     d->stgs->adaptive_lookback = ADAPTIVE_LOOKBACK;
00308
00309     d->stgs->dynamic_x = 0.8;
00310     d->stgs->dynamic_eta = 1.1;
00311
00312     d->stgs->restart_fre = 1000;
00313     d->stgs->restart_thresh = 100000;
00314
00315     d->stgs->origin_rescale = 0;
00316     d->stgs->pc_ruiz_rescale = 1;
00317     d->stgs->qp_rescale = 0;
00318     d->stgs->ruiz_iter = 10;
00319     d->stgs->hybrid_mu = 1; // whether use hybrid mu strategy
00320     d->stgs->dynamic_sigma = -1.0;
00321     d->stgs->hybrid_thresh = 1000; // the threshold to switch mu strategy
00322     d->stgs->dynamic_sigma_second = 0.5;
00323
00324     d->stgs->half_update = 0; // whether use half update
00325     d->stgs->avg_criterion = 0;
00326
00327     d->stgs->verbose = VERBOSE;
00328     d->stgs->warm_start = WARM_START;
00329 }
```

Index

- [_abip_calloc](#)
[glbopts.h](#), 184
 - [_abip_free](#)
[glbopts.h](#), 184
 - [_abip_malloc](#)
[glbopts.h](#), 184
 - [_abip_realloc](#)
[glbopts.h](#), 184
 - [_accum_by_A](#)
[common.c](#), 225
[common.h](#), 236
 - [_accum_by_Atrans](#)
[common.c](#), 225
[common.h](#), 236
 - [_ldl_factor](#)
[direct.c](#), 238
 - [_ldl_init](#)
[direct.c](#), 238
 - [_ldl_solve](#)
[direct.c](#), 239
 - [_normalize_A](#)
[common.c](#), 225
[common.h](#), 236
 - [_un_normalize_A](#)
[common.c](#), 226
[common.h](#), 236
- A
 - [abip_direct.m](#), 212
 - [abip_indirect.m](#), 214
 - [ABIP_PROBLEM_DATA](#), 18
 - [ABIP_WORK](#), 31
- ABIP
 - [abip.c](#), 270
 - [cs.h](#), 178
 - [glbopts.h](#), 184
 - [util.h](#), 208
- [abip.c](#)
 - [ABIP](#), 270
 - [finish](#), 270
 - [init](#), 271
 - [main](#), 271
 - [solve](#), 271
- [abip.h](#)
 - [ABIPAdaptWork](#), 169
 - [ABIPData](#), 170
 - [ABIPInfo](#), 170
 - [ABIPLinSysWork](#), 170
 - [ABIPMatrix](#), 170
 - [ABIPResiduals](#), 170
 - [ABIPScaling](#), 170
 - [ABIPSettings](#), 171
 - [ABIPSolution](#), 171
 - [ABIPWork](#), 171
 - [finish](#), 171
 - [init](#), 171
 - [main](#), 172
 - [solve](#), 172
 - [version](#), 172
- [ABIP_A_DATA_MATRIX](#), 7
 - [i](#), 7
 - [m](#), 7
 - [n](#), 8
 - [p](#), 8
 - [x](#), 8
- [ABIP_ADAPTIVE_WORK](#), 8
 - [delta_u](#), 9
 - [delta_ut](#), 9
 - [delta_v](#), 9
 - [k](#), 9
 - [l](#), 9
 - [total_adapt_time](#), 9
 - [u](#), 10
 - [u_next](#), 10
 - [u_prev](#), 10
 - [ut](#), 10
 - [ut_next](#), 10
 - [v](#), 10
 - [v_next](#), 11
 - [v_prev](#), 11
- [abip_amd](#)
 - [compile_direct.m](#), 42
- [abip_calloc](#)
 - [glbopts.h](#), 184
- [abip_common_linsys](#)
 - [make_abip.m](#), 258
- [abip_common_src](#)
 - [make_abip.m](#), 258
- [abip_direct.m](#)
 - [A](#), 212
 - [Ax](#), 212
 - [b](#), 212
 - [c](#), 212
 - [function](#), 213
 - [R](#), 212
 - [x](#), 213
- [abip_end_interrupt_listener](#)
 - [ctrlc.h](#), 181
- [ABIP_FAILED](#)

- glbopts.h, 185
- abip_float
 - glbopts.h, 192
- abip_free
 - glbopts.h, 185
- ABIP_INDETERMINATE
 - glbopts.h, 185
- abip_indirect.m
 - A, 214
 - Ax, 214
 - b, 214
 - c, 215
 - function, 215
 - R, 214
 - x, 215
- ABIP_INFEASIBLE
 - glbopts.h, 185
- ABIP_INFEASIBLE_INACCURATE
 - glbopts.h, 185
- ABIP_INFO, 11
 - admm_iter, 12
 - doj, 12
 - ipm_iter, 12
 - pobj, 12
 - rel_gap, 12
 - res_dual, 12
 - res_infeas, 13
 - res_pri, 13
 - res_unbdd, 13
 - setup_time, 13
 - solve_time, 13
 - status, 13
 - status_val, 14
- abip_int
 - glbopts.h, 192
- abip_is_interrupted
 - ctrlc.h, 181
- abip_ldl
 - compile_direct.m, 43
- ABIP_LIN_SYS_WORK, 14
 - At, 14
 - bp, 15
 - D, 15
 - Gp, 15
 - i, 15
 - j, 15
 - L, 15
 - M, 16
 - P, 16
 - p, 16
 - pardiso_work, 16
 - r, 16
 - tmp, 16
 - tot_cg_its, 17
 - total_solve_time, 17
 - z, 17
- abip_make_iso_compilers_happy
 - ctrlc.h, 182
- abip_malloc
 - glbopts.h, 186
- abip_mex.c
 - cast_to_abip_int_arr, 262
 - free_mex, 262
 - mexFunction, 263
 - parse_warm_start, 263
 - set_output_field, 263
- abip_mexfile
 - make_abip.m, 258
- ABIP_NULL
 - amd_global.c, 100
 - amd_internal.h, 106
 - glbopts.h, 186
- abip_pardiso.c
 - pardisoFactorize, 216
 - pardisoFree, 217
 - pardisoSolve, 217
- abip_pardiso.h
 - FACTORIZE, 219
 - pardiso, 221
 - PARDISO_BACKWARD, 219
 - PARDISO_FAC, 219
 - PARDISO_FORWARD, 219
 - PARDISO_FREE, 219
 - PARDISO_OK, 220
 - PARDISO_SOLVE, 220
 - PARDISO_SYM, 220
 - PARDISO_SYM_FAC, 220
 - pardisoFactorize, 221
 - pardisoFree, 221
 - PARDISOINDEX, 220
 - pardisoinit, 222
 - pardisoSolve, 222
 - PIVOTING, 220
 - SYMBOLIC, 221
- abip_printf
 - glbopts.h, 186
- ABIP_PROBLEM_DATA, 17
 - A, 18
