ABIP-QCP

Generated by Doxygen 1.9.5

Chapter 1

Class Index

1.1 Class List

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

Chapter 2

File Index

2.1 File List

Here is a list of all files with brief descriptions:

Chapter 3

Class Documentation

3.1 ABIP_A_DATA_MATRIX Struct Reference

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

Public Attributes

- abip_float * x
- abip_int * i
- abip_int * p
- abip_int m
- abip_int n

3.1.1 Detailed Description

Definition at line 11 of file amatrix.h.

3.1.2 Member Data Documentation

3.1.2.1 i

```
abip_int* ABIP_A_DATA_MATRIX::i
```

Definition at line 14 of file amatrix.h.

3.1.2.2 m

`abip_int ABIP_A_DATA_MATRIX::m`

Definition at line 16 of file amatrix.h.

3.1.2.3 n

`abip_int ABIP_A_DATA_MATRIX::n`

Definition at line 17 of file amatrix.h.

3.1.2.4 p

`abip_int* ABIP_A_DATA_MATRIX::p`

Definition at line 15 of file amatrix.h.

3.1.2.5 x

`abip_float* ABIP_A_DATA_MATRIX::x`

Definition at line 13 of file amatrix.h.

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

- include/amatrix.h

3.2 ABIP_CONE Struct Reference

`#include <abip.h>`

Public Attributes

- abip_int * q
- abip_int qsize
- abip_int * rq
- abip_int rqsize
- abip_int f
- abip_int z
- abip_int l

3.2.1 Detailed Description

Definition at line 67 of file abip.h.

3.2.2 Member Data Documentation

3.2.2.1 f

`abip_int ABIP_CONE::f`

Definition at line 72 of file abip.h.

3.2.2.2 l

`abip_int ABIP_CONE::l`

Definition at line 74 of file abip.h.

3.2.2.3 q

`abip_int* ABIP_CONE::q`

Definition at line 68 of file abip.h.

3.2.2.4 qsize

`abip_int ABIP_CONE::qsize`

Definition at line 69 of file abip.h.

3.2.2.5 rq

`abip_int* ABIP_CONE::rq`

Definition at line 70 of file abip.h.

3.2.2.6 rqsize

```
abip_int ABIP_CONE::rqsize
```

Definition at line 71 of file abip.h.

3.2.2.7 z

```
abip_int ABIP_CONE::z
```

Definition at line 73 of file abip.h.

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

- include/abip.h

3.3 ABIP_INFO Struct Reference

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

Public Attributes

- char status [32]
- abip_int status_val
- abip_int ipm_iter
- abip_int admm_iter
- abip_float pobj
- abip_float dobj
- abip_float res_pri
- abip_float res_dual
- abip_float rel_gap
- abip_float res_infeas
- abip_float res_unbdd
- abip_float setup_time
- abip_float solve_time
- abip_float avg_linsys_time
- abip_float avg_cg_iters

3.3.1 Detailed Description

Definition at line 139 of file abip.h.

3.3.2 Member Data Documentation

3.3.2.1 admm_iter

`abip_int ABIP_INFO::admm_iter`

Definition at line 144 of file abip.h.

3.3.2.2 avg_cg_iters

`abip_float ABIP_INFO::avg_cg_iters`

Definition at line 157 of file abip.h.

3.3.2.3 avg_linsys_time

`abip_float ABIP_INFO::avg_linsys_time`

Definition at line 156 of file abip.h.

3.3.2.4 dobj

`abip_float ABIP_INFO::dobj`

Definition at line 147 of file abip.h.

3.3.2.5 ipm_iter

`abip_int ABIP_INFO::ipm_iter`

Definition at line 143 of file abip.h.

3.3.2.6 pobj

`abip_float ABIP_INFO::pobj`

Definition at line 146 of file abip.h.

3.3.2.7 rel_gap

```
abip_float ABIP_INFO::rel_gap
```

Definition at line 150 of file abip.h.

3.3.2.8 res_dual

```
abip_float ABIP_INFO::res_dual
```

Definition at line 149 of file abip.h.

3.3.2.9 res_infeas

```
abip_float ABIP_INFO::res_infeas
```

Definition at line 151 of file abip.h.

3.3.2.10 res_pri

```
abip_float ABIP_INFO::res_pri
```

Definition at line 148 of file abip.h.

3.3.2.11 res_unbdd

```
abip_float ABIP_INFO::res_unbdd
```

Definition at line 152 of file abip.h.

3.3.2.12 setup_time

```
abip_float ABIP_INFO::setup_time
```

Definition at line 154 of file abip.h.

3.3.2.13 solve_time

`abip_float ABIP_INFO::solve_time`

Definition at line 155 of file abip.h.

3.3.2.14 status

`char ABIP_INFO::status[32]`

Definition at line 141 of file abip.h.

3.3.2.15 status_val

`abip_int ABIP_INFO::status_val`

Definition at line 142 of file abip.h.

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

- include/abip.h

3.4 ABIP_LIN_SYS_WORK Struct Reference

`#include <linsys.h>`

Public Attributes

- abip_float total_solve_time
- cs ∗ K
- _MKL_DSS_HANDLE_t handle
- void ∗ pt [64]
- MKL_INT iparm [64]
- MKL_INT maxfct
- MKL_INT mnum
- MKL_INT error
- MKL_INT msglvl
- abip_float ddum
- MKL_INT idum
- MKL_INT mtype
- abip_float ∗ M
- abip_int total_cg_iters
- css ∗ S
- csn ∗ N
- cs ∗ L
- abip_float ∗ Dinv
- abip_int nnz_LDL
- abip_int ∗ P
- abip_float ∗ bp
- abip_float ∗ U

3.4.1 Detailed Description

Definition at line 23 of file linsys.h.

3.4.2 Member Data Documentation

3.4.2.1 bp

`abip_float* ABIP_LIN_SYS_WORK::bp`

Definition at line 59 of file linsys.h.

3.4.2.2 ddum

`abip_float ABIP_LIN_SYS_WORK::ddum`

Definition at line 38 of file linsys.h.

3.4.2.3 Dinv

`abip_float* ABIP_LIN_SYS_WORK::Dinv`

Definition at line 56 of file linsys.h.

3.4.2.4 error

`MKL_INT ABIP_LIN_SYS_WORK::error`

Definition at line 37 of file linsys.h.

3.4.2.5 handle

`_MKL_DSS_HANDLE_t ABIP_LIN_SYS_WORK::handle`

Definition at line 31 of file linsys.h.

3.4.2.6 idum

MKL_INT ABIP_LIN_SYS_WORK::idum

Definition at line 39 of file linsys.h.

3.4.2.7 iparm

MKL_INT ABIP_LIN_SYS_WORK::iparm[64]

Definition at line 36 of file linsys.h.

3.4.2.8 K

cs* ABIP_LIN_SYS_WORK::K

Definition at line 28 of file linsys.h.

3.4.2.9 L

cs* ABIP_LIN_SYS_WORK::L

Definition at line 55 of file linsys.h.

3.4.2.10 M

abip_float* ABIP_LIN_SYS_WORK::M

Definition at line 45 of file linsys.h.

3.4.2.11 maxfct

MKL_INT ABIP_LIN_SYS_WORK::maxfct

Definition at line 37 of file linsys.h.

3.4.2.12 mnum

`MKL_INT ABIP_LIN_SYS_WORK::mnum`

Definition at line 37 of file linsys.h.

3.4.2.13 msglvl

`MKL_INT ABIP_LIN_SYS_WORK::msglvl`

Definition at line 37 of file linsys.h.

3.4.2.14 mtype

`MKL_INT ABIP_LIN_SYS_WORK::mtype`

Definition at line 40 of file linsys.h.

3.4.2.15 N

`csn* ABIP_LIN_SYS_WORK::N`

Definition at line 51 of file linsys.h.

3.4.2.16 nnz_LDL

`abip_int ABIP_LIN_SYS_WORK::nnz_LDL`

Definition at line 57 of file linsys.h.

3.4.2.17 P

`abip_int* ABIP_LIN_SYS_WORK::P`

Definition at line 58 of file linsys.h.

3.4.2.18 pt

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

Definition at line 35 of file linsys.h.

3.4.2.19 S

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

Definition at line 50 of file linsys.h.

3.4.2.20 total_cg_iters

```
abip_int ABIP_LIN_SYS_WORK::total_cg_iters
```

Definition at line 46 of file linsys.h.

3.4.2.21 total_solve_time

```
abip_float ABIP_LIN_SYS_WORK::total_solve_time
```

Definition at line 25 of file linsys.h.

3.4.2.22 U

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

Definition at line 64 of file linsys.h.

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

- include/linsys.h

3.5 ABIP_PROBLEM_DATA Struct Reference

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

Public Attributes

- abip_int m
- abip_int n
- ABIPMatrix * A
- ABIPMatrix * Q
- abip_float * b
- abip_float * c
- abip_float lambda
- ABIPSettings * stgs

3.5.1 Detailed Description

Definition at line 79 of file abip.h.

3.5.2 Member Data Documentation

3.5.2.1 A

`ABIPMatrix* ABIP_PROBLEM_DATA::A`

Definition at line 83 of file abip.h.

3.5.2.2 b

`abip_float* ABIP_PROBLEM_DATA::b`

Definition at line 86 of file abip.h.

3.5.2.3 c

`abip_float* ABIP_PROBLEM_DATA::c`

Definition at line 87 of file abip.h.

3.5.2.4 lambda

`abip_float ABIP_PROBLEM_DATA::lambda`

Definition at line 89 of file abip.h.

3.5.2.5 m

`abip_int ABIP_PROBLEM_DATA::m`

Definition at line 81 of file abip.h.

3.5.2.6 n

`abip_int ABIP_PROBLEM_DATA::n`

Definition at line 82 of file abip.h.

3.5.2.7 Q

`ABIPMatrix* ABIP_PROBLEM_DATA::Q`

Definition at line 84 of file abip.h.

3.5.2.8 stgs

`ABIPSettings* ABIP_PROBLEM_DATA::stgs`

Definition at line 90 of file abip.h.

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

- include/abip.h

3.6 ABIP_RESIDUALS Struct Reference

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

Public Attributes

- abip_int last_ipm_iter
- abip_int last_admm_iter
- abip_float last_mu
- abip_float res_pri
- abip_float res_dual
- abip_float rel_gap
- abip_float res_infeas
- abip_float res_unbdd
- abip_float ct_x_by_tau
- abip_float bt_y_by_tau
- abip_float pobj
- abip_float dobj
- abip_float tau
- abip_float kap
- abip_float res_dif
- abip_float error_ratio
- abip_float Ax_b_norm
- abip_float Qx_ATy_c_s_norm

3.6.1 Detailed Description

Definition at line 182 of file abip.h.

3.6.2 Member Data Documentation

3.6.2.1 Ax_b_norm

```
abip_float ABIP_RESIDUALS::Ax_b_norm
```

Definition at line 206 of file abip.h.

3.6.2.2 bt_y_by_tau

```
abip_float ABIP_RESIDUALS::bt_y_by_tau
```

Definition at line 195 of file abip.h.

3.6.2.3 ct_x_by_tau

```
abip_float ABIP_RESIDUALS::ct_x_by_tau
```

Definition at line 194 of file abip.h.

3.6.2.4 dobj

```
abip_float ABIP_RESIDUALS::dobj
```

Definition at line 198 of file abip.h.

3.6.2.5 error_ratio

```
abip_float ABIP_RESIDUALS::error_ratio
```

Definition at line 204 of file abip.h.

3.6.2.6 kap

```
abip_float ABIP_RESIDUALS::kap
```

Definition at line 201 of file abip.h.

3.6.2.7 last_admm_iter

```
abip_int ABIP_RESIDUALS::last_admm_iter
```

Definition at line 185 of file abip.h.

3.6.2.8 last_ipm_iter

```
abip_int ABIP_RESIDUALS::last_ipm_iter
```

Definition at line 184 of file abip.h.

3.6.2.9 last_mu

```
abip_float ABIP_RESIDUALS::last_mu
```

Definition at line 186 of file abip.h.

3.6.2.10 pobj

```
abip_float ABIP_RESIDUALS::pobj
```

Definition at line 197 of file abip.h.

3.6.2.11 Qx_ATy_c_s_norm

```
abip_float ABIP_RESIDUALS::Qx_ATy_c_s_norm
```

Definition at line 207 of file abip.h.

3.6.2.12 rel_gap

```
abip_float ABIP_RESIDUALS::rel_gap
```

Definition at line 190 of file abip.h.

3.6.2.13 res_dif

```
abip_float ABIP_RESIDUALS::res_dif
```

Definition at line 203 of file abip.h.

3.6.2.14 res_dual

```
abip_float ABIP_RESIDUALS::res_dual
```

Definition at line 189 of file abip.h.

3.6.2.15 res_infeas

`abip_float ABIP_RESIDUALS::res_infeas`

Definition at line 191 of file abip.h.

3.6.2.16 res_pri

`abip_float ABIP_RESIDUALS::res_pri`

Definition at line 188 of file abip.h.

3.6.2.17 res_unbdd

`abip_float ABIP_RESIDUALS::res_unbdd`

Definition at line 192 of file abip.h.

3.6.2.18 tau

`abip_float ABIP_RESIDUALS::tau`

Definition at line 200 of file abip.h.

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

- include/abip.h

3.7 ABIP_SETTINGS Struct Reference

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

Public Attributes

- abip_int normalize
- abip_int scale_E
- abip_int scale_bc
- abip_float scale
- abip_float rho_x
- abip_float rho_y
- abip_float rho_tau
- abip_int max_ipm_iters
- abip_int max_admm_iters
- abip_float eps
- abip_float eps_p
- abip_float eps_d
- abip_float eps_g
- abip_float eps_inf
- abip_float eps_unb
- abip_float err_dif
- abip_float alpha
- abip_float cg_rate
- abip_int use_indirect
- abip_int inner_check_period
- abip_int outer_check_period
- abip_int verbose
- abip_int linsys_solver
- abip_int prob_type
- abip_float time_limit
- abip_float psi
- abip_int origin_scaling
- abip_int ruiz_scaling
- abip_int pc_scaling

3.7.1 Detailed Description

Definition at line 93 of file abip.h.

3.7.2 Member Data Documentation

3.7.2.1 alpha

```
abip_float ABIP_SETTINGS::alpha
```

Definition at line 113 of file abip.h.

3.7.2.2 cg_rate

`abip_float ABIP_SETTINGS::cg_rate`

Definition at line 114 of file abip.h.

3.7.2.3 eps

`abip_float ABIP_SETTINGS::eps`

Definition at line 105 of file abip.h.

3.7.2.4 eps_d

`abip_float ABIP_SETTINGS::eps_d`

Definition at line 107 of file abip.h.

3.7.2.5 eps_g

`abip_float ABIP_SETTINGS::eps_g`

Definition at line 108 of file abip.h.

3.7.2.6 eps_inf

`abip_float ABIP_SETTINGS::eps_inf`

Definition at line 109 of file abip.h.

3.7.2.7 eps_p

`abip_float ABIP_SETTINGS::eps_p`

Definition at line 106 of file abip.h.

3.7.2.8 eps_unb

`abip_float ABIP_SETTINGS::eps_unb`

Definition at line 110 of file abip.h.

3.7.2.9 err_dif

`abip_float ABIP_SETTINGS::err_dif`

Definition at line 112 of file abip.h.

3.7.2.10 inner_check_period

`abip_int ABIP_SETTINGS::inner_check_period`

Definition at line 117 of file abip.h.

3.7.2.11 linsys_solver

`abip_int ABIP_SETTINGS::linsys_solver`

Definition at line 121 of file abip.h.

3.7.2.12 max_admm_iters

`abip_int ABIP_SETTINGS::max_admm_iters`

Definition at line 104 of file abip.h.

3.7.2.13 max_ipm_iters

`abip_int ABIP_SETTINGS::max_ipm_iters`

Definition at line 103 of file abip.h.

3.7.2.14 normalize

`abip_int ABIP_SETTINGS::normalize`

Definition at line 95 of file abip.h.

3.7.2.15 origin_scaling

`abip_int ABIP_SETTINGS::origin_scaling`

Definition at line 126 of file abip.h.

3.7.2.16 outer_check_period

`abip_int ABIP_SETTINGS::outer_check_period`

Definition at line 118 of file abip.h.

3.7.2.17 pc_scaling

`abip_int ABIP_SETTINGS::pc_scaling`

Definition at line 128 of file abip.h.

3.7.2.18 prob_type

`abip_int ABIP_SETTINGS::prob_type`

Definition at line 122 of file abip.h.

3.7.2.19 psi

`abip_float ABIP_SETTINGS::psi`

Definition at line 124 of file abip.h.

3.7.2.20 rho_tau

`abip_float ABIP_SETTINGS::rho_tau`

Definition at line 101 of file abip.h.

3.7.2.21 rho_x

`abip_float ABIP_SETTINGS::rho_x`

Definition at line 99 of file abip.h.

3.7.2.22 rho_y

`abip_float ABIP_SETTINGS::rho_y`

Definition at line 100 of file abip.h.

3.7.2.23 ruiz_scaling

`abip_int ABIP_SETTINGS::ruiz_scaling`

Definition at line 127 of file abip.h.

3.7.2.24 scale

`abip_float ABIP_SETTINGS::scale`

Definition at line 98 of file abip.h.

3.7.2.25 scale_bc

`abip_int ABIP_SETTINGS::scale_bc`

Definition at line 97 of file abip.h.

3.7.2.26 scale_E

```
abip_int ABIP_SETTINGS::scale_E
```

Definition at line 96 of file abip.h.

3.7.2.27 time_limit

```
abip_float ABIP_SETTINGS::time_limit
```

Definition at line 123 of file abip.h.

3.7.2.28 use_indirect

```
abip_int ABIP_SETTINGS::use_indirect
```

Definition at line 116 of file abip.h.

3.7.2.29 verbose

```
abip_int ABIP_SETTINGS::verbose
```

Definition at line 120 of file abip.h.