 - b, 18
 - c, 18
 - m, 18
 - n, 18
 - sp, 18
 - stgs, 19
- abip_realloc
 - glbopts.h, 186
- ABIP_RESIDUALS, 19
 - bt_y_by_tau, 19
 - ct_x_by_tau, 20
 - kap, 20
 - last_admm_iter, 20
 - last_ipm_iter, 20
 - last_mu, 20
 - rel_gap, 20
 - res_dual, 21
 - res_infeas, 21

- res_pri, [21](#)
- res_unbdd, [21](#)
- tau, [21](#)
- ABIP_SCALING, [22](#)
 - D, [22](#)
 - E, [22](#)
 - mean_norm_col_A, [22](#)
 - mean_norm_row_A, [22](#)
- ABIP_SETTINGS, [23](#)
 - adaptive, [24](#)
 - adaptive_lookback, [24](#)
 - alpha, [24](#)
 - avg_criterion, [24](#)
 - cg_rate, [24](#)
 - dynamic_eta, [24](#)
 - dynamic_sigma, [25](#)
 - dynamic_sigma_second, [25](#)
 - dynamic_x, [25](#)
 - eps, [25](#)
 - eps_cor, [25](#)
 - eps_pen, [25](#)
 - half_update, [26](#)
 - hybrid_mu, [26](#)
 - hybrid_thresh, [26](#)
 - max_admm_iters, [26](#)
 - max_ipm_iters, [26](#)
 - max_time, [26](#)
 - normalize, [27](#)
 - origin_rescale, [27](#)
 - pc_ruiz_rescale, [27](#)
 - pfeasopt, [27](#)
 - qp_rescale, [27](#)
 - restart_fre, [27](#)
 - restart_thresh, [28](#)
 - rho_y, [28](#)
 - ruiz_iter, [28](#)
 - scale, [28](#)
 - sparsity_ratio, [28](#)
 - verbose, [28](#)
 - warm_start, [29](#)
- ABIP_SIGINT
 - glbopts.h, [186](#)
- ABIP_SOL_VARS, [29](#)
 - s, [29](#)
 - x, [29](#)
 - y, [30](#)
- ABIP_SOLVED
 - glbopts.h, [186](#)
- ABIP_SOLVED_INACCURATE
 - glbopts.h, [187](#)
- abip_start_interrupt_listener
 - ctrlc.h, [181](#)
- ABIP_UNBOUNDED
 - glbopts.h, [187](#)
- ABIP_UNBOUNDED_INACCURATE
 - glbopts.h, [187](#)
- ABIP_UNFINISHED
 - glbopts.h, [187](#)
- ABIP_VERSION
 - glbopts.h, [187](#)
- abip_version.c
 - version, [298](#)
- abip_version_mex.c
 - mexFunction, [269](#)
- ABIP_WORK, [30](#)
 - A, [31](#)
 - adapt, [31](#)
 - b, [31](#)
 - beta, [31](#)
 - c, [32](#)
 - double_check, [32](#)
 - dr, [32](#)
 - final_check, [32](#)
 - fre_old, [32](#)
 - g, [32](#)
 - g_th, [33](#)
 - gamma, [33](#)
 - h, [33](#)
 - m, [33](#)
 - mu, [33](#)
 - n, [33](#)
 - nm_b, [34](#)
 - nm_c, [34](#)
 - p, [34](#)
 - pr, [34](#)
 - sc_b, [34](#)
 - sc_c, [34](#)
 - scal, [35](#)
 - sigma, [35](#)
 - sp, [35](#)
 - stgs, [35](#)
 - u, [35](#)
 - u_avg, [35](#)
 - u_avgcon, [36](#)
 - u_prev, [36](#)
 - u_sumcon, [36](#)
 - u_t, [36](#)
 - v, [36](#)
 - v_avg, [36](#)
 - v_avgcon, [37](#)
 - v_prev, [37](#)
 - v_sumcon, [37](#)
- ABIPAdaptWork
 - abip.h, [169](#)
- ABIPData
 - abip.h, [170](#)
- ABIPInfo
 - abip.h, [170](#)
- ABIPLinSysWork
 - abip.h, [170](#)
- ABIPMatrix
 - abip.h, [170](#)
- ABIPResiduals
 - abip.h, [170](#)
- ABIPScaling
 - abip.h, [170](#)

- ABIPSettings
 - abip.h, 171
- ABIPSolution
 - abip.h, 171
- ABIPWork
 - abip.h, 171
- ABS
 - glbopts.h, 187
- accum_by_A
 - direct.c, 239
 - indirect.c, 247
 - linsys.h, 202
- accum_by_Atrans
 - direct.c, 239
 - indirect.c, 247
 - linsys.h, 202
- adapt
 - ABIP_WORK, 31
- ADAPTIVE
 - glbopts.h, 188
- adaptive
 - ABIP_SETTINGS, 24
 - adaptive.c, 299
 - adaptive.h, 176
- adaptive.c
 - adaptive, 299
 - free_adapt, 299
 - get_adapt_summary, 300
 - init_adapt, 300
- adaptive.h
 - adaptive, 176
 - free_adapt, 176
 - get_adapt_summary, 176
 - init_adapt, 177
- ADAPTIVE_LOOKBACK
 - glbopts.h, 188
- adaptive_lookback
 - ABIP_SETTINGS, 24
- add_array
 - linalg.c, 313
 - linalg.h, 195
- add_scaled_array
 - linalg.c, 313
 - linalg.h, 196
- addpath
 - make_abip.m, 256
- admm_iter
 - ABIP_INFO, 12
- ALPHA
 - glbopts.h, 188
- alpha
 - ABIP_SETTINGS, 24
- alternatively
 - compile_direct.m, 43
- amd.h
 - amd_2, 56
 - AMD_AGGRESSIVE, 50
 - amd_calloc, 58
 - AMD_CONTROL, 51
 - amd_control, 56
 - AMD_DATE, 51
 - AMD_DEFAULT_AGGRESSIVE, 51
 - AMD_DEFAULT_DENSE, 51
 - amd_defaults, 56
 - AMD_DENSE, 51
 - AMD_DMAX, 51
 - amd_free, 58
 - AMD_INFO, 52
 - amd_info, 56
 - AMD_INVALID, 52
 - amd_l2, 56
 - amd_l_control, 57
 - amd_l_defaults, 57
 - amd_l_info, 57
 - amd_l_order, 57
 - amd_l_valid, 57
 - AMD_LNZ, 52
 - AMD_MAIN_VERSION, 52
 - amd_malloc, 58
 - AMD_MEMORY, 52
 - AMD_N, 52
 - AMD_NCMPA, 53
 - AMD_NDENSE, 53
 - AMD_NDIV, 53
 - AMD_NMULTSUBS_LDL, 53
 - AMD_NMULTSUBS_LU, 53
 - AMD_NZ, 53
 - AMD_NZ_A_PLUS_AT, 54
 - AMD_NZDIAG, 54
 - AMD_OK, 54
 - AMD_OK_BUT_JUMBLED, 54
 - amd_order, 58
 - AMD_OUT_OF_MEMORY, 54
 - amd_printf, 59
 - amd_realloc, 59
 - AMD_STATUS, 54
 - AMD_SUB_VERSION, 55
 - AMD_SUBSUB_VERSION, 55
 - AMD_SYMMETRY, 55
 - amd_valid, 58
 - AMD_VERSION, 55
 - AMD_VERSION_CODE, 55
 - EXTERN, 55
- AMD_1
 - amd_1.c, 64
 - amd_internal.h, 106, 112
- amd_1.c
 - AMD_1, 64
- AMD_2
 - amd_2.c, 67
 - amd_internal.h, 106
- amd_2
 - amd.h, 56
- amd_2.c
 - AMD_2, 67
- AMD_aat

- amd_aat.c, 91
- amd_internal.h, 107, 113
- amd_aat.c
 - AMD_aat, 91
- AMD_AGGRESSIVE
 - amd.h, 50
- amd_malloc
 - amd.h, 58
 - amd_global.c, 101
- AMD_CONTROL
 - amd.h, 51
- AMD_control
 - amd_control.c, 94
 - amd_internal.h, 107
- amd_control
 - amd.h, 56
- amd_control.c
 - AMD_control, 94
- AMD_DATE
 - amd.h, 51
- AMD_debug
 - amd_dump.c, 97
 - amd_internal.h, 107
- AMD_DEBUG0
 - amd_internal.h, 107
- AMD_DEBUG1
 - amd_internal.h, 107
- AMD_DEBUG2
 - amd_internal.h, 107
- AMD_DEBUG3
 - amd_internal.h, 108
- AMD_DEBUG4
 - amd_internal.h, 108
- AMD_debug_init
 - amd_dump.c, 97
 - amd_internal.h, 108
- AMD_DEFAULT_AGGRESSIVE
 - amd.h, 51
- AMD_DEFAULT_DENSE
 - amd.h, 51
- AMD_defaults
 - amd_defaults.c, 96
 - amd_internal.h, 108
- amd_defaults
 - amd.h, 56
- amd_defaults.c
 - AMD_defaults, 96
- AMD_DENSE
 - amd.h, 51
- AMD_DMAX
 - amd.h, 51
- AMD_dump
 - amd_dump.c, 97
 - amd_internal.h, 108
- amd_dump.c
 - AMD_debug, 97
 - AMD_debug_init, 97
 - AMD_dump, 97
- amd_files
 - compile_direct.m, 43
- amd_free
 - amd.h, 58
 - amd_global.c, 101
- amd_global.c
 - ABIP_NULL, 100
 - amd_malloc, 101
 - amd_free, 101
 - amd_malloc, 101
 - amd_printf, 101
 - amd_realloc, 101
- AMD_INFO
 - amd.h, 52
- AMD_info
 - amd_info.c, 103
 - amd_internal.h, 108
- amd_info
 - amd.h, 56
- amd_info.c
 - AMD_info, 103
 - PRI, 103
- amd_internal.h
 - ABIP_NULL, 106
 - AMD_1, 106, 112
 - AMD_2, 106
 - AMD_aat, 107, 113
 - AMD_control, 107
 - AMD_debug, 107
 - AMD_DEBUG0, 107
 - AMD_DEBUG1, 107
 - AMD_DEBUG2, 107
 - AMD_DEBUG3, 108
 - AMD_DEBUG4, 108
 - AMD_debug_init, 108
 - AMD_defaults, 108
 - AMD_dump, 108
 - AMD_info, 108
 - AMD_order, 109
 - AMD_post_tree, 109, 113
 - AMD_postorder, 109, 113
 - AMD_preprocess, 109, 113
 - AMD_valid, 109
 - ASSERT, 109
 - EMPTY, 110
 - FALSE, 110
 - FLIP, 110
 - GLOBAL, 110
 - ID, 110
 - IMPLIES, 111
 - Int, 111
 - Int_MAX, 111
 - MAX, 111
 - MIN, 111
 - PRINTF, 111
 - PRIVATE, 112
 - SIZE_T_MAX, 112
 - TRUE, 112

- UNFLIP, 112
- AMD_INVALID
 - amd.h, 52
- amd_l2
 - amd.h, 56
- amd_l_control
 - amd.h, 57
- amd_l_defaults
 - amd.h, 57
- amd_l_info
 - amd.h, 57
- amd_l_order
 - amd.h, 57
- amd_l_valid
 - amd.h, 57
- AMD_LNZ
 - amd.h, 52
- AMD_MAIN_VERSION
 - amd.h, 52
- amd_malloc
 - amd.h, 58
 - amd_global.c, 101
- AMD_MEMORY
 - amd.h, 52
- AMD_N
 - amd.h, 52
- AMD_NCMPPA
 - amd.h, 53
- AMD_NDENSE
 - amd.h, 53
- AMD_NDIV
 - amd.h, 53
- AMD_NMULTSUBS_LDL
 - amd.h, 53
- AMD_NMULTSUBS_LU
 - amd.h, 53
- AMD_NZ
 - amd.h, 53
- AMD_NZ_A_PLUS_AT
 - amd.h, 54
- AMD_NZDIAG
 - amd.h, 54
- AMD_OK
 - amd.h, 54
- AMD_OK_BUT_JUMBLED
 - amd.h, 54
- AMD_order
 - amd_internal.h, 109
 - amd_order.c, 118
- amd_order
 - amd.h, 58
- amd_order.c
 - AMD_order, 118
- AMD_OUT_OF_MEMORY
 - amd.h, 54
- amd_path
 - compile_direct.m, 43
- AMD_post_tree
 - amd_internal.h, 109, 113
 - amd_post_tree.c, 121
- amd_post_tree.c
 - AMD_post_tree, 121
- AMD_postorder
 - amd_internal.h, 109, 113
 - amd_postorder.c, 123
- amd_postorder.c
 - AMD_postorder, 123
- AMD_preprocess
 - amd_internal.h, 109, 113
 - amd_preprocess.c, 127
- amd_preprocess.c
 - AMD_preprocess, 127
- amd_printf
 - amd.h, 59
 - amd_global.c, 101
- amd_realloc
 - amd.h, 59
 - amd_global.c, 101
- AMD_STATUS
 - amd.h, 54
- AMD_SUB_VERSION
 - amd.h, 55
- AMD_SUBSUB_VERSION
 - amd.h, 55
- AMD_SYMMETRY
 - amd.h, 55
- AMD_valid
 - amd_internal.h, 109
 - amd_valid.c, 129
- amd_valid
 - amd.h, 58
- amd_valid.c
 - AMD_valid, 129
- AMD_VERSION
 - amd.h, 55
- AMD_VERSION_CODE
 - amd.h, 55
- arr
 - make_abip.m, 259
- ASSERT
 - amd_internal.h, 109
- At
 - ABIP_LIN_SYS_WORK, 14
- avg_criterion
 - ABIP_SETTINGS, 24
- Ax
 - abip_direct.m, 212
 - abip_indirect.m, 214
- b
 - abip_direct.m, 212
 - abip_indirect.m, 214
 - ABIP_PROBLEM_DATA, 18
 - ABIP_WORK, 31
- beta
 - ABIP_WORK, 31
- BLASLIB

- make_abip.m, 259
- bp
 - ABIP_LIN_SYS_WORK, 15
- bt_y_by_tau
 - ABIP_RESIDUALS, 19
- c
 - abip_direct.m, 212
 - abip_indirect.m, 215
 - ABIP_PROBLEM_DATA, 18
 - ABIP_WORK, 32
 - make_abip.m, 259
- calc_scaled_resids
 - normalize.c, 321
 - normalize.h, 206
- calloc_func
 - SuiteSparse_config_struct, 38
- cast_to_abip_int_arr
 - abip_mex.c, 262
- CG_BEST_TOL
 - indirect.c, 247
- CG_MIN_TOL
 - indirect.c, 247
- CG_RATE
 - glbopts.h, 188
- cg_rate
 - ABIP_SETTINGS, 24
- cmd
 - compile_direct.m, 43
 - compile_indirect.m, 48
- common.c