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

- include/abip.h

3.8 ABIP_SOL_VARS Struct Reference

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

Public Attributes

- abip_float ∗ x
- abip_float ∗ y
- abip_float ∗ s

3.8.1 Detailed Description

Definition at line 132 of file abip.h.

3.8.2 Member Data Documentation

3.8.2.1 s

`abip_float* ABIP_SOL_VARS::s`

Definition at line 136 of file abip.h.

3.8.2.2 x

`abip_float* ABIP_SOL_VARS::x`

Definition at line 134 of file abip.h.

3.8.2.3 y

`abip_float* ABIP_SOL_VARS::y`

Definition at line 135 of file abip.h.

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

- include/abip.h

3.9 ABIP_WORK Struct Reference

`#include <abip.h>`

Public Attributes

- abip_float sigma
- abip_float gamma
- abip_float mu
- abip_float beta
- abip_float * u
- abip_float * v
- abip_float * v_origin
- abip_float * u_t
- abip_float * rel_ut
- abip_float nm_inf_b
- abip_float nm_inf_c
- abip_int m
- abip_int n
- ABIPMatrix * A
- abip_float * r
- abip_float a

3.9.1 Detailed Description

Definition at line 161 of file abip.h.

3.9.2 Member Data Documentation

3.9.2.1 A

`ABIPMatrix* ABIP_WORK::A`

Definition at line 176 of file abip.h.

3.9.2.2 a

`abip_float ABIP_WORK::a`

Definition at line 178 of file abip.h.

3.9.2.3 beta

`abip_float ABIP_WORK::beta`

Definition at line 166 of file abip.h.

3.9.2.4 gamma

`abip_float ABIP_WORK::gamma`

Definition at line 164 of file abip.h.

3.9.2.5 m

`abip_int ABIP_WORK::m`

Definition at line 174 of file abip.h.

3.9.2.6 mu

`abip_float ABIP_WORK::mu`

Definition at line 165 of file abip.h.

3.9.2.7 n

`abip_int ABIP_WORK::n`

Definition at line 175 of file abip.h.

3.9.2.8 nm_inf_b

`abip_float ABIP_WORK::nm_inf_b`

Definition at line 172 of file abip.h.

3.9.2.9 nm_inf_c

`abip_float ABIP_WORK::nm_inf_c`

Definition at line 173 of file abip.h.

3.9.2.10 r

`abip_float* ABIP_WORK::r`

Definition at line 177 of file abip.h.

3.9.2.11 rel_ut

`abip_float* ABIP_WORK::rel_ut`

Definition at line 171 of file abip.h.

3.9.2.12 sigma

`abip_float ABIP_WORK::sigma`

Definition at line 163 of file abip.h.

3.9.2.13 u

`abip_float* ABIP_WORK::u`

Definition at line 167 of file abip.h.

3.9.2.14 u_t

`abip_float* ABIP_WORK::u_t`

Definition at line 170 of file abip.h.

3.9.2.15 v

`abip_float* ABIP_WORK::v`

Definition at line 168 of file abip.h.

3.9.2.16 v_origin

`abip_float* ABIP_WORK::v_origin`

Definition at line 169 of file abip.h.

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

- include/abip.h

3.10 cs_dmperm_results Struct Reference

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

Public Attributes

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

3.10.1 Detailed Description

Definition at line 82 of file cs.h.

3.10.2 Member Data Documentation

3.10.2.1 cc

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

Definition at line 90 of file cs.h.

3.10.2.2 nb

```
csi cs_dmperm_results::nb
```

Definition at line 88 of file cs.h.

3.10.2.3 p

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

Definition at line 84 of file cs.h.

3.10.2.4 q

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

Definition at line 85 of file cs.h.

3.10.2.5 r

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

Definition at line 86 of file cs.h.

3.10.2.6 rr

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

Definition at line 89 of file cs.h.

3.10.2.7 s

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

Definition at line 87 of file cs.h.

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

- csparse/Include/cs.h

3.11 cs_numeric Struct Reference

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

Public Attributes

- cs ∗ L
- cs ∗ U
- csi ∗ pinv
- double ∗ B

3.11.1 Detailed Description

Definition at line 74 of file cs.h.

3.11.2 Member Data Documentation

3.11.2.1 B

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

Definition at line 79 of file cs.h.

3.11.2.2 L

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

Definition at line 76 of file cs.h.

3.11.2.3 pinv

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

Definition at line 78 of file cs.h.

3.11.2.4 U

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

Definition at line 77 of file cs.h.

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

- csparse/Include/cs.h

3.12 cs_sparse Struct Reference

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

Public Attributes

- csi nzmax
- csi m
- csi n
- csi * p
- csi * i
- double * x
- csi nz

3.12.1 Detailed Description

Definition at line 28 of file cs.h.

3.12.2 Member Data Documentation

3.12.2.1 i

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

Definition at line 34 of file cs.h.

3.12.2.2 m

```
csi cs_sparse::m
```

Definition at line 31 of file cs.h.

3.12.2.3 n

```
csi cs_sparse::n
```

Definition at line 32 of file cs.h.

3.12.2.4 nz

```
csi cs_sparse::nz
```

Definition at line 36 of file cs.h.

3.12.2.5 nzmax

```
csi cs_sparse::nzmax
```

Definition at line 30 of file cs.h.

3.12.2.6 p

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

Definition at line 33 of file cs.h.

3.12.2.7 x

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

Definition at line 35 of file cs.h.

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

- csparse/Include/cs.h

3.13 cs_symbolic Struct Reference

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

Public Attributes

- csi * pinv
- csi * q
- csi * parent
- csi * cp
- csi * leftmost
- csi m2
- double lnz
- double unz

3.13.1 Detailed Description

Definition at line 62 of file cs.h.

3.13.2 Member Data Documentation

3.13.2.1 cp

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

Definition at line 67 of file cs.h.

3.13.2.2 leftmost

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

Definition at line 68 of file cs.h.

3.13.2.3 lnz

```
double cs_symbolic::lnz
```

Definition at line 70 of file cs.h.

3.13.2.4 m2

```
csi cs_symbolic::m2
```

Definition at line 69 of file cs.h.

3.13.2.5 parent

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

Definition at line 66 of file cs.h.

3.13.2.6 pinv

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

Definition at line 64 of file cs.h.

3.13.2.7 q

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

Definition at line 65 of file cs.h.

3.13.2.8 unz

```
double cs_symbolic::unz
```

Definition at line 71 of file cs.h.

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

- csparse/Include/cs.h

3.14 Lasso Struct Reference

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

Public Attributes

- enum problem_type pro_type
- abip_int m
- abip_int n
- abip_int p
- abip_int q
- ABIPLinSysWork * L
- ABIPSettings * stgs
- ABIPData * data
- abip_float sparsity
- abip_float * rho_dr
- ABIPMatrix * A
- ABIPMatrix * Q
- abip_float * b
- abip_float * c
- void(* scaling_data)(lasso *self, ABIPCone *k)
- void(* un_scaling_sol)(lasso *self, ABIPSolution *sol)
- void(* calc_residuals)(lasso *self, ABIPWork *w, ABIPResiduals *r, abip_int ipm_iter, abip_int admm_iter)
- abip_int(* init_spe_linsys_work)(lasso *self)
- abip_int(* solve_spe_linsys)(lasso *self, abip_float *b, abip_float *pcg_warm_start, abip_int iter, abip_float error_ratio)
- void(* free_spe_linsys_work)(lasso *self)
- void(* spe_A_times)(lasso *self, const abip_float *x, abip_float *y)
- void(* spe_AT_times)(lasso *self, const abip_float *x, abip_float *y)
- abip_float(* inner_conv_check)(lasso *self, ABIPWork *w)
- abip_float lambda
- abip_float * D_hat
- abip_float * D
- abip_float * E
- abip_float sc_b
- abip_float sc_c
- abip_float sc
- abip_float sc_cone1
- abip_float sc_cone2

3.14.1 Detailed Description

Definition at line 14 of file lasso_config.h.

3.14.2 Member Data Documentation

3.14.2.1 A

`ABIPMatrix* Lasso::A`

Definition at line 30 of file lasso_config.h.

3.14.2.2 b

`abip_float* Lasso::b`

Definition at line 32 of file lasso_config.h.

3.14.2.3 c

`abip_float* Lasso::c`

Definition at line 33 of file lasso_config.h.

3.14.2.4 calc_residuals

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

Definition at line 38 of file lasso_config.h.

3.14.2.5 D

`abip_float* Lasso::D`

Definition at line 50 of file lasso_config.h.

3.14.2.6 D_hat

`abip_float* Lasso::D_hat`

Definition at line 49 of file lasso_config.h.

3.14.2.7 data

`ABIPData* Lasso::data`

Definition at line 24 of file lasso_config.h.

3.14.2.8 E

`abip_float* Lasso::E`

Definition at line 51 of file lasso_config.h.

3.14.2.9 free_spe_linsys_work

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

Definition at line 41 of file lasso_config.h.

3.14.2.10 init_spe_linsys_work

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

Definition at line 39 of file lasso_config.h.

3.14.2.11 inner_conv_check

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

Definition at line 44 of file lasso_config.h.

3.14.2.12 L

`ABIPLinSysWork* Lasso::L`

Definition at line 22 of file lasso_config.h.

3.14.2.13 lambda

`abip_float Lasso::lambda`

Definition at line 47 of file lasso_config.h.

3.14.2.14 m

`abip_int Lasso::m`

Definition at line 18 of file lasso_config.h.

3.14.2.15 n

`abip_int Lasso::n`

Definition at line 19 of file lasso_config.h.

3.14.2.16 p

`abip_int Lasso::p`

Definition at line 20 of file lasso_config.h.

3.14.2.17 pro_type

`enum problem_type Lasso::pro_type`

Definition at line 17 of file lasso_config.h.

3.14.2.18 q

`abip_int Lasso::q`

Definition at line 21 of file lasso_config.h.

3.14.2.19 Q

`ABIPMatrix* Lasso::Q`

Definition at line 31 of file lasso_config.h.

3.14.2.20 rho_dr

`abip_float* Lasso::rho_dr`

Definition at line 27 of file lasso_config.h.

3.14.2.21 sc

`abip_float Lasso::sc`

Definition at line 54 of file lasso_config.h.

3.14.2.22 sc_b

`abip_float Lasso::sc_b`

Definition at line 52 of file lasso_config.h.

3.14.2.23 sc_c

`abip_float Lasso::sc_c`

Definition at line 53 of file lasso_config.h.

3.14.2.24 sc_cone1

```
abip_float Lasso::sc_cone1
```

Definition at line 55 of file lasso_config.h.

3.14.2.25 sc_cone2

```
abip_float Lasso::sc_cone2
```

Definition at line 56 of file lasso_config.h.

3.14.2.26 scaling_data

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

Definition at line 36 of file lasso_config.h.

3.14.2.27 solve_spe_linsys

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

Definition at line 40 of file lasso_config.h.

3.14.2.28 sparsity

```
abip_float Lasso::sparsity
```

Definition at line 25 of file lasso_config.h.

3.14.2.29 spe_A_times

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

Definition at line 42 of file lasso_config.h.

3.14.2.30 spe_AT_times

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

Definition at line 43 of file lasso_config.h.

3.14.2.31 stgs

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

Definition at line 23 of file lasso_config.h.

3.14.2.32 un_scaling_sol

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

Definition at line 37 of file lasso_config.h.

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

- include/lasso_config.h

3.15 qcp Struct Reference

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

Public Attributes

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

3.15.1 Detailed Description

Definition at line 14 of file qcp_config.h.

3.15.2 Member Data Documentation

3.15.2.1 A

`ABIPMatrix* qcp::A`

Definition at line 30 of file qcp_config.h.

3.15.2.2 b

`abip_float* qcp::b`

Definition at line 32 of file qcp_config.h.

3.15.2.3 c

`abip_float* qcp::c`

Definition at line 33 of file qcp_config.h.

3.15.2.4 calc_residuals

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

Definition at line 38 of file qcp_config.h.

3.15.2.5 D

`abip_float* qcp::D`

Definition at line 48 of file qcp_config.h.

3.15.2.6 data

```
ABIPData* qcp::data
```

Definition at line 24 of file qcp_config.h.

3.15.2.7 E

```
abip_float* qcp::E
```

Definition at line 49 of file qcp_config.h.

3.15.2.8 free_spe_linsys_work

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

Definition at line 41 of file qcp_config.h.

3.15.2.9 init_spe_linsys_work

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

Definition at line 39 of file qcp_config.h.

3.15.2.10 inner_conv_check

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

Definition at line 44 of file qcp_config.h.

3.15.2.11 L

```
ABIPLinSysWork* qcp::L
```

Definition at line 22 of file qcp_config.h.

3.15.2.12 m

`abip_int qcp::m`

Definition at line 18 of file qcp_config.h.

3.15.2.13 n

`abip_int qcp::n`

Definition at line 19 of file qcp_config.h.

3.15.2.14 p

`abip_int qcp::p`

Definition at line 20 of file qcp_config.h.

3.15.2.15 pro_type

`enum problem_type qcp::pro_type`

Definition at line 17 of file qcp_config.h.

3.15.2.16 q

`abip_int qcp::q`

Definition at line 21 of file qcp_config.h.

3.15.2.17 Q

`ABIPMatrix* qcp::Q`

Definition at line 31 of file qcp_config.h.

3.15.2.18 rho_dr

`abip_float* qcp::rho_dr`

Definition at line 27 of file qcp_config.h.

3.15.2.19 sc_b

`abip_float qcp::sc_b`

Definition at line 50 of file qcp_config.h.

3.15.2.20 sc_c

`abip_float qcp::sc_c`

Definition at line 51 of file qcp_config.h.

3.15.2.21 scaling_data

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

Definition at line 36 of file qcp_config.h.

3.15.2.22 solve_spe_linsys

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

Definition at line 40 of file qcp_config.h.

3.15.2.23 sparsity

`abip_float qcp::sparsity`

Definition at line 25 of file qcp_config.h.

3.15.2.24 spe_A_times

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

Definition at line 42 of file qcp_config.h.

3.15.2.25 spe_AT_times

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

Definition at line 43 of file qcp_config.h.

3.15.2.26 stgs

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

Definition at line 23 of file qcp_config.h.

3.15.2.27 un_scaling_sol

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

Definition at line 37 of file qcp_config.h.

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

- include/qcp_config.h

3.16 solve_specific_problem Struct Reference

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

Public Attributes

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

3.16.1 Detailed Description

Definition at line 29 of file abip.h.

3.16.2 Member Data Documentation

3.16.2.1 A

```
ABIPMatrix* solve_specific_problem::A
```

Definition at line 44 of file abip.h.

3.16.2.2 b

```
abip_float* solve_specific_problem::b
```

Definition at line 46 of file abip.h.

3.16.2.3 c

```
abip_float* solve_specific_problem::c
```

Definition at line 47 of file abip.h.

3.16.2.4 calc_residuals

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

Definition at line 52 of file abip.h.

3.16.2.5 data

```
ABIPData* solve_specific_problem::data
```

Definition at line 38 of file abip.h.

3.16.2.6 free_spe_linsys_work

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

Definition at line 55 of file abip.h.

3.16.2.7 init_spe_linsys_work

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

Definition at line 53 of file abip.h.

3.16.2.8 inner_conv_check

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

Definition at line 58 of file abip.h.

3.16.2.9 L

```
ABIPLinSysWork* solve_specific_problem::L
```

Definition at line 36 of file abip.h.

3.16.2.10 m

```
abip_int solve_specific_problem::m
```

Definition at line 32 of file abip.h.

3.16.2.11 n

```
abip_int solve_specific_problem::n
```

Definition at line 33 of file abip.h.

3.16.2.12 p

```
abip_int solve_specific_problem::p
```

Definition at line 34 of file abip.h.

3.16.2.13 pro_type

```
enum problem_type solve_specific_problem::pro_type
```

Definition at line 31 of file abip.h.

3.16.2.14 q

```
abip_int solve_specific_problem::q
```

Definition at line 35 of file abip.h.

3.16.2.15 Q

`ABIPMatrix* solve_specific_problem::Q`

Definition at line 45 of file abip.h.

3.16.2.16 rho_dr

`abip_float* solve_specific_problem::rho_dr`

Definition at line 41 of file abip.h.

3.16.2.17 scaling_data

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

Definition at line 50 of file abip.h.

3.16.2.18 solve_spe_linsys

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

Definition at line 54 of file abip.h.

3.16.2.19 sparsity

`abip_float solve_specific_problem::sparsity`

Definition at line 39 of file abip.h.

3.16.2.20 spe_A_times

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

Definition at line 56 of file abip.h.

3.16.2.21 spe_AT_times

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

Definition at line 57 of file abip.h.

3.16.2.22 stgs

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

Definition at line 37 of file abip.h.

3.16.2.23 un_scaling_sol

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

Definition at line 51 of file abip.h.

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

- include/abip.h

3.17 SuiteSparse_config_struct Struct Reference

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

Public Attributes

- void *(* malloc_func)(size_t)
- void *(* calloc_func)(size_t, size_t)
- void *(* realloc_func)(void *, size_t)
- void(* free_func)(void *)
- int(* printf_func)(const char *,...)
- abip_float(* hypot_func)(abip_float, abip_float)
- int(* divcomplex_func)(abip_float, abip_float, abip_float, abip_float, abip_float *, abip_float *)

3.17.1 Detailed Description

Definition at line 87 of file SuiteSparse_config.h.

3.17.2 Member Data Documentation

3.17.2.1 calloc_func

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

Definition at line 90 of file SuiteSparse_config.h.

3.17.2.2 divcomplex_func

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

Definition at line 95 of file SuiteSparse_config.h.

3.17.2.3 free_func

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

Definition at line 92 of file SuiteSparse_config.h.

3.17.2.4 hypot_func

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

Definition at line 94 of file SuiteSparse_config.h.

3.17.2.5 malloc_func

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

Definition at line 89 of file SuiteSparse_config.h.

3.17.2.6 printf_func

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

Definition at line 93 of file SuiteSparse_config.h.

3.17.2.7 realloc_func

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

Definition at line 91 of file SuiteSparse_config.h.

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

- amd/SuiteSparse_config.h

3.18 Svm Struct Reference

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

Public Attributes

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

- abip_float ∗ sc_E
- abip_float ∗ sc_F
- abip_float sc_b
- abip_float sc_c
- abip_float sc
- abip_float sc_cone1
- abip_float sc_cone2
- ABIPMatrix ∗ wA
- abip_float ∗ wy
- abip_float ∗ wB
- abip_float ∗ wC
- abip_float ∗ wD
- abip_float ∗ wE
- abip_float ∗ wF
- abip_float ∗ wG
- abip_float ∗ wH
- ABIPMatrix ∗ wX

3.18.1 Detailed Description

Definition at line 14 of file svm_config.h.

3.18.2 Member Data Documentation

3.18.2.1 A

`ABIPMatrix* Svm::A`

Definition at line 30 of file svm_config.h.

3.18.2.2 b

`abip_float* Svm::b`

Definition at line 32 of file svm_config.h.

3.18.2.3 c

`abip_float* Svm::c`

Definition at line 33 of file svm_config.h.

3.18.2.4 calc_residuals

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

Definition at line 38 of file svm_config.h.

3.18.2.5 data

ABIPData* Svm::data

Definition at line 24 of file svm_config.h.

3.18.2.6 free_spe_linsys_work

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

Definition at line 41 of file svm_config.h.

3.18.2.7 init_spe_linsys_work

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

Definition at line 39 of file svm_config.h.

3.18.2.8 inner_conv_check

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

Definition at line 44 of file svm_config.h.

3.18.2.9 L

ABIPLinSysWork* Svm::L

Definition at line 22 of file svm_config.h.

3.18.2.10 lambda

`abip_float Svm::lambda`

Definition at line 47 of file svm_config.h.

3.18.2.11 m

`abip_int Svm::m`

Definition at line 18 of file svm_config.h.

3.18.2.12 n

`abip_int Svm::n`

Definition at line 19 of file svm_config.h.

3.18.2.13 p

`abip_int Svm::p`

Definition at line 20 of file svm_config.h.

3.18.2.14 pro_type

`enum problem_type Svm::pro_type`

Definition at line 17 of file svm_config.h.

3.18.2.15 q

`abip_int Svm::q`

Definition at line 21 of file svm_config.h.

3.18.2.16 Q

```
ABIPMatrix* Svm::Q
```

Definition at line 31 of file svm_config.h.

3.18.2.17 rho_dr

```
abip_float* Svm::rho_dr
```

Definition at line 27 of file svm_config.h.

3.18.2.18 sc

```
abip_float Svm::sc
```

Definition at line 54 of file svm_config.h.

3.18.2.19 sc_b

```
abip_float Svm::sc_b
```

Definition at line 52 of file svm_config.h.

3.18.2.20 sc_c

```
abip_float Svm::sc_c
```

Definition at line 53 of file svm_config.h.

3.18.2.21 sc_cone1

```
abip_float Svm::sc_cone1
```

Definition at line 55 of file svm_config.h.

3.18.2.22 sc_cone2

```
abip_float Svm::sc_cone2
```

Definition at line 56 of file svm_config.h.

3.18.2.23 sc_D

```
abip_float* Svm::sc_D
```

Definition at line 49 of file svm_config.h.

3.18.2.24 sc_E

```
abip_float* Svm::sc_E
```

Definition at line 50 of file svm_config.h.

3.18.2.25 sc_F

```
abip_float* Svm::sc_F
```

Definition at line 51 of file svm_config.h.

3.18.2.26 scaling_data

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

Definition at line 36 of file svm_config.h.

3.18.2.27 solve_spe_linsys

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

Definition at line 40 of file svm_config.h.

3.18.2.28 sparsity

```
abip_float Svm::sparsity
```

Definition at line 25 of file svm_config.h.

3.18.2.29 spe_A_times

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

Definition at line 42 of file svm_config.h.

3.18.2.30 spe_AT_times

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

Definition at line 43 of file svm_config.h.

3.18.2.31 stgs

```
ABIPSettings* Svm::stgs
```

Definition at line 23 of file svm_config.h.

3.18.2.32 un_scaling_sol

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

Definition at line 37 of file svm_config.h.

3.18.2.33 wA

```
ABIPMatrix* Svm::wA
```

Definition at line 58 of file svm_config.h.

3.18.2.34 wB

`abip_float* Svm::wB`

Definition at line 60 of file svm_config.h.

3.18.2.35 wC

`abip_float* Svm::wC`

Definition at line 61 of file svm_config.h.

3.18.2.36 wD

`abip_float* Svm::wD`

Definition at line 62 of file svm_config.h.

3.18.2.37 wE

`abip_float* Svm::wE`

Definition at line 63 of file svm_config.h.

3.18.2.38 wF

`abip_float* Svm::wF`

Definition at line 64 of file svm_config.h.

3.18.2.39 wG

`abip_float* Svm::wG`

Definition at line 65 of file svm_config.h.

3.18.2.40 wH

```
abip_float* Svm::wH
```

Definition at line 66 of file svm_config.h.

3.18.2.41 wX

```
ABIPMatrix* Svm::wX
```

Definition at line 67 of file svm_config.h.

3.18.2.42 wy

```
abip_float* Svm::wy
```

Definition at line 59 of file svm_config.h.

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

- include/svm_config.h

3.19 SVMqp Struct Reference

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

Public Attributes

- enum problem_type pro_type
- abip_int m
- abip_int n
- abip_int p
- abip_int q
- ABIPLinSysWork ∗ L
- ABIPSettings ∗ stgs
- ABIPData ∗ data
- abip_float sparsity
- abip_float ∗ rho_dr
- ABIPMatrix ∗ A
- ABIPMatrix ∗ Q
- abip_float ∗ b
- abip_float ∗ c
- void(∗ scaling_data)(svmqp ∗self, ABIPCone ∗k)
- void(∗ un_scaling_sol)(svmqp ∗self, ABIPSolution ∗sol)
- void(∗ calc_residuals)(svmqp ∗self, ABIPWork ∗w, ABIPResiduals ∗r, abip_int ipm_iter, abip_int admm_iter)

- abip_int(∗ init_spe_linsys_work)(svmqp ∗self)
- abip_int(∗ solve_spe_linsys)(svmqp ∗self, abip_float ∗b, abip_float ∗pcg_warm_start, abip_int iter, abip_float error_ratio)
- void(∗ free_spe_linsys_work)(svmqp ∗self)
- void(∗ spe_A_times)(svmqp ∗self, const abip_float ∗x, abip_float ∗y)
- void(∗ spe_AT_times)(svmqp ∗self, const abip_float ∗x, abip_float ∗y)
- abip_float(∗ inner_conv_check)(svmqp ∗self, ABIPWork ∗w)
- abip_float lambda
- abip_float ∗ D
- abip_float ∗ E
- abip_float ∗ F
- abip_float ∗ H
- abip_float sc_b
- abip_float sc_c

3.19.1 Detailed Description

Definition at line 14 of file svm_qp_config.h.

3.19.2 Member Data Documentation

3.19.2.1 A

`ABIPMatrix* SVMqp::A`

Definition at line 30 of file svm_qp_config.h.

3.19.2.2 b

`abip_float* SVMqp::b`

Definition at line 32 of file svm_qp_config.h.

3.19.2.3 c

`abip_float* SVMqp::c`

Definition at line 33 of file svm_qp_config.h.

3.19.2.4 calc_residuals

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

Definition at line 38 of file svm_qp_config.h.

3.19.2.5 D

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

Definition at line 49 of file svm_qp_config.h.

3.19.2.6 data

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

Definition at line 24 of file svm_qp_config.h.

3.19.2.7 E

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

Definition at line 50 of file svm_qp_config.h.

3.19.2.8 F

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

Definition at line 51 of file svm_qp_config.h.

3.19.2.9 free_spe_linsys_work

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

Definition at line 41 of file svm_qp_config.h.

3.19.2.10 H

```
abip_float* SVMqp::H
```

Definition at line 52 of file svm_qp_config.h.

3.19.2.11 init_spe_linsys_work

```
abip_int(* SVMqp::init_spe_linsys_work) (svmqp *self)
```

Definition at line 39 of file svm_qp_config.h.

3.19.2.12 inner_conv_check

```
abip_float(* SVMqp::inner_conv_check) (svmqp *self, ABIPWork *w)
```

Definition at line 44 of file svm_qp_config.h.

3.19.2.13 L

```
ABIPLinSysWork* SVMqp::L
```

Definition at line 22 of file svm_qp_config.h.

3.19.2.14 lambda

```
abip_float SVMqp::lambda
```

Definition at line 47 of file svm_qp_config.h.

3.19.2.15 m

```
abip_int SVMqp::m
```

Definition at line 18 of file svm_qp_config.h.

3.19.2.16 n

```
abip_int SVMqp::n
```

Definition at line 19 of file svm_qp_config.h.

3.19.2.17 p

```
abip_int SVMqp::p
```

Definition at line 20 of file svm_qp_config.h.

3.19.2.18 pro_type

```
enum problem_type SVMqp::pro_type
```

Definition at line 17 of file svm_qp_config.h.

3.19.2.19 q

```
abip_int SVMqp::q
```

Definition at line 21 of file svm_qp_config.h.

3.19.2.20 Q

```
ABIPMatrix* SVMqp::Q
```

Definition at line 31 of file svm_qp_config.h.

3.19.2.21 rho_dr

```
abip_float* SVMqp::rho_dr
```

Definition at line 27 of file svm_qp_config.h.

3.19.2.22 sc_b

`abip_float SVMqp::sc_b`

Definition at line 53 of file svm_qp_config.h.

3.19.2.23 sc_c

`abip_float SVMqp::sc_c`

Definition at line 54 of file svm_qp_config.h.

3.19.2.24 scaling_data

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

Definition at line 36 of file svm_qp_config.h.

3.19.2.25 solve_spe_linsys

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

Definition at line 40 of file svm_qp_config.h.

3.19.2.26 sparsity

`abip_float SVMqp::sparsity`

Definition at line 25 of file svm_qp_config.h.

3.19.2.27 spe_A_times

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

Definition at line 42 of file svm_qp_config.h.

3.19.2.28 spe_AT_times

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

Definition at line 43 of file svm_qp_config.h.

3.19.2.29 stgs

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

Definition at line 23 of file svm_qp_config.h.

3.19.2.30 un_scaling_sol

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

Definition at line 37 of file svm_qp_config.h.

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

- include/svm_qp_config.h

Chapter 4

File Documentation

4.1 amd/amd.h File Reference

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

Macros

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

Functions

- int amd_order (int n, const int Ap[ ], const int Ai[ ], int P[ ], abip_float Control[ ], abip_float ABIPInfo[ ])
- SuiteSparse_long amd_l_order (SuiteSparse_long n, const SuiteSparse_long Ap[ ], const SuiteSparse_long Ai[ ], SuiteSparse_long P[ ], abip_float Control[ ], abip_float ABIPInfo[ ])
- void amd_2 (int n, int Pe[ ], int 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[ ])
- 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[ ])
- int amd_valid (int n_row, int n_col, const int Ap[ ], const int Ai[ ])
- SuiteSparse_long amd_l_valid (SuiteSparse_long n_row, SuiteSparse_long n_col, const SuiteSparse_long Ap[ ], const SuiteSparse_long Ai[ ])
- void amd_defaults (abip_float Control[ ])
- void amd_l_defaults (abip_float Control[ ])
- void amd_control (abip_float Control[ ])
- void amd_l_control (abip_float Control[ ])
- void amd_info (abip_float ABIPInfo[ ])
- void amd_l_info (abip_float ABIPInfo[ ])

Variables

- EXTERN void ∗(∗ amd_malloc)(size_t)
- EXTERN void(∗ amd_free)(void ∗)
- EXTERN void ∗(∗ amd_realloc)(void ∗, size_t)
- EXTERN void ∗(∗ amd_calloc)(size_t, size_t)
- EXTERN int(∗ amd_printf)(const char ∗,...)

4.1.1 Macro Definition Documentation

4.1.1.1 AMD_AGGRESSIVE

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

Definition at line 341 of file amd.h.

4.1.1.2 AMD_CONTROL

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

Definition at line 336 of file amd.h.

4.1.1.3 AMD_DATE

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

Definition at line 394 of file amd.h.

4.1.1.4 AMD_DEFAULT_AGGRESSIVE

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

Definition at line 345 of file amd.h.

4.1.1.5 AMD_DEFAULT_DENSE

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

Definition at line 344 of file amd.h.

4.1.1.6 AMD_DENSE

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

Definition at line 340 of file amd.h.

4.1.1.7 AMD_DMAX

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

Definition at line 361 of file amd.h.

4.1.1.8 AMD_INFO

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

Definition at line 337 of file amd.h.

4.1.1.9 AMD_INVALID

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

Definition at line 369 of file amd.h.

4.1.1.10 AMD_LNZ

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

Definition at line 357 of file amd.h.

4.1.1.11 AMD_MAIN_VERSION

```
#define AMD_MAIN_VERSION 2
```

Definition at line 396 of file amd.h.

4.1.1.12 AMD_MEMORY

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

Definition at line 355 of file amd.h.

4.1.1.13 AMD_N

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

Definition at line 349 of file amd.h.

4.1.1.14 AMD_NCMPA

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

Definition at line 356 of file amd.h.

4.1.1.15 AMD_NDENSE

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

Definition at line 354 of file amd.h.

4.1.1.16 AMD_NDIV

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

Definition at line 358 of file amd.h.

4.1.1.17 AMD_NMULTSUBS_LDL

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

Definition at line 359 of file amd.h.

4.1.1.18 AMD_NMULTSUBS_LU

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

Definition at line 360 of file amd.h.

4.1.1.19 AMD_NZ

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

Definition at line 350 of file amd.h.

4.1.1.20 AMD_NZ_A_PLUS_AT

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

Definition at line 353 of file amd.h.

4.1.1.21 AMD_NZDIAG

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

Definition at line 352 of file amd.h.

4.1.1.22 AMD_OK

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

Definition at line 367 of file amd.h.

4.1.1.23 AMD_OK_BUT_JUMBLED

```
#define AMD_OK_BUT_JUMBLED
```

Value:

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

Definition at line 370 of file amd.h.

4.1.1.24 AMD_OUT_OF_MEMORY

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

Definition at line 368 of file amd.h.

4.1.1.25 AMD_STATUS

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

Definition at line 348 of file amd.h.

4.1.1.26 AMD_SUB_VERSION

```
#define AMD_SUB_VERSION 4
```

Definition at line 397 of file amd.h.

4.1.1.27 AMD_SUBSUB_VERSION

```
#define AMD_SUBSUB_VERSION 6
```

Definition at line 398 of file amd.h.

4.1.1.28 AMD_SYMMETRY

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

Definition at line 351 of file amd.h.

4.1.1.29 AMD_VERSION

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

Definition at line 399 of file amd.h.

4.1.1.30 AMD_VERSION_CODE

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

Definition at line 395 of file amd.h.

4.1.1.31 EXTERN

```
#define EXTERN extern
```

Definition at line 311 of file amd.h.

4.1.2 Function Documentation

4.1.2.1 amd_2()

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

4.1.2.2 amd_control()

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

4.1.2.3 amd_defaults()

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

4.1.2.4 amd_info()

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

4.1.2.5 amd_l2()

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

4.1.2.6 amd_l_control()

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

4.1.2.7 amd_l_defaults()

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

4.1.2.8 amd_l_info()

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

4.1.2.9 amd_l_order()

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

4.1.2.10 amd_l_valid()

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

4.1.2.11 amd_order()

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

4.1.2.12 amd_valid()

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

4.1.3 Variable Documentation

4.1.3.1 amd_calloc

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

Definition at line 317 of file amd.h.

4.1.3.2 amd_free

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

Definition at line 315 of file amd.h.

4.1.3.3 amd_malloc

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

Definition at line 314 of file amd.h.

4.1.3.4 amd_printf

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

Definition at line 318 of file amd.h.

4.1.3.5 amd_realloc

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

Definition at line 316 of file amd.h.