 - _accum_by_A, 225
 - _accum_by_Atrans, 225
 - _normalize_A, 225
 - _un_normalize_A, 226
 - copy_A_matrix, 226
 - cumsum, 226
 - free_A_matrix, 226
 - MAX_SCALE, 224
 - MIN_SCALE, 225
 - validate_lin_sys, 227
- common.h
 - _accum_by_A, 236
 - _accum_by_Atrans, 236
 - _normalize_A, 236
 - _un_normalize_A, 236
 - cumsum, 237
- common_abip
 - make_abip.m, 259
- compile_direct
 - compile_direct.m, 42
 - make_abip.m, 257
- compile_direct.m, 41
 - abip_amd, 42
 - abip_ldl, 43
 - alternatively, 43
 - amd_files, 43
 - amd_path, 43
 - cmd, 43
 - compile_direct, 42
 - eval, 42
 - example, 44
 - exist, 42
 - following, 44
 - fprintf, 42
 - if, 42
 - intel64, 44
 - ldl_files, 44
 - ldl_path, 44
 - lib_path, 45
 - link, 45
 - linux, 45
 - mexext, 45
 - MKL, 45
 - mkl_macro, 46
 - oneapi, 46
 - pardiso_src, 46
 - platform, 46
 - self, 46
- compile_indirect
 - compile_indirect.m, 48
 - make_abip.m, 257
- compile_indirect.m, 48
 - cmd, 48
 - compile_indirect, 48
 - eval, 48
 - if, 48
- CONVERGED_INTERVAL
 - glbopts.h, 188
- copy_A_matrix
 - common.c, 226
 - linsys.h, 202
- cs
 - cs.h, 180
- cs.c
 - cs_compress, 305
 - cs_cumsum, 306
 - cs_pinv, 306
 - cs_spalloc, 306
 - cs_spfree, 306
 - cs_symperm, 306
 - cs_transpose, 307
- cs.h
 - ABIP, 178
 - cs, 180
 - cs_compress, 178
 - cs_cumsum, 178
 - cs_pinv, 178
 - cs_spalloc, 179
 - cs_spfree, 179
 - cs_symperm, 179
 - cs_transpose, 179
- cs_compress
 - cs.c, 305
 - cs.h, 178
- cs_cumsum
 - cs.c, 306

- cs.h, 178
- cs_pinv
 - cs.c, 306
 - cs.h, 178
- cs_spalloc
 - cs.c, 306
 - cs.h, 179
- cs_spfree
 - cs.c, 306
 - cs.h, 179
- cs_symperm
 - cs.c, 306
 - cs.h, 179
- cs_transpose
 - cs.c, 307
 - cs.h, 179
- ct_x_by_tau
 - ABIP_RESIDUALS, 20
- ctrlc.h
 - abip_end_interrupt_listener, 181
 - abip_is_interrupted, 181
 - abip_make_iso_compilers_happy, 182
 - abip_start_interrupt_listener, 181
- cumsum
 - common.c, 226
 - common.h, 237
- D
 - ABIP_LIN_SYS_WORK, 15
 - ABIP_SCALING, 22
- DEBUG_FUNC
 - glbopts.h, 188
- delete
 - make_abip.m, 257
- delta_u
 - ABIP_ADAPTIVE_WORK, 9
- delta_ut
 - ABIP_ADAPTIVE_WORK, 9
- delta_v
 - ABIP_ADAPTIVE_WORK, 9
- direct.c
 - _ldl_factor, 238
 - _ldl_init, 238
 - _ldl_solve, 239
 - accum_by_A, 239
 - accum_by_Atrans, 239
 - factorize, 239
 - form_kkt, 240
 - free_lin_sys_work, 240
 - get_lin_sys_method, 240
 - get_lin_sys_summary, 240
 - init_lin_sys_work, 240
 - normalize_A, 241
 - solve_lin_sys, 241
 - un_normalize_A, 241
- divcomplex_func
 - SuiteSparse_config_struct, 38
- dojb
 - ABIP_INFO, 12
- dot
 - linalg.c, 313
 - linalg.h, 196
- double_check
 - ABIP_WORK, 32
- dr
 - ABIP_WORK, 32
- dynamic_eta
 - ABIP_SETTINGS, 24
- dynamic_sigma
 - ABIP_SETTINGS, 25
- dynamic_sigma_second
 - ABIP_SETTINGS, 25
- dynamic_x
 - ABIP_SETTINGS, 25
- E
 - ABIP_SCALING, 22
- elseif
 - make_abip.m, 257
- EMPTY
 - amd_internal.h, 110
- EPS
 - glbopts.h, 189
- eps
 - ABIP_SETTINGS, 25
- EPS_COR
 - glbopts.h, 189
- eps_cor
 - ABIP_SETTINGS, 25
- EPS_PEN
 - glbopts.h, 189
- eps_pen
 - ABIP_SETTINGS, 25
- EPS_TOL
 - glbopts.h, 189
- eval
 - compile_direct.m, 42
 - compile_indirect.m, 48
- example
 - compile_direct.m, 44
- exist
 - compile_direct.m, 42
- EXTERN
 - amd.h, 55
- external/amd/amd.h, 49, 59
- external/amd/amd_1.c, 64, 65
- external/amd/amd_2.c, 67, 68
- external/amd/amd_aat.c, 91, 92
- external/amd/amd_control.c, 94, 95
- external/amd/amd_defaults.c, 95, 96
- external/amd/amd_dump.c, 96, 98
- external/amd/amd_global.c, 100, 102
- external/amd/amd_info.c, 103, 104
- external/amd/amd_internal.h, 105, 114
- external/amd/amd_order.c, 118
- external/amd/amd_post_tree.c, 121, 122
- external/amd/amd_postorder.c, 123, 124
- external/amd/amd_preprocess.c, 127

- external/amd/amd_valid.c, 129
- external/ldl/ldl.c, 131, 133
- external/ldl/ldl.h, 141, 149
- external/README.md, 150
- external/SuiteSparse_config.c, 150, 154
- external/SuiteSparse_config.h, 160, 166
- EXTRA_VERBOSE
 - make_abip.m, 259
- FACTORIZE
 - abip_pardiso.h, 219
- factorize
 - direct.c, 239
- FALSE
 - amd_internal.h, 110
- final_check
 - ABIP_WORK, 32
- finish
 - abip.c, 270
 - abip.h, 171
- FLIP
 - amd_internal.h, 110
- float
 - make_abip.m, 259
- following
 - compile_direct.m, 44
- form_kkt
 - direct.c, 240
- fprintf
 - compile_direct.m, 42
- fre_old
 - ABIP_WORK, 32
- free_A_matrix
 - common.c, 226
 - linsys.h, 202
- free_adapt
 - adaptive.c, 299
 - adaptive.h, 176
- free_data
 - util.c, 324
 - util.h, 208
- free_func
 - SuiteSparse_config_struct, 38