4.2 amd.h

Go to the documentation of this file.

```
/* ========================================================================= */
/* === AMD:  approximate minimum degree ordering =========================== */
/* ========================================================================= */

/* ------------------------------------------------------------------------- */
/* AMD Version 2.4, Copyright (c) 1996-2013 by Timothy A. Davis,             */
/* Patrick R. Amestoy, and Iain S. Duff.  See ../README.txt for License.     */
/* email: DrTimothyAldenDavis@gmail.com                                      */
/* ------------------------------------------------------------------------- */

/* AMD finds a symmetric ordering P of a matrix A so that the Cholesky
 * factorization of P*A*P' has fewer nonzeros and takes less work than the
 * Cholesky factorization of A.  If A is not symmetric, then it performs its
 * ordering on the matrix A+A'.  Two sets of user-callable routines are
 * provided, one for int integers and the other for SuiteSparse_long integers.
 *
 * The method is based on the approximate minimum degree algorithm, discussed
 * in Amestoy, Davis, and Duff, "An approximate degree ordering algorithm",
 * SIAM Journal of Matrix Analysis and Applications, vol. 17, no. 4, pp.
 * 886-905, 1996.  This package can perform both the AMD ordering (with
 * aggressive absorption), and the AMDBAR ordering (without aggressive
 * absorption) discussed in the above paper.  This package differs from the
 * Fortran codes discussed in the paper:
 *
 *       (1) it can ignore "dense" rows and columns, leading to faster run times
 *       (2) it computes the ordering of A+A' if A is not symmetric
 *       (3) it is followed by a depth-first post-ordering of the assembly tree
 *           (or supernodal elimination tree)
 *
 * For historical reasons, the Fortran versions, amd.f and amdbar.f, have
 * been left (nearly) unchanged.  They compute the identical ordering as
 * described in the above paper.
 */
```

```
#ifndef AMD_H
#define AMD_H

/* make it easy for C++ programs to include AMD */
#ifdef __cplusplus
extern "C" {
#endif

/* get the definition of size_t: */
#include <stddef.h>

#include "SuiteSparse_config.h"

int amd_order                           /* returns AMD_OK, AMD_OK_BUT_JUMBLED,
                                         * AMD_INVALID, or AMD_OUT_OF_MEMORY */
(
            int n,      /* A is n-by-n.  n must be >= 0. */
            const int Ap [ ],      /* column pointers for A, of size n+1 */
            const int Ai [ ],      /* row indices of A, of size nz = Ap [n] */
            int P [ ],      /* output permutation, of size n */
            abip_float Control [ ],      /* input Control settings, of size AMD_CONTROL */
            abip_float ABIPInfo [ ]      /* output ABIPInfo statistics, of size AMD_INFO */
) ;

SuiteSparse_long amd_l_order    /* see above for description of arguments */
(
            SuiteSparse_long n,
            const SuiteSparse_long Ap [ ],
            const SuiteSparse_long Ai [ ],
            SuiteSparse_long P [ ],
            abip_float Control [ ],
            abip_float ABIPInfo [ ]
) ;

/* Input arguments (not modified):
 *
 *      n: the matrix A is n-by-n.
 *      Ap: an int/SuiteSparse_long array of size n+1, containing column
 *              pointers of A.
 *      Ai: an int/SuiteSparse_long array of size nz, containing the row
 *              indices of A, where nz = Ap [n].
 *      Control:  a double array of size AMD_CONTROL, containing control
 *          parameters.  Defaults are used if Control is ABIP_NULL.
 *
 * Output arguments (not defined on input):
 *
 *      P: an int/SuiteSparse_long array of size n, containing the output
 *          permutation. If row i is the kth pivot row, then P [k] = i.  In
 *          MATLAB notation, the reordered matrix is A (P,P).
 *      ABIPInfo: a double array of size AMD_INFO, containing statistical
 *          information.  Ignored if ABIPInfo is ABIP_NULL.
 *
 * On input, the matrix A is stored in column-oriented form.  The row indices
 * of nonzero entries in column j are stored in Ai [Ap [j] ... Ap [j+1]-1].
 *
 * If the row indices appear in ascending order in each column, and there
 * are no duplicate entries, then amd_order is slightly more efficient in
 * terms of time and memory usage.  If this condition does not hold, a copy
 * of the matrix is created (where these conditions do hold), and the copy is
 * ordered.  This feature is new to v2.0 (v1.2 and earlier required this
 * condition to hold for the input matrix).
 *
 * Row indices must be in the range 0 to
 * n-1.  Ap [0] must be zero, and thus nz = Ap [n] is the number of nonzeros
 * in A.  The array Ap is of size n+1, and the array Ai is of size nz = Ap [n].
 * The matrix does not need to be symmetric, and the diagonal does not need to
 * be present (if diagonal entries are present, they are ignored except for
 * the output statistic Info [AMD_NZDIAG]).  The arrays Ai and Ap are not
 * modified.  This form of the Ap and Ai arrays to represent the nonzero
 * pattern of the matrix A is the same as that used internally by MATLAB.
 * If you wish to use a more flexible input structure, please see the
 * umfpack_*_triplet_to_col routines in the UMFPACK package, at
 * http://www.suitesparse.com.
 *
 * Restrictions:  n >= 0.  Ap [0] = 0.  Ap [j] <= Ap [j+1] for all j in the
 *      range 0 to n-1.  nz = Ap [n] >= 0.  Ai [0..nz-1] must be in the range 0
 *      to n-1.  Finally, Ai, Ap, and P must not be NULL.  If any of these
 *      restrictions are not met, AMD returns AMD_INVALID.
 *
 * AMD returns:
 *
 *      AMD_OK if the matrix is valid and sufficient memory can be allocated to
 *          perform the ordering.
 *
 *      AMD_OUT_OF_MEMORY if not enough memory can be allocated.
 *
```

```
 *       AMD_INVALID if the input arguments n, Ap, Ai are invalid, or if P is
 *           NULL.
 *
 *       AMD_OK_BUT_JUMBLED if the matrix had unsorted columns, and/or duplicate
 *           entries, but was otherwise valid.
 *
 * The AMD routine first forms the pattern of the matrix A+A', and then
 * computes a fill-reducing ordering, P.  If P [k] = i, then row/column i of
 * the original is the kth pivotal row.  In MATLAB notation, the permuted
 * matrix is A (P,P), except that 0-based indexing is used instead of the
 * 1-based indexing in MATLAB.
 *
 * The Control array is used to set various parameters for AMD.  If a NULL
 * pointer is passed, default values are used.  The Control array is not
 * modified.
 *
 *       Control [AMD_DENSE]:  controls the threshold for "dense" rows/columns.
 *           A dense row/column in A+A' can cause AMD to spend a lot of time in
 *           ordering the matrix.  If Control [AMD_DENSE] >= 0, rows/columns
 *           with more than Control [AMD_DENSE] * sqrt (n) entries are ignored
 *           during the ordering, and placed last in the output order.  The
 *           default value of Control [AMD_DENSE] is 10.  If negative, no
 *           rows/columns are treated as "dense".  Rows/columns with 16 or
 *           fewer off-diagonal entries are never considered "dense".
 *
 *       Control [AMD_AGGRESSIVE]: controls whether or not to use aggressive
 *           absorption, in which a prior element is absorbed into the current
 *           element if is a subset of the current element, even if it is not
 *           adjacent to the current pivot element (refer to Amestoy, Davis,
 *           & Duff, 1996, for more details).  The default value is nonzero,
 *           which means to perform aggressive absorption.  This nearly always
 *           leads to a better ordering (because the approximate degrees are
 *           more accurate) and a lower execution time.  There are cases where
 *           it can lead to a slightly worse ordering, however.  To turn it off,
 *           set Control [AMD_AGGRESSIVE] to 0.
 *
 *       Control [2..4] are not used in the current version, but may be used in
 *           future versions.
 *
 * The ABIPInfo array provides statistics about the ordering on output.  If it is
 * not present, the statistics are not returned.  This is not an error
 * condition.
 *
 *       ABIPInfo [AMD_STATUS]:  the return value of AMD, either AMD_OK,
 *           AMD_OK_BUT_JUMBLED, AMD_OUT_OF_MEMORY, or AMD_INVALID.
 *
 *       ABIPInfo [AMD_N]: n, the size of the input matrix
 *
 *       ABIPInfo [AMD_NZ]: the number of nonzeros in A, nz = Ap [n]
 *
 *       ABIPInfo [AMD_SYMMETRY]:  the symmetry of the matrix A.  It is the number
 *           of "matched" off-diagonal entries divided by the total number of
 *           off-diagonal entries.  An entry A(i,j) is matched if A(j,i) is also
 *           an entry, for any pair (i,j) for which i != j.  In MATLAB notation,
 *                S = spones (A) ;
 *                B = tril (S, -1) + triu (S, 1) ;
 *                symmetry = nnz (B & B') / nnz (B) ;
 *
 *       ABIPInfo [AMD_NZDIAG]: the number of entries on the diagonal of A.
 *
 *       ABIPInfo [AMD_NZ_A_PLUS_AT]:  the number of nonzeros in A+A', excluding the
 *           diagonal.  If A is perfectly symmetric (Info [AMD_SYMMETRY] = 1)
 *           with a fully nonzero diagonal, then Info [AMD_NZ_A_PLUS_AT] = nz-n
 *           (the smallest possible value).  If A is perfectly unsymmetric
 *           (Info [AMD_SYMMETRY] = 0, for an upper triangular matrix, for
 *           example) with no diagonal, then Info [AMD_NZ_A_PLUS_AT] = 2*nz
 *           (the largest possible value).
 *
 *       ABIPInfo [AMD_NDENSE]: the number of "dense" rows/columns of A+A' that were
 *           removed from A prior to ordering.  These are placed last in the
 *           output order P.
 *
 *       ABIPInfo [AMD_MEMORY]: the amount of memory used by AMD, in bytes.  In the
 *           current version, this is 1.2 * Info  [AMD_NZ_A_PLUS_AT] + 9*n
 *           times the size of an integer.  This is at most 2.4nz + 9n.  This
 *           excludes the size of the input arguments Ai, Ap, and P, which have
 *           a total size of nz + 2*n + 1 integers.
 *
 *       ABIPInfo [AMD_NCMPA]: the number of garbage collections performed.
 *
 *       ABIPInfo [AMD_LNZ]: the number of nonzeros in L (excluding the diagonal).
 *           This is a slight upper bound because mass elimination is combined
 *           with the approximate degree update.  It is a rough upper bound if
 *           there are many "dense" rows/columns.  The rest of the statistics,
 *           below, are also slight or rough upper bounds, for the same reasons.
 *           The post-ordering of the assembly tree might also not exactly
 *           correspond to a true elimination tree postordering.
```

```
 *
 *      ABIPInfo [AMD_NDIV]: the number of divide operations for a subsequent LDL'
 *          or LU factorization of the permuted matrix A (P,P).
 *
 *      ABIPInfo [AMD_NMULTSUBS_LDL]:  the number of multiply-subtract pairs for a
 *          subsequent LDL' factorization of A (P,P).
 *
 *      ABIPInfo [AMD_NMULTSUBS_LU]:  the number of multiply-subtract pairs for a
 *          subsequent LU factorization of A (P,P), assuming that no numerical
 *          pivoting is required.
 *
 *      ABIPInfo [AMD_DMAX]:  the maximum number of nonzeros in any column of L,
 *          including the diagonal.
 *
 *      ABIPInfo [14..19] are not used in the current version, but may be used in
 *          future versions.
 */

/* ------------------------------------------------------------------------- */
/* direct interface to AMD */
/* ------------------------------------------------------------------------- */

/* amd_2 is the primary AMD ordering routine.  It is not meant to be
 * user-callable because of its restrictive inputs and because it destroys
 * the user's input matrix.  It does not check its inputs for errors, either.
 * However, if you can work with these restrictions it can be faster than
 * amd_order and use less memory (assuming that you can create your own copy
 * of the matrix for AMD to destroy).  Refer to AMD/Source/amd_2.c for a
 * description of each parameter. */

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 [ ]
) ;

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 [ ]
) ;

/* ------------------------------------------------------------------------- */
/* amd_valid */
/* ------------------------------------------------------------------------- */

/* Returns AMD_OK or AMD_OK_BUT_JUMBLED if the matrix is valid as input to
 * amd_order; the latter is returned if the matrix has unsorted and/or
 * duplicate row indices in one or more columns.  Returns AMD_INVALID if the
 * matrix cannot be passed to amd_order.  For amd_order, the matrix must also
 * be square.  The first two arguments are the number of rows and the number
 * of columns of the matrix.  For its use in AMD, these must both equal n.
 *
 * NOTE: this routine returned TRUE/FALSE in v1.2 and earlier.
 */

int amd_valid
(
            int n_row,                  /* # of rows */
            int n_col,                  /* # of columns */
            const int Ap [ ],           /* column pointers, of size n_col+1 */
```

```
            const int Ai [ ]            /* row indices, of size Ap [n_col] */
) ;

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

/* ------------------------------------------------------------------------- */
/* AMD memory manager and printf routines */
/* ------------------------------------------------------------------------- */

#ifndef EXTERN
#define EXTERN extern
#endif

EXTERN void *(*amd_malloc) (size_t) ;                   /* pointer to malloc */
EXTERN void (*amd_free) (void *) ;                         /* pointer to free */
EXTERN void *(*amd_realloc) (void *, size_t) ;          /* pointer to realloc */
EXTERN void *(*amd_calloc) (size_t, size_t) ;           /* pointer to calloc */
EXTERN int (*amd_printf) (const char *, ...) ;            /* pointer to printf */

/* ------------------------------------------------------------------------- */
/* AMD Control and Info arrays */
/* ------------------------------------------------------------------------- */

/* amd_defaults:  sets the default control settings */
void amd_defaults   (abip_float Control [ ]) ;
void amd_l_defaults (abip_float Control [ ]) ;

/* amd_control: prints the control settings */
void amd_control    (abip_float Control [ ]) ;
void amd_l_control  (abip_float Control [ ]) ;

/* amd_info: prints the statistics */
void amd_info       (abip_float ABIPInfo [ ]) ;
void amd_l_info     (abip_float ABIPInfo [ ]) ;

#define AMD_CONTROL 5          /* size of Control array */
#define AMD_INFO 20            /* size of Info array */

/* contents of Control */
#define AMD_DENSE 0            /* "dense" if degree > Control [0] * sqrt (n) */
#define AMD_AGGRESSIVE 1    /* do aggressive absorption if Control [1] != 0 */

/* default Control settings */
#define AMD_DEFAULT_DENSE 10.0          /* default "dense" degree 10*sqrt(n) */
#define AMD_DEFAULT_AGGRESSIVE 1    /* do aggressive absorption by default */

/* contents of Info */
#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 */

/* ------------------------------------------------------------------------- */
/* return values of AMD */
/* ------------------------------------------------------------------------- */

#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 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.  */

/* ========================================================================== */
/* === AMD version ========================================================== */
/* ========================================================================== */

/* AMD Version 1.2 and later include the following definitions.
 * As an example, to test if the version you are using is 1.2 or later:
 *
```

```
 * #ifdef AMD_VERSION
 *       if (AMD_VERSION >= AMD_VERSION_CODE (1,2)) ...
 * #endif
 *
 * This also works during compile-time:
 *
 *       #if defined(AMD_VERSION) && (AMD_VERSION >= AMD_VERSION_CODE (1,2))
 *           printf ("This is version 1.2 or later\n") ;
 *       #else
 *           printf ("This is an early version\n") ;
 *       #endif
 *
 * Versions 1.1 and earlier of AMD do not include a #define'd version number.
 */

#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)

#ifdef __cplusplus
}
#endif

#endif
```

4.3 amd/amd_1.c File Reference

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

Functions

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

4.3.1 Function Documentation

4.3.1.1 AMD_1()

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

Definition at line 29 of file amd_1.c.

4.4 amd_1.c

Go to the documentation of this file.

```
/* ========================================================================= */
/* === AMD_1 =============================================================== */
/* ========================================================================= */

/* ------------------------------------------------------------------------- */
/* AMD, Copyright (c) Timothy A. Davis,                                     */
/* Patrick R. Amestoy, and Iain S. Duff.  See ../README.txt for License.     */
/* email: DrTimothyAldenDavis@gmail.com                                      */
/* ------------------------------------------------------------------------- */

/* AMD_1: Construct A+A' for a sparse matrix A and perform the AMD ordering.
 *
 * The n-by-n sparse matrix A can be unsymmetric.  It is stored in MATLAB-style
 * compressed-column form, with sorted row indices in each column, and no
 * duplicate entries.  Diagonal entries may be present, but they are ignored.
 * Row indices of column j of A are stored in Ai [Ap [j] ... Ap [j+1]-1].
 * Ap [0] must be zero, and nz = Ap [n] is the number of entries in A.  The
 * size of the matrix, n, must be greater than or equal to zero.
 *
 * This routine must be preceded by a call to AMD_aat, which computes the
 * number of entries in each row/column in A+A', excluding the diagonal.
 * Len [j], on input, is the number of entries in row/column j of A+A'.  This
 * routine constructs the matrix A+A' and then calls AMD_2.  No error checking
 * is performed (this was done in AMD_valid).
 */

#include "amd_internal.h"

GLOBAL void AMD_1
(
        Int n,            /* n > 0 */
        const Int Ap [ ],     /* input of size n+1, not modified */
        const Int Ai [ ],         /* input of size nz = Ap [n], not modified */
        Int P [ ],        /* size n output permutation */
        Int Pinv [ ],         /* size n output inverse permutation */
        Int Len [ ],      /* size n input, undefined on output */
        Int slen,         /* slen >= sum (Len [0..n-1]) + 7n,
                 * ideally slen = 1.2 * sum (Len) + 8n */
        Int S [ ],        /* size slen workspace */
        abip_float Control [ ],    /* input array of size AMD_CONTROL */
        abip_float ABIPInfo [ ]    /* output array of size AMD_INFO */
)
{
        Int i;
        Int j;
        Int k;
        Int p;
        Int pfree;
        Int iwlen;
        Int pj;
        Int p1;
        Int p2;
        Int pj2;

        Int *Iw;
        Int *Pe;
        Int *Nv;
        Int *Head;
    Int *Elen;
    Int *Degree;
    Int *s;
    Int *W;
    Int *Sp;
    Int *Tp;

        /* --------------------------------------------------------------------- */
        /* construct the matrix for AMD_2 */
        /* --------------------------------------------------------------------- */

        ASSERT (n > 0) ;

        iwlen = slen - 6*n ;
        s = S ;
        Pe = s ;
        s += n ;
        Nv = s ;
        s += n ;
        Head = s ;
        s += n ;
        Elen = s ;
        s += n ;
        Degree = s ;
```

```
    s += n ;
    W = s ;
    s += n ;
    Iw = s ;
    s += iwlen ;

    ASSERT (AMD_valid (n, n, Ap, Ai) == AMD_OK) ;

    /* construct the pointers for A+A' */
    Sp = Nv ;           /* use Nv and W as workspace for Sp and Tp [ */
    Tp = W ;
    pfree = 0 ;

    for (j = 0 ; j < n ; j++)
    {
    Pe [j] = pfree ;
    Sp [j] = pfree ;
    pfree += Len [j] ;
    }

    /* Note that this restriction on iwlen is slightly more restrictive than
    * what is strictly required in AMD_2.  AMD_2 can operate with no elbow
    * room at all, but it will be very slow.  For better performance, at
    * least size-n elbow room is enforced. */
    ASSERT (iwlen >= pfree + n) ;

#ifndef NDEBUG
    for (p = 0 ; p < iwlen ; p++) Iw [p] = EMPTY ;
#endif

    for (k = 0 ; k < n ; k++)
    {
    AMD_DEBUG1 (("Construct row/column k= "ID" of A+A'\n", k))  ;
    p1 = Ap [k] ;
    p2 = Ap [k+1] ;

    /* construct A+A' */
    for (p = p1 ; p < p2 ; )
    {
            /* scan the upper triangular part of A */
            j = Ai [p] ;
            ASSERT (j >= 0 && j < n) ;

            if (j < k)
            {
            /* entry A (j,k) in the strictly upper triangular part */
            ASSERT (Sp [j] < (j == n-1 ? pfree : Pe [j+1])) ;
            ASSERT (Sp [k] < (k == n-1 ? pfree : Pe [k+1])) ;
            Iw [Sp [j]++] = k ;
            Iw [Sp [k]++] = j ;
            p++ ;
            }
            else if (j == k)
            {
            /* skip the diagonal */
            p++ ;
            break ;
            }
            else /* j > k */
            {
            /* first entry below the diagonal */
            break ;
            }

            /* scan lower triangular part of A, in column j until reaching
            * row k.  Start where last scan left off. */
            ASSERT (Ap [j] <= Tp [j] && Tp [j] <= Ap [j+1]) ;
            pj2 = Ap [j+1] ;

            for (pj = Tp [j] ; pj < pj2 ; )
            {
            i = Ai [pj] ;
            ASSERT (i >= 0 && i < n) ;

            if (i < k)
            {
                    /* A (i,j) is only in the lower part, not in upper */
                    ASSERT (Sp [i] < (i == n-1 ? pfree : Pe [i+1])) ;
                    ASSERT (Sp [j] < (j == n-1 ? pfree : Pe [j+1])) ;
                    Iw [Sp [i]++] = j ;
                    Iw [Sp [j]++] = i ;
                    pj++ ;
            }
            else if (i == k)
            {
                    /* entry A (k,j) in lower part and A (j,k) in upper */
                    pj++ ;
```

```
                    break ;
                }
                else /* i > k */
                {
                    /* consider this entry later, when k advances to i */
                    break ;
                }
            }
            Tp [j] = pj ;
        }
        Tp [k] = p ;
    }

    /* clean up, for remaining mismatched entries */
    for (j = 0 ; j < n ; j++)
    {
        for (pj = Tp [j] ; pj < Ap [j+1] ; pj++)
        {
            i = Ai [pj] ;
            ASSERT (i >= 0 && i < n) ;

            /* A (i,j) is only in the lower part, not in upper */
            ASSERT (Sp [i] < (i == n-1 ? pfree : Pe [i+1])) ;
            ASSERT (Sp [j] < (j == n-1 ? pfree : Pe [j+1])) ;
            Iw [Sp [i]++] = j ;
            Iw [Sp [j]++] = i ;
        }
    }

#ifndef NDEBUG

    for (j = 0 ; j < n-1 ; j++) ASSERT (Sp [j] == Pe [j+1]) ;
    ASSERT (Sp [n-1] == pfree) ;

#endif

    /* Tp and Sp no longer needed ] */

    /* --------------------------------------------------------------------- */
    /* order the matrix */
    /* --------------------------------------------------------------------- */

    AMD_2 (n, Pe, Iw, Len, iwlen, pfree, Nv, Pinv, P, Head, Elen, Degree, W, Control, ABIPInfo) ;
}
```