- free_lin_sys_work
 - direct.c, 240
 - indirect.c, 248
 - linsys.h, 202
- free_lin_sys_work_pds
 - linsys.h, 203
- free_mex
 - abip_mex.c, 262
- free_sol
 - util.c, 324
 - util.h, 208
- function
 - abip_direct.m, 213
 - abip_indirect.m, 215
- g
 - ABIP_WORK, 32
- g_th
 - ABIP_WORK, 33
- gamma
 - ABIP_WORK, 33
- get_adapt_summary
 - adaptive.c, 300
 - adaptive.h, 176
- get_lin_sys_method
 - direct.c, 240
 - indirect.c, 248
 - linsys.h, 203
- get_lin_sys_summary
 - direct.c, 240
 - indirect.c, 248
 - linsys.h, 203
- glbopts.h
 - _abip_calloc, 184
 - _abip_free, 184
 - _abip_malloc, 184
 - _abip_realloc, 184
 - ABIP, 184
 - abip_calloc, 184
 - ABIP_FAILED, 185
 - abip_float, 192
 - abip_free, 185
 - ABIP_INDETERMINATE, 185
 - ABIP_INFEASIBLE, 185
 - ABIP_INFEASIBLE_INACCURATE, 185
 - abip_int, 192
 - abip_malloc, 186
 - ABIP_NULL, 186
 - abip_printf, 186
 - abip_realloc, 186
 - ABIP_SIGINT, 186
 - ABIP_SOLVED, 186
 - ABIP_SOLVED_INACCURATE, 187
 - ABIP_UNBOUNDED, 187
 - ABIP_UNBOUNDED_INACCURATE, 187
 - ABIP_UNFINISHED, 187
 - ABIP_VERSION, 187
 - ABS, 187
 - ADAPTIVE, 188
 - ADAPTIVE_LOOKBACK, 188
 - ALPHA, 188
 - CG_RATE, 188
 - CONVERGED_INTERVAL, 188
 - DEBUG_FUNC, 188
 - EPS, 189
 - EPS_COR, 189
 - EPS_PEN, 189
 - EPS_TOL, 189
 - INDETERMINATE_TOL, 189
 - INFINITY, 189
 - MAX, 190
 - MAX_ADMM_ITERS, 190
 - MAX_IPM_ITERS, 190
 - MIN, 190

- NAN, 190
- NORMALIZE, 190
- POWF, 191
- RETURN, 191
- RHO_Y, 191
- SAFEDIV_POS, 191
- SCALE, 191
- SPARSITY_RATIO, 191
- SQRTF, 192
- VERBOSE, 192
- WARM_START, 192
- GLOBAL
 - amd_internal.h, 110
- Gp
 - ABIP_LIN_SYS_WORK, 15
- gpu
 - make_abip.m, 260
- h
 - ABIP_WORK, 33
- half_update
 - ABIP_SETTINGS, 26
- hybrid_mu
 - ABIP_SETTINGS, 26
- hybrid_thresh
 - ABIP_SETTINGS, 26
- hypot_func
 - SuiteSparse_config_struct, 38
- i
 - ABIP_A_DATA_MATRIX, 7
 - ABIP_LIN_SYS_WORK, 15
- ID
 - amd_internal.h, 110
- if
 - compile_direct.m, 42
 - compile_indirect.m, 48
 - make_abip.m, 257, 258
- IMPLIES
 - amd_internal.h, 111
- include/abip.h, 169, 173
- include/abip_blas.h, 175
- include/adaptive.h, 176, 177
- include/cs.h, 177, 180
- include/ctrlc.h, 181, 182
- include/glbopts.h, 182, 193
- include/linalg.h, 195, 200
- include/linsys.h, 201, 205
- include/normalize.h, 206, 207
- include/util.h, 207, 211
- INCS
 - make_abip.m, 260
- INDETERMINATE_TOL
 - glbopts.h, 189
- indirect.c
 - accum_by_A, 247
 - accum_by_Atrans, 247
 - CG_BEST_TOL, 247
 - CG_MIN_TOL, 247
 - free_lin_sys_work, 248
 - get_lin_sys_method, 248
 - get_lin_sys_summary, 248
 - init_lin_sys_work, 248
 - normalize_A, 248
 - solve_lin_sys, 249
 - un_normalize_A, 249
- INFINITY
 - glbopts.h, 189
- init
 - abip.c, 271
 - abip.h, 171
- init_adapt
 - adaptive.c, 300
 - adaptive.h, 177
- init_lin_sys_work
 - direct.c, 240
 - indirect.c, 248
 - linsys.h, 203
- INT
 - make_abip.m, 260
- Int
 - amd_internal.h, 111
- int
 - make_abip.m, 256
- Int_MAX
 - amd_internal.h, 111
- intel64
 - compile_direct.m, 44
- interface/abip_direct.m, 211, 213
- interface/abip_indirect.m, 214, 216
- ipm_iter
 - ABIP_INFO, 12
- j
 - ABIP_LIN_SYS_WORK, 15
- k
 - ABIP_ADAPTIVE_WORK, 9
- kap
 - ABIP_RESIDUALS, 20
- L
 - ABIP_LIN_SYS_WORK, 15
- l
 - ABIP_ADAPTIVE_WORK, 9
- last_admm_iter
 - ABIP_RESIDUALS, 20
- last_ipm_iter
 - ABIP_RESIDUALS, 20
- last_mu
 - ABIP_RESIDUALS, 20
- LCFLAG
 - make_abip.m, 260
- ldl.c
 - LDL_dsolve, 131
 - LDL_ksolve, 131
 - LDL_itsolve, 131
 - LDL_numeric, 132

- LDL_perm, 132
- LDL_permt, 132
- LDL_symbolic, 132
- LDL_valid_matrix, 133
- LDL_valid_perm, 133
- ldl.h
 - LDL_DATE, 142
 - LDL_dsolve, 142
 - ldl_dsolve, 145
 - LDL_ID, 142
 - LDL_int, 142
 - ldl_l_dsolve, 145
 - ldl_l_ksolve, 145
 - ldl_l_ksolve, 145
 - ldl_l_numeric, 145
 - ldl_l_perm, 146
 - ldl_l_permt, 146
 - ldl_l_symbolic, 146
 - ldl_l_valid_matrix, 146
 - ldl_l_valid_perm, 147
 - LDL_ksolve, 142
 - ldl_ksolve, 147
 - LDL_ksolve, 143
 - ldl_ksolve, 147
 - LDL_MAIN_VERSION, 143
 - LDL_numeric, 143
 - ldl_numeric, 147
 - LDL_perm, 143
 - ldl_perm, 148
 - LDL_permt, 143
 - ldl_permt, 148
 - LDL_SUB_VERSION, 143
 - LDL_SUBSUB_VERSION, 144
 - LDL_symbolic, 144
 - ldl_symbolic, 148
 - LDL_valid_matrix, 144
 - ldl_valid_matrix, 148
 - LDL_valid_perm, 144
 - ldl_valid_perm, 149
 - LDL_VERSION, 144
 - LDL_VERSION_CODE, 144
- LDL_DATE
 - ldl.h, 142
- LDL_dsolve
 - ldl.c, 131
 - ldl.h, 142
- ldl_dsolve
 - ldl.h, 145
- ldl_files
 - compile_direct.m, 44
- LDL_ID
 - ldl.h, 142
- LDL_int
 - ldl.h, 142
- ldl_l_dsolve
 - ldl.h, 145
- ldl_l_ksolve
 - ldl.h, 145
- ldl_l_numeric
 - ldl.h, 145
- ldl_l_perm
 - ldl.h, 146
- ldl_l_permt
 - ldl.h, 146
- ldl_l_symbolic
 - ldl.h, 146
- ldl_l_valid_matrix
 - ldl.h, 146
- ldl_l_valid_perm
 - ldl.h, 147
- LDL_ksolve
 - ldl.c, 131
 - ldl.h, 142
- ldl_ksolve
 - ldl.h, 147
- LDL_MAIN_VERSION
 - ldl.h, 143
- LDL_numeric
 - ldl.c, 132
 - ldl.h, 143
- ldl_numeric
 - ldl.h, 147
- ldl_path
 - compile_direct.m, 44
- LDL_perm
 - ldl.c, 132
 - ldl.h, 143
- ldl_perm
 - ldl.h, 148
- LDL_permt
 - ldl.c, 132
 - ldl.h, 143
- ldl_permt
 - ldl.h, 148
- LDL_SUB_VERSION
 - ldl.h, 143
- LDL_SUBSUB_VERSION
 - ldl.h, 144
- LDL_symbolic
 - ldl.c, 132
 - ldl.h, 144
- ldl_symbolic
 - ldl.h, 148
- LDL_valid_matrix
 - ldl.c, 133
 - ldl.h, 144
- ldl_valid_matrix
 - ldl.h, 148
- LDL_valid_perm
 - ldl.h, 148

- ldl.c, [133](#)
- ldl.h, [144](#)
- ldl_valid_perm
 - ldl.h, [149](#)
- LDL_VERSION
 - ldl.h, [144](#)
- LDL_VERSION_CODE
 - ldl.h, [144](#)
- lib_path
 - compile_direct.m, [45](#)
- linalg.c
 - add_array, [313](#)
 - add_scaled_array, [313](#)
 - dot, [313](#)
 - min_abs_sqrt, [313](#)
 - norm, [314](#)
 - norm_diff, [314](#)
 - norm_inf, [314](#)
 - norm_inf_diff, [314](#)
 - norm_inf_sqrt, [315](#)
 - norm_one, [315](#)
 - norm_one_sqrt, [315](#)
 - norm_sq, [315](#)
 - scale_array, [316](#)
 - set_as_scaled_array, [316](#)
 - set_as_sq, [316](#)
 - set_as_sqrt, [316](#)
- linalg.h
 - add_array, [195](#)
 - add_scaled_array, [196](#)
 - dot, [196](#)
 - min_abs_sqrt, [196](#)
 - norm, [196](#)
 - norm_diff, [197](#)
 - norm_inf, [197](#)
 - norm_inf_diff, [197](#)
 - norm_inf_sqrt, [197](#)
 - norm_one, [198](#)
 - norm_one_sqrt, [198](#)
 - norm_sq, [198](#)
 - scale_array, [198](#)
 - set_as_scaled_array, [199](#)
 - set_as_sq, [199](#)
 - set_as_sqrt, [199](#)
- link
 - compile_direct.m, [45](#)
 - make_abip.m, [260](#)
- linsys.h
 - accum_by_A, [202](#)
 - accum_by_Atrans, [202](#)
 - copy_A_matrix, [202](#)
 - free_A_matrix, [202](#)
 - free_lin_sys_work, [202](#)
 - free_lin_sys_work_pds, [203](#)
 - get_lin_sys_method, [203](#)
 - get_lin_sys_summary, [203](#)
 - init_lin_sys_work, [203](#)
 - normalize_A, [203](#)
 - solve_lin_sys, [204](#)
 - un_normalize_A, [204](#)
 - validate_lin_sys, [204](#)
- linsys/abip_pardiso.c, [216](#), [217](#)
- linsys/abip_pardiso.h, [218](#), [222](#)
- linsys/amatrix.h, [223](#)
- linsys/common.c, [224](#), [227](#)
- linsys/common.h, [235](#), [237](#)
- linsys/direct.c, [238](#), [242](#)
- linsys/direct.h, [245](#), [246](#)
- linsys/indirect.c, [246](#), [249](#)
- linsys/indirect.h, [254](#), [255](#)
- linux
 - compile_direct.m, [45](#)
- LOCS
 - make_abip.m, [260](#)
- M
 - ABIP_LIN_SYS_WORK, [16](#)
- m
 - ABIP_A_DATA_MATRIX, [7](#)
 - ABIP_PROBLEM_DATA, [18](#)
 - ABIP_WORK, [33](#)
- main
 - abip.c, [271](#)
 - abip.h, [172](#)
- make_abip.m, [255](#)
 - abip_common_linsys, [258](#)
 - abip_common_src, [258](#)
 - abip_mexfile, [258](#)
 - addpath, [256](#)
 - arr, [259](#)
 - BLASLIB, [259](#)
 - c, [259](#)
 - common_abip, [259](#)
 - compile_direct, [257](#)
 - compile_indirect, [257](#)
 - delete, [257](#)
 - elseif, [257](#)
 - EXTRA_VERBOSE, [259](#)
 - float, [259](#)
 - gpu, [260](#)
 - if, [257](#), [258](#)
 - INCS, [260](#)
 - INT, [260](#)
 - int, [256](#)
 - LCFLAG, [260](#)
 - link, [260](#)
 - LOCS, [260](#)
 - movefile, [258](#)
 - WARNING, [261](#)
- malloc_func
 - SuiteSparse_config_struct, [38](#)
- MAX
 - amd_internal.h, [111](#)
 - glbopts.h, [190](#)
- MAX_ADMM_ITERS
 - glbopts.h, [190](#)
- max_admm_iters

- ABIP_SETTINGS, 26
- MAX_IPM_ITERS
 - glbopts.h, 190
- max_ipm_iters
 - ABIP_SETTINGS, 26
- MAX_SCALE
 - common.c, 224
 - normalize.c, 320
- max_time
 - ABIP_SETTINGS, 26
- mean_norm_col_A
 - ABIP_SCALING, 22
- mean_norm_row_A
 - ABIP_SCALING, 22
- mexext
 - compile_direct.m, 45
- mexfile/abip_mex.c, 262, 264
- mexfile/abip_version_mex.c, 269, 270
- mexFunction
 - abip_mex.c, 263
 - abip_version_mex.c, 269
- MIN
 - amd_internal.h, 111
 - glbopts.h, 190
- min_abs_sqrt
 - linalg.c, 313
 - linalg.h, 196
- MIN_SCALE
 - common.c, 225
 - normalize.c, 320
- MKL
 - compile_direct.m, 45
- mkl_macro
 - compile_direct.m, 46
- movefile
 - make_abip.m, 258
- mu
 - ABIP_WORK, 33
- n
 - ABIP_A_DATA_MATRIX, 8
 - ABIP_PROBLEM_DATA, 18
 - ABIP_WORK, 33
- NAN
 - glbopts.h, 190
- nm_b
 - ABIP_WORK, 34
- nm_c
 - ABIP_WORK, 34
- norm
 - linalg.c, 314
 - linalg.h, 196
- norm_diff
 - linalg.c, 314
 - linalg.h, 197