4.5 amd/amd_2.c File Reference

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

Functions

- 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[ ])

4.5.1 Function Documentation

4.5.1.1 AMD_2()

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

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

Definition at line 43 of file amd_2.c.

4.6 amd_2.c

Go to the documentation of this file.

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

```
           Int Elen [ ],                /* the size columns of L for each supernode */
           Int Degree [ ],
           Int W [ ],

           /* control parameters and output statistics */
           abip_float Control [ ],   /* array of size AMD_CONTROL */
           abip_float ABIPInfo [ ]   /* array of size AMD_INFO */
)
{

/*
 * Given a representation of the nonzero pattern of a symmetric matrix, A,
 * (excluding the diagonal) perform an approximate minimum (UMFPACK/MA38-style)
 * degree ordering to compute a pivot order such that the introduction of
 * nonzeros (fill-in) in the Cholesky factors A = LL' is kept low.  At each
 * step, the pivot selected is the one with the minimum UMFAPACK/MA38-style
 * upper-bound on the external degree.  This routine can optionally perform
 * aggresive absorption (as done by MC47B in the Harwell Subroutine
 * Library).
 *
 * The approximate degree algorithm implemented here is the symmetric analog of
 * the degree update algorithm in MA38 and UMFPACK (the Unsymmetric-pattern
 * MultiFrontal PACKage, both by Davis and Duff).  The routine is based on the
 * MA27 minimum degree ordering algorithm by Iain Duff and John Reid.
 *
 * This routine is a translation of the original AMDBAR and MC47B routines,
 * in Fortran, with the following modifications:
 *
 * (1) dense rows/columns are removed prior to ordering the matrix, and placed
 *  last in the output order.  The presence of a dense row/column can
 *  increase the ordering time by up to O(n^2), unless they are removed
 *  prior to ordering.
 *
 * (2) the minimum degree ordering is followed by a postordering (depth-first
 *  search) of the assembly tree.  Note that mass elimination (discussed
 *  below) combined with the approximate degree update can lead to the mass
 *  elimination of nodes with lower exact degree than the current pivot
 *  element.  No additional fill-in is caused in the representation of the
 *  Schur complement.  The mass-eliminated nodes merge with the current
 *  pivot element.  They are ordered prior to the current pivot element.
 *  Because they can have lower exact degree than the current element, the
 *  merger of two or more of these nodes in the current pivot element can
 *  lead to a single element that is not a "fundamental supernode".  The
 *  diagonal block can have zeros in it.  Thus, the assembly tree used here
 *  is not guaranteed to be the precise supernodal elemination tree (with
 *  "funadmental" supernodes), and the postordering performed by this
 *  routine is not guaranteed to be a precise postordering of the
 *  elimination tree.
 *
 * (3) input parameters are added, to control aggressive absorption and the
 *  detection of "dense" rows/columns of A.
 *
 * (4) additional statistical information is returned, such as the number of
 *  nonzeros in L, and the flop counts for subsequent LDL' and LU
 *  factorizations.  These are slight upper bounds, because of the mass
 *  elimination issue discussed above.
 *
 * (5) additional routines are added to interface this routine to MATLAB
 *  to provide a simple C-callable user-interface, to check inputs for
 *  errors, compute the symmetry of the pattern of A and the number of
 *  nonzeros in each row/column of A+A', to compute the pattern of A+A',
 *  to perform the assembly tree postordering, and to provide debugging
 *  ouput.  Many of these functions are also provided by the Fortran
 *  Harwell Subroutine Library routine MC47A.
 *
 * (6) both int and SuiteSparse_long versions are provided.  In the
 *      descriptions below and integer is and int or SuiteSparse_long depending
 *      on which version is being used.

 **********************************************************************
 ***** CAUTION:  ARGUMENTS ARE NOT CHECKED FOR ERRORS ON INPUT.  ******
 **********************************************************************
 ** If you want error checking, a more versatile input format, and a **
 ** simpler user interface, use amd_order or amd_l_order instead.    **
 ** This routine is not meant to be user-callable.                   **
 **********************************************************************

 * ------------------------------------------------------------------------
 * References:
 * ------------------------------------------------------------------------
 *
 *  [1] Timothy A. Davis and Iain Duff, "An unsymmetric-pattern multifrontal
 *  method for sparse LU factorization", SIAM J. Matrix Analysis and
 *  Applications, vol. 18, no. 1, pp. 140-158.  Discusses UMFPACK / MA38,
 *  which first introduced the approximate minimum degree used by this
 *  routine.
 *
```

```
 *  [2] Patrick Amestoy, Timothy A. Davis, and Iain S. Duff, "An approximate
 *  minimum degree ordering algorithm," SIAM J. Matrix Analysis and
 *  Applications, vol. 17, no. 4, pp. 886-905, 1996.  Discusses AMDBAR and
 *  MC47B, which are the Fortran versions of this routine.
 *
 *  [3] Alan George and Joseph Liu, "The evolution of the minimum degree
 *  ordering algorithm," SIAM Review, vol. 31, no. 1, pp. 1-19, 1989.
 *  We list below the features mentioned in that paper that this code
 *  includes:
 *
 *  mass elimination:
 *      Yes.  MA27 relied on supervariable detection for mass elimination.
 *
 *  indistinguishable nodes:
 *      Yes (we call these "supervariables").  This was also in the MA27
 *      code - although we modified the method of detecting them (the
 *      previous hash was the true degree, which we no longer keep track
 *      of).  A supervariable is a set of rows with identical nonzero
 *      pattern.  All variables in a supervariable are eliminated together.
 *      Each supervariable has as its numerical name that of one of its
 *      variables (its principal variable).
 *
 *  quotient graph representation:
 *      Yes.  We use the term "element" for the cliques formed during
 *      elimination.  This was also in the MA27 code.  The algorithm can
 *      operate in place, but it will work more efficiently if given some
 *      "elbow room."
 *
 *  element absorption:
 *      Yes.  This was also in the MA27 code.
 *
 *  external degree:
 *      Yes.  The MA27 code was based on the true degree.
 *
 *  incomplete degree update and multiple elimination:
 *      No.  This was not in MA27, either.  Our method of degree update
 *      within MC47B is element-based, not variable-based.  It is thus
 *      not well-suited for use with incomplete degree update or multiple
 *      elimination.
 *
 * Authors, and Copyright (C) 2004 by:
 * Timothy A. Davis, Patrick Amestoy, Iain S. Duff, John K. Reid.
 *
 * Acknowledgements: This work (and the UMFPACK package) was supported by the
 * National Science Foundation (ASC-9111263, DMS-9223088, and CCR-0203270).
 * The UMFPACK/MA38 approximate degree update algorithm, the unsymmetric analog
 * which forms the basis of AMD, was developed while Tim Davis was supported by
 * CERFACS (Toulouse, France) in a post-doctoral position.  This C version, and
 * the etree postorder, were written while Tim Davis was on sabbatical at
 * Stanford University and Lawrence Berkeley National Laboratory.

 * ----------------------------------------------------------------------------
 * INPUT ARGUMENTS (unaltered):
 * ----------------------------------------------------------------------------

 * n:  The matrix order.  Restriction:  n >= 1.
 *
 * iwlen:  The size of the Iw array.  On input, the matrix is stored in
 *  Iw [0..pfree-1].  However, Iw [0..iwlen-1] should be slightly larger
 *  than what is required to hold the matrix, at least iwlen >= pfree + n.
 *  Otherwise, excessive compressions will take place.  The recommended
 *  value of iwlen is 1.2 * pfree + n, which is the value used in the
 *  user-callable interface to this routine (amd_order.c).  The algorithm
 *  will not run at all if iwlen < pfree.  Restriction: iwlen >= pfree + n.
 *  Note that this is slightly more restrictive than the actual minimum
 *  (iwlen >= pfree), but AMD_2 will be very slow with no elbow room.
 *  Thus, this routine enforces a bare minimum elbow room of size n.
 *
 * pfree: On input the tail end of the array, Iw [pfree..iwlen-1], is empty,
 *  and the matrix is stored in Iw [0..pfree-1].  During execution,
 *  additional data is placed in Iw, and pfree is modified so that
 *  Iw [pfree..iwlen-1] is always the unused part of Iw.
 *
 * Control:  A abip_float array of size AMD_CONTROL containing input parameters
 *  that affect how the ordering is computed.  If ABIP_NULL, then default
 *  settings are used.
 *
 *  Control [AMD_DENSE] is used to determine whether or not a given input
 *  row is "dense".  A row is "dense" if the number of entries in the row
 *  exceeds Control [AMD_DENSE] times sqrt (n), except that rows with 16 or
 *  fewer entries are never considered "dense".  To turn off the detection
 *  of dense rows, set Control [AMD_DENSE] to a negative number, or to a
 *  number larger than sqrt (n).  The default value of Control [AMD_DENSE]
 *  is AMD_DEFAULT_DENSE, which is defined in amd.h as 10.
 *
 *  Control [AMD_AGGRESSIVE] is used to determine whether or not aggressive
 *  absorption is to be performed.  If nonzero, then aggressive absorption
```

```
 *  is performed (this is the default).

 * ---------------------------------------------------------------------------
 * INPUT/OUPUT ARGUMENTS:
 * ---------------------------------------------------------------------------
 *
 * Pe:  An integer array of size n.  On input, Pe [i] is the index in Iw of
 *  the start of row i.  Pe [i] is ignored if row i has no off-diagonal
 *  entries.  Thus Pe [i] must be in the range 0 to pfree-1 for non-empty
 *  rows.
 *
 *  During execution, it is used for both supervariables and elements:
 *
 *  Principal supervariable i:  index into Iw of the description of
 *      supervariable i.  A supervariable represents one or more rows of
 *      the matrix with identical nonzero pattern.  In this case,
 *      Pe [i] >= 0.
 *
 *  Non-principal supervariable i:  if i has been absorbed into another
 *      supervariable j, then Pe [i] = FLIP (j), where FLIP (j) is defined
 *      as (-(j)-2).  Row j has the same pattern as row i.  Note that j
 *      might later be absorbed into another supervariable j2, in which
 *      case Pe [i] is still FLIP (j), and Pe [j] = FLIP (j2) which is
 *      < EMPTY, where EMPTY is defined as (-1) in amd_internal.h.
 *
 *  Unabsorbed element e:  the index into Iw of the description of element
 *      e, if e has not yet been absorbed by a subsequent element.  Element
 *      e is created when the supervariable of the same name is selected as
 *      the pivot.  In this case, Pe [i] >= 0.
 *
 *  Absorbed element e:  if element e is absorbed into element e2, then
 *      Pe [e] = FLIP (e2).  This occurs when the pattern of e (which we
 *      refer to as Le) is found to be a subset of the pattern of e2 (that
 *      is, Le2).  In this case, Pe [i] < EMPTY.  If element e is "null"
 *      (it has no nonzeros outside its pivot block), then Pe [e] = EMPTY,
 *      and e is the root of an assembly subtree (or the whole tree if
 *      there is just one such root).
 *
 *  Dense variable i:  if i is "dense", then Pe [i] = EMPTY.
 *
 *  On output, Pe holds the assembly tree/forest, which implicitly
 *  represents a pivot order with identical fill-in as the actual order
 *  (via a depth-first search of the tree), as follows.  If Nv [i] > 0,
 *  then i represents a node in the assembly tree, and the parent of i is
 *  Pe [i], or EMPTY if i is a root.  If Nv [i] = 0, then (i, Pe [i])
 *  represents an edge in a subtree, the root of which is a node in the
 *  assembly tree.  Note that i refers to a row/column in the original
 *  matrix, not the permuted matrix.
 *
 * ABIPInfo:  A abip_float array of size AMD_INFO.  If present, (that is, not ABIP_NULL),
 *  then statistics about the ordering are returned in the ABIPInfo array.
 *  See amd.h for a description.

 * ---------------------------------------------------------------------------
 * INPUT/MODIFIED (undefined on output):
 * ---------------------------------------------------------------------------
 *
 * Len:  An integer array of size n.  On input, Len [i] holds the number of
 *  entries in row i of the matrix, excluding the diagonal.  The contents
 *  of Len are undefined on output.
 *
 * Iw:  An integer array of size iwlen.  On input, Iw [0..pfree-1] holds the
 *  description of each row i in the matrix.  The matrix must be symmetric,
 *  and both upper and lower triangular parts must be present.  The
 *  diagonal must not be present.  Row i is held as follows:
 *
 *      Len [i]:  the length of the row i data structure in the Iw array.
 *      Iw [Pe [i] ... Pe [i] + Len [i] - 1]:
 *      the list of column indices for nonzeros in row i (simple
 *      supervariables), excluding the diagonal.  All supervariables
 *      start with one row/column each (supervariable i is just row i).
 *      If Len [i] is zero on input, then Pe [i] is ignored on input.
 *
 *      Note that the rows need not be in any particular order, and there
 *      may be empty space between the rows.
 *
 *  During execution, the supervariable i experiences fill-in.  This is
 *  represented by placing in i a list of the elements that cause fill-in
 *  in supervariable i:
 *
 *      Len [i]:  the length of supervariable i in the Iw array.
 *      Iw [Pe [i] ... Pe [i] + Elen [i] - 1]:
 *      the list of elements that contain i.  This list is kept short
 *      by removing absorbed elements.
 *      Iw [Pe [i] + Elen [i] ... Pe [i] + Len [i] - 1]:
 *      the list of supervariables in i.  This list is kept short by
 *      removing nonprincipal variables, and any entry j that is also
```

```
 *      contained in at least one of the elements (j in Le) in the list
 *      for i (e in row i).
 *
 *  When supervariable i is selected as pivot, we create an element e of
 *  the same name (e=i):
 *
 *      Len [e]:  the length of element e in the Iw array.
 *      Iw [Pe [e] ... Pe [e] + Len [e] - 1]:
 *      the list of supervariables in element e.
 *
 *  An element represents the fill-in that occurs when supervariable i is
 *  selected as pivot (which represents the selection of row i and all
 *  non-principal variables whose principal variable is i).  We use the
 *  term Le to denote the set of all supervariables in element e.  Absorbed
 *  supervariables and elements are pruned from these lists when
 *  computationally convenient.
 *
 *  CAUTION:  THE INPUT MATRIX IS OVERWRITTEN DURING COMPUTATION.
 *  The contents of Iw are undefined on output.

 * ----------------------------------------------------------------------------
 * OUTPUT (need not be set on input):
 * ----------------------------------------------------------------------------
 *
 * Nv:  An integer array of size n.  During execution, ABS (Nv [i]) is equal to
 *  the number of rows that are represented by the principal supervariable
 *  i.  If i is a nonprincipal or dense variable, then Nv [i] = 0.
 *  Initially, Nv [i] = 1 for all i.  Nv [i] < 0 signifies that i is a
 *  principal variable in the pattern Lme of the current pivot element me.
 *  After element me is constructed, Nv [i] is set back to a positive
 *  value.
 *
 *  On output, Nv [i] holds the number of pivots represented by super
 *  row/column i of the original matrix, or Nv [i] = 0 for non-principal
 *  rows/columns.  Note that i refers to a row/column in the original
 *  matrix, not the permuted matrix.
 *
 * Elen:  An integer array of size n.  See the description of Iw above.  At the
 *  start of execution, Elen [i] is set to zero for all rows i.  During
 *  execution, Elen [i] is the number of elements in the list for
 *  supervariable i.  When e becomes an element, Elen [e] = FLIP (esize) is
 *  set, where esize is the size of the element (the number of pivots, plus
 *  the number of nonpivotal entries).  Thus Elen [e] < EMPTY.
 *  Elen (i) = EMPTY set when variable i becomes nonprincipal.
 *
 *  For variables, Elen (i) >= EMPTY holds until just before the
 *  postordering and permutation vectors are computed.  For elements,
 *  Elen [e] < EMPTY holds.
 *
 *  On output, Elen [i] is the degree of the row/column in the Cholesky
 *  factorization of the permuted matrix, corresponding to the original row
 *  i, if i is a super row/column.  It is equal to EMPTY if i is
 *  non-principal.  Note that i refers to a row/column in the original
 *  matrix, not the permuted matrix.
 *
 *  Note that the contents of Elen on output differ from the Fortran
 *  version (Elen holds the inverse permutation in the Fortran version,
 *  which is instead returned in the Next array in this C version,
 *  described below).
 *
 * Last: In a degree list, Last [i] is the supervariable preceding i, or EMPTY
 *  if i is the head of the list.  In a hash bucket, Last [i] is the hash
 *  key for i.
 *
 *  Last [Head [hash]] is also used as the head of a hash bucket if
 *  Head [hash] contains a degree list (see the description of Head,
 *  below).
 *
 *  On output, Last [0..n-1] holds the permutation.  That is, if
 *  i = Last [k], then row i is the kth pivot row (where k ranges from 0 to
 *  n-1).  Row Last [k] of A is the kth row in the permuted matrix, PAP'.
 *
 * Next: Next [i] is the supervariable following i in a link list, or EMPTY if
 *  i is the last in the list.  Used for two kinds of lists:  degree lists
 *  and hash buckets (a supervariable can be in only one kind of list at a
 *  time).
 *
 *  On output Next [0..n-1] holds the inverse permutation.  That is, if
 *  k = Next [i], then row i is the kth pivot row. Row i of A appears as
 *  the (Next[i])-th row in the permuted matrix, PAP'.
 *
 *  Note that the contents of Next on output differ from the Fortran
 *  version (Next is undefined on output in the Fortran version).

 * ----------------------------------------------------------------------------
 * LOCAL WORKSPACE (not input or output - used only during execution):
 * ----------------------------------------------------------------------------
```

```
 *
 * Degree:  An integer array of size n.  If i is a supervariable, then
 *  Degree [i] holds the current approximation of the external degree of
 *  row i (an upper bound).  The external degree is the number of nonzeros
 *  in row i, minus ABS (Nv [i]), the diagonal part.  The bound is equal to
 *  the exact external degree if Elen [i] is less than or equal to two.
 *
 *  We also use the term "external degree" for elements e to refer to
 *  |Le \ Lme|.  If e is an element, then Degree [e] is |Le|, which is the
 *  degree of the off-diagonal part of the element e (not including the
 *  diagonal part).
 *
 * Head:   An integer array of size n.  Head is used for degree lists.
 *  Head [deg] is the first supervariable in a degree list.  All
 *  supervariables i in a degree list Head [deg] have the same approximate
 *  degree, namely, deg = Degree [i].  If the list Head [deg] is empty then
 *  Head [deg] = EMPTY.
 *
 *  During supervariable detection Head [hash] also serves as a pointer to
 *  a hash bucket.  If Head [hash] >= 0, there is a degree list of degree
 *  hash.  The hash bucket head pointer is Last [Head [hash]].  If
 *  Head [hash] = EMPTY, then the degree list and hash bucket are both
 *  empty.  If Head [hash] < EMPTY, then the degree list is empty, and
 *  FLIP (Head [hash]) is the head of the hash bucket.  After supervariable
 *  detection is complete, all hash buckets are empty, and the
 *  (Last [Head [hash]] = EMPTY) condition is restored for the non-empty
 *  degree lists.
 *
 * W:  An integer array of size n.  The flag array W determines the status of
 *  elements and variables, and the external degree of elements.
 *
 *  for elements:
 *      if W [e] = 0, then the element e is absorbed.
 *      if W [e] >= wflg, then W [e] - wflg is the size of the set
 *      |Le \ Lme|, in terms of nonzeros (the sum of ABS (Nv [i]) for
 *      each principal variable i that is both in the pattern of
 *      element e and NOT in the pattern of the current pivot element,
 *      me).
 *      if wflg > W [e] > 0, then e is not absorbed and has not yet been
 *      seen in the scan of the element lists in the computation of
 *      |Le\Lme| in Scan 1 below.
 *
 *  for variables:
 *      during supervariable detection, if W [j] != wflg then j is
 *      not in the pattern of variable i.
 *
 *  The W array is initialized by setting W [i] = 1 for all i, and by
 *  setting wflg = 2.  It is reinitialized if wflg becomes too large (to