- norm_inf
 - linalg.c, 314
 - linalg.h, 197
- norm_inf_diff
 - linalg.c, 314
 - linalg.h, 197
- norm_inf_sqrt
 - linalg.c, 315
 - linalg.h, 197
- norm_one
 - linalg.c, 315
 - linalg.h, 198
- norm_one_sqrt
 - linalg.c, 315
 - linalg.h, 198
- norm_sq
 - linalg.c, 315
 - linalg.h, 198
- NORMALIZE
 - glbopts.h, 190
- normalize
 - ABIP_SETTINGS, 27
- normalize.c
 - calc_scaled_resids, 321
 - MAX_SCALE, 320
 - MIN_SCALE, 320
 - normalize_b_c, 321
 - normalize_warm_start, 321
 - un_normalize_sol, 321
- normalize.h
 - calc_scaled_resids, 206
 - normalize_b_c, 206
 - normalize_warm_start, 206
 - un_normalize_sol, 207
- normalize_A
 - direct.c, 241
 - indirect.c, 248
 - linsys.h, 203
- normalize_b_c
 - normalize.c, 321
 - normalize.h, 206
- normalize_warm_start
 - normalize.c, 321
 - normalize.h, 206
- oneapi
 - compile_direct.m, 46
- origin_rescale
 - ABIP_SETTINGS, 27
- P
 - ABIP_LIN_SYS_WORK, 16
- p
 - ABIP_A_DATA_MATRIX, 8
 - ABIP_LIN_SYS_WORK, 16
 - ABIP_WORK, 34
- pardiso
 - abip_pardiso.h, 221
- PARDISO_BACKWARD
 - abip_pardiso.h, 219
- PARDISO_FAC
 - abip_pardiso.h, 219
- PARDISO_FORWARD

- abip_pardiso.h, 219
- PARDISO_FREE
 - abip_pardiso.h, 219
- PARDISO_OK
 - abip_pardiso.h, 220
- PARDISO_SOLVE
 - abip_pardiso.h, 220
- pardiso_src
 - compile_direct.m, 46
- PARDISO_SYM
 - abip_pardiso.h, 220
- PARDISO_SYM_FAC
 - abip_pardiso.h, 220
- pardiso_work
 - ABIP_LIN_SYS_WORK, 16
- pardisoFactorize
 - abip_pardiso.c, 216
 - abip_pardiso.h, 221
- pardisoFree
 - abip_pardiso.c, 217
 - abip_pardiso.h, 221
- PARDISOINDEX
 - abip_pardiso.h, 220
- pardisoinit
 - abip_pardiso.h, 222
- pardisoSolve
 - abip_pardiso.c, 217
 - abip_pardiso.h, 222
- parse_warm_start
 - abip_mex.c, 263
- pc_ruiz_rescale
 - ABIP_SETTINGS, 27
- pfeasopt
 - ABIP_SETTINGS, 27
- PIVOTING
 - abip_pardiso.h, 220
- platform
 - compile_direct.m, 46
- pobj
 - ABIP_INFO, 12
- POWF
 - glbopts.h, 191
- pr
 - ABIP_WORK, 34
- PRI
 - amd_info.c, 103
- print_array
 - util.c, 324
 - util.h, 209
- print_data
 - util.c, 325
 - util.h, 209
- print_work
 - util.c, 325
 - util.h, 209
- PRINTF
 - amd_internal.h, 111
- printf_func
 - SuiteSparse_config_struct, 38
- PRIVATE
 - amd_internal.h, 112
- qp_rescale
 - ABIP_SETTINGS, 27
- R
 - abip_direct.m, 212
 - abip_indirect.m, 214
- r
 - ABIP_LIN_SYS_WORK, 16
- realloc_func
 - SuiteSparse_config_struct, 39
- rel_gap
 - ABIP_INFO, 12
 - ABIP_RESIDUALS, 20
- res_dual
 - ABIP_INFO, 12
 - ABIP_RESIDUALS, 21
- res_infeas
 - ABIP_INFO, 13
 - ABIP_RESIDUALS, 21
- res_pri
 - ABIP_INFO, 13
 - ABIP_RESIDUALS, 21
- res_unbdd
 - ABIP_INFO, 13
 - ABIP_RESIDUALS, 21
- restart_fre
 - ABIP_SETTINGS, 27
- restart_thresh
 - ABIP_SETTINGS, 28
- RETURN
 - glbopts.h, 191
- RHO_Y
 - glbopts.h, 191
- rho_y
 - ABIP_SETTINGS, 28
- ruiz_iter
 - ABIP_SETTINGS, 28
- s
 - ABIP_SOL_VARS, 29
- SAFEDIV_POS
 - glbopts.h, 191
- sc_b
 - ABIP_WORK, 34
- sc_c
 - ABIP_WORK, 34
- scal
 - ABIP_WORK, 35
- SCALE
 - glbopts.h, 191
- scale
 - ABIP_SETTINGS, 28
- scale_array
 - linalg.c, 316
 - linalg.h, 198

- self
 - compile_direct.m, 46
- set_as_scaled_array
 - linalg.c, 316
 - linalg.h, 199
- set_as_sq
 - linalg.c, 316
 - linalg.h, 199
- set_as_sqrt
 - linalg.c, 316
 - linalg.h, 199
- set_default_settings
 - util.c, 325
 - util.h, 209
- set_output_field
 - abip_mex.c, 263
- setup_time
 - ABIP_INFO, 13
- sigma
 - ABIP_WORK, 35
- SIZE_T_MAX
 - amd_internal.h, 112
- solve
 - abip.c, 271
 - abip.h, 172
- solve_lin_sys
 - direct.c, 241
 - indirect.c, 249
 - linsys.h, 204
- solve_time
 - ABIP_INFO, 13
- sp
 - ABIP_PROBLEM_DATA, 18
 - ABIP_WORK, 35
- SPARSITY_RATIO
 - glbopts.h, 191
- sparsity_ratio
 - ABIP_SETTINGS, 28
- SQRTF
 - glbopts.h, 192
- src/abip.c, 270, 272
- src/abip_version.c, 298, 299
- src/adaptive.c, 299, 300
- src/cs.c, 305, 307
- src/ctrlc.c, 311
- src/linalg.c, 312, 317
- src/normalize.c, 320, 322
- src/util.c, 323, 327
- status
 - ABIP_INFO, 13
- status_val
 - ABIP_INFO, 14
- stgs
 - ABIP_PROBLEM_DATA, 19
 - ABIP_WORK, 35
- str_toc
 - util.c, 325
 - util.h, 210
- SuiteSparse_calloc
 - SuiteSparse_config.c, 151
 - SuiteSparse_config.h, 163
- SuiteSparse_config
 - SuiteSparse_config.c, 153
 - SuiteSparse_config.h, 165
- SuiteSparse_config.c
 - SuiteSparse_calloc, 151
 - SuiteSparse_config, 153
 - SuiteSparse_divcomplex, 151
 - SuiteSparse_finish, 151
 - SuiteSparse_free, 151
 - SuiteSparse_hypot, 152
 - SuiteSparse_malloc, 152
 - SuiteSparse_realloc, 152
 - SuiteSparse_start, 152
 - SuiteSparse_tic, 152