 *  ensure that wflg+n does not cause integer overflow).

 * ----------------------------------------------------------------------------
 * LOCAL INTEGERS:
 * ----------------------------------------------------------------------------
 */

          Int deg;
          Int degme;
          Int dext;
          Int lemax;

          Int e;
          Int elenme;
          Int eln;

          Int i;
          Int ilast;
          Int inext;

          Int j;
          Int jlast;
          Int jnext;

          Int k;
          Int knt1;
          Int knt2;
          Int knt3;

          Int lenj;
          Int ln;
          Int me;
          Int mindeg;
          Int nel;
          Int nleft;
          Int nvi;
          Int nvj;
          Int nvpiv;
          Int slenme;
```

```
            Int wbig;
            Int we;
            Int wflg;
            Int wnvi;

            Int ok;
            Int ndense;
            Int ncmpa;
            Int dense;
            Int aggressive;

            unsigned Int hash ;      /* unsigned, so that hash % n is well defined.*/

/*
 * deg:      the degree of a variable or element
 * degme:    size, |Lme|, of the current element, me (= Degree [me])
 * dext:     external degree, |Le \ Lme|, of some element e
 * lemax:    largest |Le| seen so far (called dmax in Fortran version)
 * e:        an element
 * elenme:   the length, Elen [me], of element list of pivotal variable
 * eln:      the length, Elen [...], of an element list
 * hash:     the computed value of the hash function
 * i:        a supervariable
 * ilast:    the entry in a link list preceding i
 * inext:    the entry in a link list following i
 * j:        a supervariable
 * jlast:    the entry in a link list preceding j
 * jnext:    the entry in a link list, or path, following j
 * k:        the pivot order of an element or variable
 * knt1:     loop counter used during element construction
 * knt2:     loop counter used during element construction
 * knt3:     loop counter used during compression
 * lenj:     Len [j]
 * ln:       length of a supervariable list
 * me:       current supervariable being eliminated, and the current
 *           element created by eliminating that supervariable
 * mindeg:   current minimum degree
 * nel:      number of pivots selected so far
 * nleft:    n - nel, the number of nonpivotal rows/columns remaining
 * nvi:      the number of variables in a supervariable i (= Nv [i])
 * nvj:      the number of variables in a supervariable j (= Nv [j])
 * nvpiv:    number of pivots in current element
 * slenme:   number of variables in variable list of pivotal variable
 * wbig:     = (INT_MAX - n) for the int version, (SuiteSparse_long_max - n)
 *                   for the SuiteSparse_long version.  wflg is not allowed to
 *                   be >= wbig.
 * we:       W [e]
 * wflg:     used for flagging the W array.  See description of Iw.
 * wnvi:     wflg - Nv [i]
 * x:        either a supervariable or an element
 *
 * ok:       true if supervariable j can be absorbed into i
 * ndense:   number of "dense" rows/columns
 * dense:    rows/columns with initial degree > dense are considered "dense"
 * aggressive:  true if aggressive absorption is being performed
 * ncmpa:    number of garbage collections

 * ---------------------------------------------------------------------------
 * LOCAL DOUBLES, used for statistical output only (except for alpha):
 * ---------------------------------------------------------------------------
 */

            abip_float f;
            abip_float r;
            abip_float ndiv;
            abip_float s;
            abip_float nms_lu;
            abip_float nms_ldl;
            abip_float dmax;
            abip_float alpha;
            abip_float lnz;
            abip_float lnzme;

/*
 * f:        nvpiv
 * r:        degme + nvpiv
 * ndiv:     number of divisions for LU or LDL' factorizations
 * s:        number of multiply-subtract pairs for LU factorization, for the
 *           current element me
 * nms_lu    number of multiply-subtract pairs for LU factorization
 * nms_ldl   number of multiply-subtract pairs for LDL' factorization
 * dmax:     the largest number of entries in any column of L, including the
 *           diagonal
 * alpha:    "dense" degree ratio
 * lnz:      the number of nonzeros in L (excluding the diagonal)
 * lnzme:    the number of nonzeros in L (excl. the diagonal) for the
```

```
 *          current element me

 * ----------------------------------------------------------------------------
 * LOCAL "POINTERS" (indices into the Iw array)
 * ----------------------------------------------------------------------------
*/

            Int p;
            Int p1;
            Int p2;
            Int p3;
            Int p4;
            Int pdst;
            Int pend;
            Int pj;
            Int pme;
            Int pme1;
            Int pme2;
            Int pn;
            Int psrc;

/*
 * Any parameter (Pe [...] or pfree) or local variable starting with "p" (for
 * Pointer) is an index into Iw, and all indices into Iw use variables starting
 * with "p."  The only exception to this rule is the iwlen input argument.
 *
 * p:           pointer into lots of things
 * p1:          Pe [i] for some variable i (start of element list)
 * p2:          Pe [i] + Elen [i] -  1 for some variable i
 * p3:          index of first supervariable in clean list
 * p4:
 * pdst:        destination pointer, for compression
 * pend:        end of memory to compress
 * pj:          pointer into an element or variable
 * pme:         pointer into the current element (pme1...pme2)
 * pme1:        the current element, me, is stored in Iw [pme1...pme2]
 * pme2:        the end of the current element
 * pn:          pointer into a "clean" variable, also used to compress
 * psrc:        source pointer, for compression
*/

/* ========================================================================= */
/*  INITIALIZATIONS */
/* ========================================================================= */

            /* Note that this restriction on iwlen is slightly more restrictive than
            * what is actually required in AMD_2.  AMD_2 can operate with no elbow
            * room at all, but it will be slow.  For better performance, at least
            * size-n elbow room is enforced. */
            ASSERT (iwlen >= pfree + n) ;
            ASSERT (n > 0) ;

            /* initialize output statistics */
            lnz = 0 ;
            ndiv = 0 ;
            nms_lu = 0 ;
            nms_ldl = 0 ;
            dmax = 1 ;
            me = EMPTY ;

            mindeg = 0 ;
            ncmpa = 0 ;
            nel = 0 ;
            lemax = 0 ;

            /* get control parameters */
            if (Control != (abip_float *) ABIP_NULL)
            {
               alpha = Control [AMD_DENSE] ;
               aggressive = (Control [AMD_AGGRESSIVE] != 0) ;
            }
            else
            {
               alpha = AMD_DEFAULT_DENSE ;
               aggressive = AMD_DEFAULT_AGGRESSIVE ;
            }

            /* Note: if alpha is NaN, this is undefined: */
            if (alpha < 0)
            {
               /* only remove completely dense rows/columns */
               dense = n-2 ;
            }
            else
            {
               dense = alpha * sqrt ((abip_float) n) ;
            }
```

```
            dense = MAX (16, dense) ;
            dense = MIN (n,  dense) ;
            AMD_DEBUG1 (("\n\nAMD (debug), alpha %g, aggr. "ID"\n", alpha, aggressive)) ;

            for (i = 0 ; i < n ; i++)
            {
               Last [i] = EMPTY ;
               Head [i] = EMPTY ;
               Next [i] = EMPTY ;

                        /* if separate Hhead array is used for hash buckets: Hhead [i] = EMPTY ; */
               Nv [i] = 1 ;
               W [i] = 1 ;
               Elen [i] = 0 ;
               Degree [i] = Len [i] ;
            }

            #ifndef NDEBUG

            AMD_DEBUG1 (("\n======Nel "ID" initial\n", nel)) ;
            AMD_dump (n, Pe, Iw, Len, iwlen, pfree, Nv, Next, Last, Head, Elen, Degree, W, -1) ;

            #endif

            /* initialize wflg */
            wbig = Int_MAX - n ;
            wflg = clear_flag (0, wbig, W, n) ;

            /* --------------------------------------------------------------------- */
            /* initialize degree lists and eliminate dense and empty rows */
            /* --------------------------------------------------------------------- */

            ndense = 0 ;

            for (i = 0 ; i < n ; i++)
            {
               deg = Degree [i] ;
               ASSERT (deg >= 0 && deg < n) ;

                        if (deg == 0)
               {
                            /* -------------------------------------------------------------
                        * we have a variable that can be eliminated at once because
                        * there is no off-diagonal non-zero in its row.  Note that
                        * Nv [i] = 1 for an empty variable i.  It is treated just
                        * the same as an eliminated element i.
                        * ------------------------------------------------------------- */

                        Elen [i] = FLIP (1) ;
                        nel++ ;
                        Pe [i] = EMPTY ;
                        W [i] = 0 ;
                        }
               else if (deg > dense)
               {
                            /* -------------------------------------------------------------
                        * Dense variables are not treated as elements, but as unordered,
                        * non-principal variables that have no parent.  They do not take
                        * part in the postorder, since Nv [i] = 0.  Note that the Fortran
                        * version does not have this option.
                        * ------------------------------------------------------------- */

                        AMD_DEBUG1 (("Dense node "ID" degree "ID"\n", i, deg)) ;
                        ndense++ ;
                        Nv [i] = 0 ;         /* do not postorder this node */
                        Elen [i] = EMPTY ;
                        nel++ ;
                        Pe [i] = EMPTY ;
               }
               else
               {
                            /* -------------------------------------------------------------
                        * place i in the degree list corresponding to its degree
                        * ------------------------------------------------------------- */

                        inext = Head [deg] ;
                        ASSERT (inext >= EMPTY && inext < n) ;
                        if (inext != EMPTY) Last [inext] = i ;
                        Next [i] = inext ;
                        Head [deg] = i ;
                        }
            }

            /* ========================================================================= */
            /* WHILE (selecting pivots) DO */
            /* ========================================================================= */
```

```
	while (nel < n)
	{

	    #ifndef NDEBUG

	    AMD_DEBUG1 (("\n======Nel "ID"\n", nel)) ;
	    if (AMD_debug >= 2)
	    {
		AMD_dump (n, Pe, Iw, Len, iwlen, pfree, Nv, Next, Last, Head, Elen, Degree, W, nel) ;
	    }

	    #endif

	    /* ========================================================================= */
	    /* GET PIVOT OF MINIMUM DEGREE */
	    /* ========================================================================= */

	    /* ----------------------------------------------------------------- */
	    /* find next supervariable for elimination */
	    /* ----------------------------------------------------------------- */

	    ASSERT (mindeg >= 0 && mindeg < n) ;
	    for (deg = mindeg ; deg < n ; deg++)
	    {
		me = Head [deg] ;
		if (me != EMPTY) break ;
	    }
	    mindeg = deg ;
	    ASSERT (me >= 0 && me < n) ;
	    AMD_DEBUG1 (("=================me: "ID"\n", me)) ;

	    /* ----------------------------------------------------------------- */
	    /* remove chosen variable from link list */
	    /* ----------------------------------------------------------------- */

	    inext = Next [me] ;
	    ASSERT (inext >= EMPTY && inext < n) ;
	    if (inext != EMPTY) Last [inext] = EMPTY ;
	    Head [deg] = inext ;

	    /* ----------------------------------------------------------------- */
	    /* me represents the elimination of pivots nel to nel+Nv[me]-1. */
	    /* place me itself as the first in this set. */
	    /* ----------------------------------------------------------------- */

	    elenme = Elen [me] ;
	    nvpiv = Nv [me] ;
	    ASSERT (nvpiv > 0) ;
	    nel += nvpiv ;

	    /* ========================================================================= */
	    /* CONSTRUCT NEW ELEMENT */
	    /* ========================================================================= */

	    /* -----------------------------------------------------------------
	     * At this point, me is the pivotal supervariable.  It will be
	     * converted into the current element.  Scan list of the pivotal
	     * supervariable, me, setting tree pointers and constructing new list
	     * of supervariables for the new element, me.  p is a pointer to the
	     * current position in the old list.
	     * ----------------------------------------------------------------- */

	    /* flag the variable "me" as being in Lme by negating Nv [me] */
	    Nv [me] = -nvpiv ;
	    degme = 0 ;
	    ASSERT (Pe [me] >= 0 && Pe [me] < iwlen) ;

	    if (elenme == 0)
	    {
		/* ------------------------------------------------------------- */
		/* construct the new element in place */
		/* ------------------------------------------------------------- */

		pme1 = Pe [me] ;
		pme2 = pme1 - 1 ;

		for (p = pme1 ; p <= pme1 + Len [me] - 1 ; p++)
		{
		    i = Iw [p] ;
		    ASSERT (i >= 0 && i < n && Nv [i] >= 0) ;
```

```
                                        nvi = Nv [i] ;

                                                    if (nvi > 0)
                                        {
                                                            /* ------------------------------------------------------- */
                                                    /* i is a principal variable not yet placed in Lme. */
                                                    /* store i in new list */
                                                    /* ------------------------------------------------------- */

                                                    /* flag i as being in Lme by negating Nv [i] */
                                                    degme += nvi ;
                                                    Nv [i] = -nvi ;
                                                    Iw [++pme2] = i ;

                                                    /* ------------------------------------------------------- */
                                                    /* remove variable i from degree list. */
                                                    /* ------------------------------------------------------- */
                                                                  ilast = Last [i] ;
                                                                  inext = Next [i] ;
                                                    ASSERT (ilast >= EMPTY && ilast < n) ;
                                                    ASSERT (inext >= EMPTY && inext < n) ;
                                                    if (inext != EMPTY) Last [inext] = ilast ;
                                                    if (ilast != EMPTY)
                                                    {
                                                       Next [ilast] = inext ;
                                                    }
                                                    else
                                                    {
                                                       /* i is at the head of the degree list */
                                                       ASSERT (Degree [i] >= 0 && Degree [i] < n) ;
                                                       Head [Degree [i]] = inext ;
                                                    }
                                         }
                                    }
                  }
                  else
                  {

                                /* ------------------------------------------------------------- */
                                /* construct the new element in empty space, Iw [pfree ...] */
                                /* ------------------------------------------------------------- */

                                p = Pe [me] ;
                                pme1 = pfree ;
                                slenme = Len [me] - elenme ;

                                for (knt1 = 1 ; knt1 <= elenme + 1 ; knt1++)
                                {

                                       if (knt1 > elenme)
                                       {
                                                    /* search the supervariables in me. */
                                                    e = me ;
                                                    pj = p ;
                                                    ln = slenme ;
                                                    AMD_DEBUG2 (("Search sv: "ID" "ID" "ID"\n", me,pj,ln)) ;
                                       }
                                       else
                                       {
                                                    /* search the elements in me. */
                                                    e = Iw [p++] ;
                                                    ASSERT (e >= 0 && e < n) ;
                                                    pj = Pe [e] ;
                                                    ln = Len [e] ;
                                                    AMD_DEBUG2 (("Search element e "ID" in me "ID"\n", e,me)) ;
                                                    ASSERT (Elen [e] < EMPTY && W [e] > 0 && pj >= 0) ;
                                       }

                                                         ASSERT (ln >= 0 && (ln == 0 || (pj >= 0 && pj < iwlen))) ;

                                      /* -------------------------------------------------------
                                       * search for different supervariables and add them to the
                                       * new list, compressing when necessary. this loop is
                                       * executed once for each element in the list and once for
                                       * all the supervariables in the list.
                                       * ------------------------------------------------------- */

                                       for (knt2 = 1 ; knt2 <= ln ; knt2++)
                                       {
                                                    i = Iw [pj++] ;
                                                    ASSERT (i >= 0 && i < n && (i == me || Elen [i] >=
```

```
EMPTY));
                nvi = Nv [i] ;
                AMD_DEBUG2 ((": "ID" "ID" "ID" "ID"\n", i, Elen [i], Nv [i], wflg)) ;

                if (nvi > 0)
                {

                    /* ------------------------------------------------- */
                    /* compress Iw, if necessary */
                    /* ------------------------------------------------- */

                    if (pfree >= iwlen)
                    {

                        AMD_DEBUG1 (("GARBAGE COLLECTION\n")) ;

                        /* prepare for compressing Iw by adjusting pointers
                         * and lengths so that the lists being searched in
                         * the inner and outer loops contain only
                         * the remaining entries. */

                        Pe [me] = p ;
                        Len [me] -= knt1 ;

                        /* check if nothing left of supervariable me */
                        if (Len [me] == 0) Pe [me] = EMPTY ;
                        Pe [e] = pj ;
                        Len [e] = ln - knt2 ;

                        /* nothing left of element e */
                        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 < iwlen) ;
                                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 object j: "ID"\n", j)) ;
                                Iw [pdst] = Pe [j] ;
                                Pe [j] = pdst++ ;
                                lenj = Len [j] ;

                                /* copy from source to destination */
                                for (knt3 = 0 ; knt3 <= lenj - 2 ; knt3++)
                                {
                                    Iw [pdst++] = Iw [psrc++] ;
                                }
                            }
                        }

```

```
                                                /* move the new partially-constructed element */
                                                p1 = pdst ;
                                                for (psrc = pme1 ; psrc <= pfree-1 ; psrc++)
                                                {
                                                    Iw [pdst++] = Iw [psrc] ;
                                                }
                                                pme1 = p1 ;
                                                pfree = pdst ;
                                                pj = Pe [e] ;
                                                p = Pe [me] ;
                                        }

                                        /* ------------------------------------------------- */
                                        /* i is a principal variable not yet placed in Lme */
                                        /* store i in new list */
                                        /* ------------------------------------------------- */

                                        /* flag i as being in Lme by negating Nv [i] */
                                        degme += nvi ;
                                        Nv [i] = -nvi ;
                                        Iw [pfree++] = i ;
                                        AMD_DEBUG2 (("     s: "ID"     nv "ID"\n", i, Nv [i]));

                                        /* ------------------------------------------------- */
                                        /* remove variable i from degree link list */
                                        /* ------------------------------------------------- */

                                        ilast = Last [i] ;
                                        inext = Next [i] ;
                                        ASSERT (ilast >= EMPTY && ilast < n) ;
                                        ASSERT (inext >= EMPTY && inext < n) ;
                                        if (inext != EMPTY) Last [inext] = ilast ;
                                        if (ilast != EMPTY)
                                        {
                                                Next [ilast] = inext ;
                                        }
                                        else
                                        {
                                                /* i is at the head of the degree list */
                                                ASSERT (Degree [i] >= 0 && Degree [i] < n) ;
                                                Head [Degree [i]] = inext ;
                                        }
                                    }
                            }

                            if (e != me)
                            {
                                    /* set tree pointer and flag to indicate element e is
                                     * absorbed into new element me (the parent of e is me) */
                                    AMD_DEBUG1 ((" Element "ID" => "ID"\n", e, me)) ;
                                    Pe [e] = FLIP (me) ;
                                    W [e] = 0 ;
                            }
                    }

                    pme2 = pfree - 1 ;
            }

            /* ----------------------------------------------------------------- */
            /* me has now been converted into an element in Iw [pme1..pme2] */
            /* ----------------------------------------------------------------- */

            /* degme holds the external degree of new element */
            Degree [me] = degme ;
            Pe [me] = pme1 ;
            Len [me] = pme2 - pme1 + 1 ;
            ASSERT (Pe [me] >= 0 && Pe [me] < iwlen) ;

            Elen [me] = FLIP (nvpiv + degme) ;
            /* FLIP (Elen (me)) is now the degree of pivot (including
            * diagonal part). */

                    #ifndef NDEBUG

                    AMD_DEBUG2 (("New element structure: length= "ID"\n", pme2-pme1+1)) ;
            for (pme = pme1 ; pme <= pme2 ; pme++) AMD_DEBUG3 ((" "ID"", Iw[pme]));
            AMD_DEBUG3 (("\n")) ;