 - SuiteSparse_time, 153
 - SuiteSparse_toc, 153
 - SuiteSparse_version, 153
- SuiteSparse_config.h
 - SuiteSparse_calloc, 163
 - SuiteSparse_config, 165
 - SUITESPARSE_DATE, 161
 - SuiteSparse_divcomplex, 163
 - SuiteSparse_finish, 163
 - SuiteSparse_free, 164
 - SUITESPARSE_HAS_VERSION_FUNCTION, 161
 - SuiteSparse_hypot, 164
 - SuiteSparse_long, 161
 - SuiteSparse_long_id, 161
 - SuiteSparse_long_idd, 162
 - SuiteSparse_long_max, 162
 - SUITESPARSE_MAIN_VERSION, 162
 - SuiteSparse_malloc, 164
 - SUITESPARSE_PRINTF, 162
 - SuiteSparse_realloc, 164
 - SuiteSparse_start, 164
 - SUITESPARSE_SUB_VERSION, 162
 - SUITESPARSE_SUBSUB_VERSION, 162
 - SuiteSparse_tic, 165
 - SuiteSparse_time, 165
 - SuiteSparse_toc, 165
 - SUITESPARSE_VER_CODE, 163
 - SUITESPARSE_VERSION, 163
 - SuiteSparse_version, 165
- SuiteSparse_config_struct, 37
 - calloc_func, 38
 - divcomplex_func, 38
 - free_func, 38
 - hypot_func, 38
 - malloc_func, 38
 - printf_func, 38
 - realloc_func, 39
- SUITESPARSE_DATE
 - SuiteSparse_config.h, 161
- SuiteSparse_divcomplex
 - SuiteSparse_config.c, 151

- SuiteSparse_config.h, 163
- SuiteSparse_finish
 - SuiteSparse_config.c, 151
 - SuiteSparse_config.h, 163
- SuiteSparse_free
 - SuiteSparse_config.c, 151
 - SuiteSparse_config.h, 164
- SUITESPARSE_HAS_VERSION_FUNCTION
 - SuiteSparse_config.h, 161
- SuiteSparse_hypot
 - SuiteSparse_config.c, 152
 - SuiteSparse_config.h, 164
- SuiteSparse_long
 - SuiteSparse_config.h, 161
- SuiteSparse_long_id
 - SuiteSparse_config.h, 161
- SuiteSparse_long_idd
 - SuiteSparse_config.h, 162
- SuiteSparse_long_max
 - SuiteSparse_config.h, 162
- SUITESPARSE_MAIN_VERSION
 - SuiteSparse_config.h, 162
- SuiteSparse_malloc
 - SuiteSparse_config.c, 152
 - SuiteSparse_config.h, 164
- SUITESPARSE_PRINTF
 - SuiteSparse_config.h, 162
- SuiteSparse_realloc
 - SuiteSparse_config.c, 152
 - SuiteSparse_config.h, 164
- SuiteSparse_start
 - SuiteSparse_config.c, 152
 - SuiteSparse_config.h, 164
- SUITESPARSE_SUB_VERSION
 - SuiteSparse_config.h, 162
- SUITESPARSE_SUBSUB_VERSION
 - SuiteSparse_config.h, 162
- SuiteSparse_tic
 - SuiteSparse_config.c, 152
 - SuiteSparse_config.h, 165
- SuiteSparse_time
 - SuiteSparse_config.c, 153
 - SuiteSparse_config.h, 165
- SuiteSparse_toc
 - SuiteSparse_config.c, 153
 - SuiteSparse_config.h, 165
- SUITESPARSE_VER_CODE
 - SuiteSparse_config.h, 163
- SUITESPARSE_VERSION
 - SuiteSparse_config.h, 163
- SuiteSparse_version
 - SuiteSparse_config.c, 153
 - SuiteSparse_config.h, 165
- SYMBOLIC
 - abip_pardiso.h, 221
- tau
 - ABIP_RESIDUALS, 21
- tic
 - util.c, 326
 - util.h, 210
- tmp
 - ABIP_LIN_SYS_WORK, 16
- toc
 - util.c, 326
 - util.h, 210
- tocq
 - util.c, 326
 - util.h, 210
- tot_cg_its
 - ABIP_LIN_SYS_WORK, 17
- total_adapt_time
 - ABIP_ADAPTIVE_WORK, 9
- total_solve_time
 - ABIP_LIN_SYS_WORK, 17
- TRUE
 - amd_internal.h, 112
- u
 - ABIP_ADAPTIVE_WORK, 10
 - ABIP_WORK, 35
- u_avg
 - ABIP_WORK, 35
- u_avgcon
 - ABIP_WORK, 36
- u_next
 - ABIP_ADAPTIVE_WORK, 10
- u_prev
 - ABIP_ADAPTIVE_WORK, 10
 - ABIP_WORK, 36
- u_sumcon
 - ABIP_WORK, 36
- u_t
 - ABIP_WORK, 36
- un_normalize_A
 - direct.c, 241
 - indirect.c, 249
 - linsys.h, 204
- un_normalize_sol
 - normalize.c, 321
 - normalize.h, 207
- UNFLIP
 - amd_internal.h, 112
- ut
 - ABIP_ADAPTIVE_WORK, 10
- ut_next
 - ABIP_ADAPTIVE_WORK, 10
- util.c
 - free_data, 324
 - free_sol, 324
 - print_array, 324
 - print_data, 325
 - print_work, 325
 - set_default_settings, 325
 - str_toc, 325
 - tic, 326
 - toc, 326
 - tocq, 326

util.h

- ABIP, [208](#)
- free_data, [208](#)
- free_sol, [208](#)
- print_array, [209](#)
- print_data, [209](#)
- print_work, [209](#)
- set_default_settings, [209](#)
- str_toc, [210](#)
- tic, [210](#)
- toc, [210](#)
- tocq, [210](#)

v

- ABIP_ADAPTIVE_WORK, [10](#)
- ABIP_WORK, [36](#)

v_avg

- ABIP_WORK, [36](#)

v_avgcon

- ABIP_WORK, [37](#)

v_next

- ABIP_ADAPTIVE_WORK, [11](#)

v_prev

- ABIP_ADAPTIVE_WORK, [11](#)
- ABIP_WORK, [37](#)

v_sumcon

- ABIP_WORK, [37](#)

validate_lin_sys

- common.c, [227](#)
- linsys.h, [204](#)

VERBOSE

- glbopts.h, [192](#)

verbose

- ABIP_SETTINGS, [28](#)

version

- abip.h, [172](#)
- abip_version.c, [298](#)

WARM_START

- glbopts.h, [192](#)

warm_start

- ABIP_SETTINGS, [29](#)

WARNING

- make_abip.m, [261](#)

x

- ABIP_A_DATA_MATRIX, [8](#)
- abip_direct.m, [213](#)
- abip_indirect.m, [215](#)
- ABIP_SOL_VARS, [29](#)

y

- ABIP_SOL_VARS, [30](#)

z

- ABIP_LIN_SYS_WORK, [17](#)