```

```
            #endif

        /* --------------------------------------------------------------- */
        /* make sure that wflg is not too large. */
        /* --------------------------------------------------------------- */

        /* With the current value of wflg, wflg+n must not cause integer
        * overflow */

        wflg = clear_flag (wflg, wbig, W, n) ;

            /* =====================================================================
*/
            /* COMPUTE (W [e] - wflg) = |Le\Lme| FOR ALL ELEMENTS */
            /* =====================================================================
*/

        /* -----------------------------------------------------------------
        * Scan 1:  compute the external degrees of previous elements with
        * respect to the current element.  That is:
        *       (W [e] - wflg) = |Le \ Lme|
        * for each element e that appears in any supervariable in Lme.  The
        * notation Le refers to the pattern (list of supervariables) of a
        * previous element e, where e is not yet absorbed, stored in
        * Iw [Pe [e] + 1 ... Pe [e] + Len [e]].  The notation Lme
        * refers to the pattern of the current element (stored in
        * Iw [pme1..pme2]).   If aggressive absorption is enabled, and
        * (W [e] - wflg) becomes zero, then the element e will be absorbed
        * in Scan 2.
        * ----------------------------------------------------------------- */

        AMD_DEBUG2 (("me: ")) ;
        for (pme = pme1 ; pme <= pme2 ; pme++)
        {
                i = Iw [pme] ;
                ASSERT (i >= 0 && i < n) ;
                eln = Elen [i] ;
                AMD_DEBUG3 ((""ID" Elen "ID": \n", i, eln)) ;

                    if (eln > 0)
                {
                    /* note that Nv [i] has been negated to denote i in Lme: */
                    nvi = -Nv [i] ;
                    ASSERT (nvi > 0 && Pe [i] >= 0 && Pe [i] < iwlen) ;
                    wnvi = wflg - nvi ;
                    for (p = Pe [i] ; p <= Pe [i] + eln - 1 ; p++)
                    {
                            e = Iw [p] ;
                            ASSERT (e >= 0 && e < n) ;
                            we = W [e] ;
                            AMD_DEBUG4 (("    e "ID" we "ID" ", e, we)) ;

                                  if (we >= wflg)
                            {
                              /* unabsorbed element e has been seen in this loop */
                              AMD_DEBUG4 (("    unabsorbed, first time seen")) ;
                              we -= nvi ;
                            }
                            else if (we != 0)
                            {
                              /* e is an unabsorbed element */
                              /* this is the first we have seen e in all of Scan 1
*/
                              AMD_DEBUG4 (("    unabsorbed")) ;
                              we = Degree [e] + wnvi ;
                            }

                                  AMD_DEBUG4 (("\n")) ;
                            W [e] = we ;
                    }
                }
        }

            AMD_DEBUG2 (("\n")) ;

            /* =====================================================================
*/
            /* DEGREE UPDATE AND ELEMENT ABSORPTION */
            /* =====================================================================
*/

        /* -----------------------------------------------------------------
        * Scan 2:  for each i in Lme, sum up the degree of Lme (which is
        * degme), plus the sum of the external degrees of each Le for the
        * elements e appearing within i, plus the supervariables in i.
        * Place i in hash list.
        * ----------------------------------------------------------------- */
```

```

            for (pme = pme1 ; pme <= pme2 ; pme++)
            {
                i = Iw [pme] ;
                ASSERT (i >= 0 && i < n && Nv [i] < 0 && Elen [i] >= 0) ;
                AMD_DEBUG2 (("Updating: i "ID" "ID" "ID"\n", i, Elen[i], Len [i]));
                p1 = Pe [i] ;
                p2 = p1 + Elen [i] - 1 ;
                pn = p1 ;
                hash = 0 ;
                deg = 0 ;
                ASSERT (p1 >= 0 && p1 < iwlen && p2 >= -1 && p2 < iwlen) ;

                /* ------------------------------------------------------------- */
                /* scan the element list associated with supervariable i */
                /* ------------------------------------------------------------- */

                /* UMFPACK/MA38-style approximate degree: */
                if (aggressive)
                {
                    for (p = p1 ; p <= p2 ; p++)
                    {
                        e = Iw [p] ;
                        ASSERT (e >= 0 && e < n) ;
                        we = W [e] ;
                        if (we != 0)
                        {
                            /* e is an unabsorbed element */
                            /* dext = | Le \ Lme | */
                            dext = we - wflg ;
                            if (dext > 0)
                            {
                                deg += dext ;
                                Iw [pn++] = e ;
                                hash += e ;
                                AMD_DEBUG4 ((" e: "ID" hash = "ID"\n",e,hash)) ;
                            }
                            else
                            {
                                /* external degree of e is zero, absorb e into me*/
                                AMD_DEBUG1 ((" Element "ID" =>"ID" (aggressive)\n", e, me)) ;
                                ASSERT (dext == 0) ;
                                Pe [e] = FLIP (me) ;
                                W [e] = 0 ;
                            }
                        }
                    }
                }
                else
                {
                    for (p = p1 ; p <= p2 ; p++)
                    {
                        e = Iw [p] ;
                        ASSERT (e >= 0 && e < n) ;
                        we = W [e] ;
                        if (we != 0)
                        {
                            /* e is an unabsorbed element */
                            dext = we - wflg ;
                            ASSERT (dext >= 0) ;
                            deg += dext ;
                            Iw [pn++] = e ;
                            hash += e ;
                            AMD_DEBUG4 (("   e: "ID" hash = "ID"\n",e,hash)) ;
                        }
                    }
                }

                /* count the number of elements in i (including me): */
                Elen [i] = pn - p1 + 1 ;

                /* ------------------------------------------------------------- */
                /* scan the supervariables in the list associated with i */
                /* ------------------------------------------------------------- */

                /* The bulk of the AMD run time is typically spent in this loop,
                * particularly if the matrix has many dense rows that are not
                * removed prior to ordering. */
                p3 = pn ;
                p4 = p1 + Len [i] ;
                for (p = p2 + 1 ; p < p4 ; p++)
                {
                    j = Iw [p] ;
                    ASSERT (j >= 0 && j < n) ;
```

```
                nvj = Nv [j] ;
                if (nvj > 0)
                {
                    /* j is unabsorbed, and not in Lme. */
                    /* add to degree and add to new list */
                    deg += nvj ;
                    Iw [pn++] = j ;
                    hash += j ;
                    AMD_DEBUG4 ((" s: "ID" hash "ID" Nv[j]= "ID"\n", j, hash, nvj)) ;
                }
            }

            /* ------------------------------------------------------------- */
            /* update the degree and check for mass elimination */
            /* ------------------------------------------------------------- */

            /* with aggressive absorption, deg==0 is identical to the
             * Elen [i] == 1 && p3 == pn test, below. */

                 ASSERT (IMPLIES (aggressive, (deg==0) == (Elen[i]==1 && p3==pn))) ;

            if (Elen [i] == 1 && p3 == pn)
            {

                /* --------------------------------------------------------- */
                /* mass elimination */
                /* --------------------------------------------------------- */

                /* There is nothing left of this node except for an edge to
                 * the current pivot element.  Elen [i] is 1, and there are
                 * no variables adjacent to node i.  Absorb i into the
                 * current pivot element, me.  Note that if there are two or
                 * more mass eliminations, fillin due to mass elimination is
                 * possible within the nvpiv-by-nvpiv pivot block.  It is this
                 * step that causes AMD's analysis to be an upper bound.
                 *
                 * The reason is that the selected pivot has a lower
                 * approximate degree than the true degree of the two mass
                 * eliminated nodes.  There is no edge between the two mass
                 * eliminated nodes.  They are merged with the current pivot
                 * anyway.
                 *
                 * No fillin occurs in the Schur complement, in any case,
                 * and this effect does not decrease the quality of the
                 * ordering itself, just the quality of the nonzero and
                 * flop count analysis.  It also means that the post-ordering
                 * is not an exact elimination tree post-ordering. */

                AMD_DEBUG1 ((" MASS i "ID" => parent e "ID"\n", i, me)) ;
                Pe [i] = FLIP (me) ;
                nvi = -Nv [i] ;
                degme -= nvi ;
                nvpiv += nvi ;
                nel += nvi ;
                Nv [i] = 0 ;
                Elen [i] = EMPTY ;
                  }
            else
            {
                                /* --------------------------------------------------------- */
                /* update the upper-bound degree of i */
                /* --------------------------------------------------------- */

                /* the following degree does not yet include the size
                 * of the current element, which is added later: */

                Degree [i] = MIN (Degree [i], deg) ;

                /* --------------------------------------------------------- */
                /* add me to the list for i */
                /* --------------------------------------------------------- */

                /* move first supervariable to end of list */
                Iw [pn] = Iw [p3] ;

                                /* move first element to end of element part of list */
                Iw [p3] = Iw [p1] ;

                                /* add new element, me, to front of list. */
                Iw [p1] = me ;

                                /* store the new length of the list in Len [i] */
                Len [i] = pn - p1 + 1 ;
```

```

                            /* ------------------------------------------------------ */
                            /* 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] = degme ;

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

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

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

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

                            /* ---------------------------------------------------------
                            * examine all hash buckets with 2 or more variables.  We do
                            * this by examing 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 ;
```

```
                        }
                        else
                        {
                                /* degree list is not empty, restore Last [j] of head j
*/
                                i = Last [j] ;
                                Last [j] = EMPTY ;
                        }

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

                        ASSERT (i >= EMPTY && i < n) ;
                        AMD_DEBUG2 (("----i "ID" hash "ID"\n", i, hash)) ;

                        while (i != EMPTY && Next [i] != EMPTY)
                        {

                                /* -----------------------------------------------------
                                * this bucket has one or more variables following i.
                                * scan all of them to see if i can absorb any entries
                                * that follow i in hash bucket.  Scatter i into w.
                                * -----------------------------------------------------
*/

                                ln = Len [i] ;
                                eln = Elen [i] ;
                                ASSERT (ln >= 0 && eln >= 0) ;
                                ASSERT (Pe [i] >= 0 && Pe [i] < iwlen) ;
                                            /* do not flag the first element in
the list (me) */
                                for (p = Pe [i] + 1 ; p <= Pe [i] + ln - 1 ; p++)
                                {
                                  ASSERT (Iw [p] >= 0 && Iw [p] < n) ;
                                  W [Iw [p]] = wflg ;
                                }

                                /* -----------------------------------------------------
*/
                                /* scan every other entry j following i in bucket */
                                /* -----------------------------------------------------
*/

                                jlast = i ;
                                j = Next [i] ;
                                ASSERT (j >= EMPTY && j < n) ;

                                while (j != EMPTY)
                                {
                                  /* -------------------------------------------------
*/
                                  /* check if j and i have identical nonzero pattern */
                                  /* -------------------------------------------------
*/

                                  AMD_DEBUG3 (("compare i "ID" and j "ID"\n", i,j)) ;

                                  /* check if i and j have the same Len and Elen */
                                  ASSERT (Len [j] >= 0 && Elen [j] >= 0) ;
                                  ASSERT (Pe [j] >= 0 && Pe [j] < iwlen) ;
                                  ok = (Len [j] == ln) && (Elen [j] == eln) ;
                                                  /* skip the first element
in the list (me) */
                                  for (p = Pe [j] + 1 ; ok && p <= Pe [j] + ln - 1 ;
p++)
                                  {
                                          ASSERT (Iw [p] >= 0 && Iw [p] < n) ;
                                          if (W [Iw [p]] != wflg) ok = 0 ;
                                  }

                                                  if (ok)
                                  {
                                          /*
--------------------------------------------- */
                                          /* found it!  j can be absorbed into i */
                                          /*
--------------------------------------------- */

                                          AMD_DEBUG1 (("found it! j "ID" => i
"ID"\n", j,i));
                                          Pe [j] = FLIP (i) ;

                                                        /* both Nv [i]
```

```
and Nv [j] are negated since they */
                                        /* are in Lme, and the absolute values of
each */
                                        /* are the number of variables in i and
j: */
                                        Nv [i] += Nv [j] ;
                                        Nv [j] = 0 ;
                                        Elen [j] = EMPTY ;

                                                                    /* delete j
from hash bucket */
                                        ASSERT (j != Next [j]) ;
                                        j = Next [j] ;
                                        Next [jlast] = j ;
                                                  }
                                else
                                {
                                        /* j cannot be absorbed into i */
                                        jlast = j ;
                                        ASSERT (j != Next [j]) ;
                                        j = Next [j] ;
                                }
                                ASSERT (j >= EMPTY && j < n) ;
                              }

                              /* -------------------------------------------------------
                              * no more variables can be absorbed into i
                              * go to next i in bucket and clear flag array
                              * -------------------------------------------------------
*/

                              wflg++ ;
                              i = Next [i] ;
                              ASSERT (i >= EMPTY && i < n) ;
                                 }
                  }
        }

                AMD_DEBUG2 (("detect done\n")) ;

                /* =========================================================================
*/
                /* RESTORE DEGREE LISTS AND REMOVE NONPRINCIPAL SUPERVARIABLES FROM ELEMENT */
                /* =========================================================================
*/

        p = pme1 ;
        nleft = n - nel ;
        for (pme = pme1 ; pme <= pme2 ; pme++)
        {
                i = Iw [pme] ;
                ASSERT (i >= 0 && i < n) ;
                nvi = -Nv [i] ;
                AMD_DEBUG3 (("Restore i "ID" "ID"\n", i, nvi)) ;
                if (nvi > 0)
                {
                        /* i is a principal variable in Lme */
                        /* restore Nv [i] to signify that i is principal */
                        Nv [i] = nvi ;

                        /* --------------------------------------------------------- */
                        /* compute the external degree (add size of current element) */
                        /* --------------------------------------------------------- */

                        deg = Degree [i] + degme - nvi ;
                        deg = MIN (deg, nleft - nvi) ;
                        ASSERT (IMPLIES (aggressive, deg > 0) && deg >= 0 && deg < n) ;

                        /* --------------------------------------------------------- */
                        /* place the supervariable at the head of the degree list */
                        /* --------------------------------------------------------- */

                        inext = Head [deg] ;
                        ASSERT (inext >= EMPTY && inext < n) ;
                        if (inext != EMPTY) Last [inext] = i ;
                        Next [i] = inext ;
                        Last [i] = EMPTY ;
                        Head [deg] = i ;

                        /* --------------------------------------------------------- */
                        /* save the new degree, and find the minimum degree */
                        /* --------------------------------------------------------- */

                        mindeg = MIN (mindeg, deg) ;
                        Degree [i] = deg ;

                        /* --------------------------------------------------------- */
```

```
                        /* place the supervariable in the element pattern */
                        /* ---------------------------------------------------------- */

                        Iw [p++] = i ;
                          }
              }
              AMD_DEBUG2 (("restore done\n")) ;

                     /* =====================================================================
*/
                     /* FINALIZE THE NEW ELEMENT */
                     /* =====================================================================
*/

                 AMD_DEBUG2 (("ME = "ID" DONE\n", me)) ;
              Nv [me] = nvpiv ;

                     /* save the length of the list for the new element me */
              Len [me] = p - pme1 ;
              if (Len [me] == 0)
              {
                        /* there is nothing left of the current pivot element */
                        /* it is a root of the assembly tree */
                        Pe [me] = EMPTY ;
                        W [me] = 0 ;
              }

                     if (elenme != 0)
              {
                        /* element was not constructed in place: deallocate part of */
                        /* it since newly nonprincipal variables may have been removed */
                        pfree = p ;
              }

              /* The new element has nvpiv pivots and the size of the contribution
              * block for a multifrontal method is degme-by-degme, not including
              * the "dense" rows/columns.  If the "dense" rows/columns are included,
              * the frontal matrix is no larger than
              * (degme+ndense)-by-(degme+ndense).
              */

              if (ABIPInfo != (abip_float *) ABIP_NULL)
              {
                        f = nvpiv ;
                        r = degme + ndense ;
                        dmax = MAX (dmax, f + r) ;

                        /* number of nonzeros in L (excluding the diagonal) */
                        lnzme = f*r + (f-1)*f/2 ;
                        lnz += lnzme ;

                        /* number of divide operations for LDL' and for LU */
                        ndiv += lnzme ;

                        /* number of multiply-subtract pairs for LU */
                        s = f*r*r + r*(f-1)*f + (f-1)*f*(2*f-1)/6 ;
                        nms_lu += s ;

                        /* number of multiply-subtract pairs for LDL' */
                        nms_ldl += (s + lnzme)/2 ;
              }

                     #ifndef NDEBUG

                     AMD_DEBUG2 (("finalize done nel "ID" n "ID"\n   ::::\n", nel, n)) ;
              for (pme = Pe [me] ; pme <= Pe [me] + Len [me] - 1 ; pme++)
              {
                        AMD_DEBUG3 ((" "ID"", Iw [pme])) ;
              }
              AMD_DEBUG3 (("\n")) ;

                     #endif
           }

           /* ========================================================================= */
           /* DONE SELECTING PIVOTS */
           /* ========================================================================= */

           if (ABIPInfo != (abip_float *) ABIP_NULL)
           {

              /* count the work to factorize the ndense-by-ndense submatrix */
              f = ndense ;
              dmax = MAX (dmax, (abip_float) ndense) ;

              /* number of nonzeros in L (excluding the diagonal) */
              lnzme = (f-1)*f/2 ;
```

```
            lnz += lnzme ;

            /* number of divide operations for LDL' and for LU */
            ndiv += lnzme ;

            /* number of multiply-subtract pairs for LU */
            s = (f-1)*f*(2*f-1)/6 ;
            nms_lu += s ;

            /* number of multiply-subtract pairs for LDL' */
            nms_ldl += (s + lnzme)/2 ;

            /* number of nz's in L (excl. diagonal) */
            ABIPInfo [AMD_LNZ] = lnz ;

            /* number of divide ops for LU and LDL' */
            ABIPInfo [AMD_NDIV] = ndiv ;

            /* number of multiply-subtract pairs for LDL' */
            ABIPInfo [AMD_NMULTSUBS_LDL] = nms_ldl ;

            /* number of multiply-subtract pairs for LU */
            ABIPInfo [AMD_NMULTSUBS_LU] = nms_lu ;

            /* number of "dense" rows/columns */
            ABIPInfo [AMD_NDENSE] = ndense ;

            /* largest front is dmax-by-dmax */
            ABIPInfo [AMD_DMAX] = dmax ;

            /* number of garbage collections in AMD */
            ABIPInfo [AMD_NCMPA] = ncmpa ;

            /* successful ordering */
            ABIPInfo [AMD_STATUS] = AMD_OK ;
         }

         /* ========================================================================= */
         /* POST-ORDERING */
         /* ========================================================================= */

         /* -------------------------------------------------------------------------
         * Variables at this point:
         *
         * Pe: holds the elimination tree.  The parent of j is FLIP (Pe [j]),
         *   or EMPTY if j is a root.  The tree holds both elements and
         *   non-principal (unordered) variables absorbed into them.
         *   Dense variables are non-principal and unordered.
         *
         * Elen: holds the size of each element, including the diagonal part.
         *   FLIP (Elen [e]) > 0 if e is an element.  For unordered
         *   variables i, Elen [i] is EMPTY.
         *
         * Nv: Nv [e] > 0 is the number of pivots represented by the element e.
         *   For unordered variables i, Nv [i] is zero.
         *
         * Contents no longer needed:
         *   W, Iw, Len, Degree, Head, Next, Last.
         *
         * The matrix itself has been destroyed.
         *
         * n: the size of the matrix.
         * No other scalars needed (pfree, iwlen, etc.)
         * ------------------------------------------------------------------------- */

         /* restore Pe */
         for (i = 0 ; i < n ; i++)
         {
            Pe [i] = FLIP (Pe [i]) ;
         }

         /* restore Elen, for output information, and for postordering */
         for (i = 0 ; i < n ; i++)
         {
            Elen [i] = FLIP (Elen [i]) ;
         }

         /* Now the parent of j is Pe [j], or EMPTY if j is a root.  Elen [e] > 0
         * is the size of element e.  Elen [i] is EMPTY for unordered variable i. */

         #ifndef NDEBUG

         AMD_DEBUG2 (("\nTree:\n")) ;
         for (i = 0 ; i < n ; i++)
         {
            AMD_DEBUG2 ((" "ID" parent: "ID"   ", i, Pe [i])) ;
            ASSERT (Pe [i] >= EMPTY && Pe [i] < n) ;
```

```
        if (Nv [i] > 0)
        {
            /* this is an element */
            e = i ;
            AMD_DEBUG2 ((" element, size is "ID"\n", Elen [i])) ;
            ASSERT (Elen [e] > 0) ;
        }
        AMD_DEBUG2 (("\n")) ;
    }

    AMD_DEBUG2 (("\nelements:\n")) ;

    for (e = 0 ; e < n ; e++)
    {
        if (Nv [e] > 0)
        {
            AMD_DEBUG3 (("Element e= "ID" size "ID" nv "ID" \n", e, Elen [e], Nv [e])) ;
        }
    }

    AMD_DEBUG2 (("\nvariables:\n")) ;
    for (i = 0 ; i < n ; i++)
    {
        Int cnt ;
        if (Nv [i] == 0)
        {
            AMD_DEBUG3 (("i unordered: "ID"\n", i)) ;
            j = Pe [i] ;
            cnt = 0 ;
            AMD_DEBUG3 (("  j: "ID"\n", j)) ;
            if (j == EMPTY)
            {
                AMD_DEBUG3 (("   i is a dense variable\n")) ;
            }
            else
            {
                ASSERT (j >= 0 && j < n) ;
                while (Nv [j] == 0)
                {
                    AMD_DEBUG3 (("   j : "ID"\n", j)) ;
                    j = Pe [j] ;
                    AMD_DEBUG3 (("   j:: "ID"\n", j)) ;
                    cnt++ ;
                    if (cnt > n) break ;
                }
                e = j ;
                AMD_DEBUG3 (("   got to e: "ID"\n", e)) ;
            }
        }
    }

#endif

    /* ========================================================================= */
    /* compress the paths of the variables */
    /* ========================================================================= */

    for (i = 0 ; i < n ; i++)
    {
        if (Nv [i] == 0)
        {

            /* -------------------------------------------------------------
             * i is an un-ordered row.  Traverse the tree from i until
             * reaching an element, e.  The element, e, was the principal
             * supervariable of i and all nodes in the path from i to when e
             * was selected as pivot.
             * ------------------------------------------------------------- */

            AMD_DEBUG1 (("Path compression, i unordered: "ID"\n", i)) ;
            j = Pe [i] ;
            ASSERT (j >= EMPTY && j < n) ;
            AMD_DEBUG3 ((" j: "ID"\n", j)) ;

            if (j == EMPTY)
            {
                /* Skip a dense variable.  It has no parent. */
                AMD_DEBUG3 (("      i is a dense variable\n")) ;
                continue ;
            }

            /* while (j is a variable) */
            while (Nv [j] == 0)
            {
                AMD_DEBUG3 (("      j : "ID"\n", j)) ;
                j = Pe [j] ;
```

```
                        AMD_DEBUG3 (("      j:: "ID"\n", j)) ;
                        ASSERT (j >= 0 && j < n) ;
                    }

                        /* got to an element e */
                    e = j ;
                    AMD_DEBUG3 (("got to e: "ID"\n", e)) ;

                    /* -------------------------------------------------------------
                     * traverse the path again from i to e, and compress the path
                     * (all nodes point to e).  Path compression allows this code to
                     * compute in O(n) time.
                     * ----------------------------------------------------------- */

                    j = i ;

                        /* while (j is a variable) */
                    while (Nv [j] == 0)
                    {
                        jnext = Pe [j] ;
                        AMD_DEBUG3 (("j "ID" jnext "ID"\n", j, jnext)) ;
                        Pe [j] = e ;
                        j = jnext ;
                        ASSERT (j >= 0 && j < n) ;
                    }
            }
        }

        /* ========================================================================= */
        /* postorder the assembly tree */
        /* ========================================================================= */

        AMD_postorder (n, Pe, Nv, Elen,
            W,          /* output order */
            Head, Next, Last) ;  /* workspace */

        /* ========================================================================= */
        /* compute output permutation and inverse permutation */
        /* ========================================================================= */

        /* W [e] = k means that element e is the kth element in the new
         * order.  e is in the range 0 to n-1, and k is in the range 0 to
         * the number of elements.  Use Head for inverse order. */

        for (k = 0 ; k < n ; k++)
        {
            Head [k] = EMPTY ;
            Next [k] = EMPTY ;
        }

        for (e = 0 ; e < n ; e++)
        {
            k = W [e] ;
            ASSERT ((k == EMPTY) == (Nv [e] == 0)) ;
            if (k != EMPTY)
            {
                    ASSERT (k >= 0 && k < n) ;
                    Head [k] = e ;
            }
        }

        /* construct output inverse permutation in Next,
         * and permutation in Last */
        nel = 0 ;

        for (k = 0 ; k < n ; k++)
        {
            e = Head [k] ;
            if (e == EMPTY) break ;
            ASSERT (e >= 0 && e < n && Nv [e] > 0) ;
            Next [e] = nel ;
            nel += Nv [e] ;
        }
        ASSERT (nel == n - ndense) ;

        /* order non-principal variables (dense, & those merged into supervar's) */
        for (i = 0 ; i < n ; i++)
        {
            if (Nv [i] == 0)
            {
                    e = Pe [i] ;
                    ASSERT (e >= EMPTY && e < n) ;

                        if (e != EMPTY)
                    {
                        /* This is an unordered variable that was merged
                         * into element e via supernode detection or mass
```

```
                                * elimination of i when e became the pivot element.
                                * Place i in order just before e. */
                                ASSERT (Next [i] == EMPTY && Nv [e] > 0) ;
                                Next [i] = Next [e] ;
                                Next [e]++ ;
                          }
                          else
                          {
                                /* This is a dense unordered variable, with no parent.
                                * Place it last in the output order. */
                                Next [i] = nel++ ;
                          }
                    }
            }

            ASSERT (nel == n) ;

            AMD_DEBUG2 (("\n\nPerm:\n")) ;

            for (i = 0 ; i < n ; i++)
            {
               k = Next [i] ;
               ASSERT (k >= 0 && k < n) ;
               Last [k] = i ;
               AMD_DEBUG2 (("   perm ["ID"] = "ID"\n", k, i)) ;
            }
}
```

4.7 amd/amd_aat.c File Reference

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

Functions

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

4.7.1 Function Documentation

4.7.1.1 AMD_aat()

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

Definition at line 20 of file amd_aat.c.

4.8 amd_aat.c

Go to the documentation of this file.

```
/* ========================================================================= */
/* === AMD_aat ============================================================= */
/* ========================================================================= */

/* ------------------------------------------------------------------------- */
/* AMD, Copyright (c) Timothy A. Davis,                                  */
/* Patrick R. Amestoy, and Iain S. Duff.  See ../README.txt for License.     */
/* email: DrTimothyAldenDavis@gmail.com                                      */
/* ------------------------------------------------------------------------- */

/* AMD_aat:  compute the symmetry of the pattern of A, and count the number of
 * nonzeros each column of A+A' (excluding the diagonal).  Assumes the input
 * matrix has no errors, with sorted columns and no duplicates
 * (AMD_valid (n, n, Ap, Ai) must be AMD_OK, but this condition is not
 * checked).
 */

#include "amd_internal.h"

GLOBAL size_t AMD_aat  /* returns nz in A+A' */
(
        Int n,
        const Int Ap [ ],
        const Int Ai [ ],
        Int Len [ ],      /* Len [j]: length of column j of A+A', excl diagonal*/
        Int Tp [ ],       /* workspace of size n */
        abip_float ABIPInfo  [ ]
)
{
        Int p1;
        Int p2;
        Int p;
        Int pj;
        Int pj2;

        Int i;
        Int j;
        Int k;

        Int nzdiag;
        Int nzboth;
        Int nz;

        abip_float sym ;
        size_t nzaat ;

    #ifndef NDEBUG

        AMD_debug_init ("AMD AAT") ;
        for (k = 0 ; k < n ; k++) Tp [k] = EMPTY ;
        ASSERT (AMD_valid (n, n, Ap, Ai) == AMD_OK) ;

    #endif

        if (ABIPInfo != (abip_float *) ABIP_NULL)
        {
        /* clear the ABIPInfo array, if it exists */
        for (i = 0 ; i < AMD_INFO ; i++)
        {
                ABIPInfo [i] = EMPTY ;
        }
        ABIPInfo [AMD_STATUS] = AMD_OK ;
        }

        for (k = 0 ; k < n ; k++)
        {
        Len [k] = 0 ;
        }

        nzdiag = 0 ;
        nzboth = 0 ;
        nz = Ap [n] ;

        for (k = 0 ; k < n ; k++)
        {
        p1 = Ap [k] ;
        p2 = Ap [k+1] ;
        AMD_DEBUG2 (("\nAAT Column: "ID" p1: "ID" p2: "ID"\n", k, p1, p2)) ;

        /* construct A+A' */
        for (p = p1 ; p < p2 ; )
        {
```

```c
                /* scan the upper triangular part of A */
                j = Ai [p] ;

                if (j < k)
                {
                /* entry A (j,k) is in the strictly upper triangular part,
                * add both A (j,k) and A (k,j) to the matrix A+A' */
                Len [j]++ ;
                Len [k]++ ;
                AMD_DEBUG3 (("    upper ("ID","ID") ("ID","ID")\n", j,k, k,j));
                p++ ;
                }
                else if (j == k)
                {
                /* skip the diagonal */
                p++ ;
                nzdiag++ ;
                break ;
                }
                else /* j > k */
                {
                /* first entry below the diagonal */
                break ;
                }

                /* scan lower triangular part of A, in column j until reaching
                * row k.  Start where last scan left off. */
                ASSERT (Tp [j] != EMPTY) ;
                ASSERT (Ap [j] <= Tp [j] && Tp [j] <= Ap [j+1]) ;
                pj2 = Ap [j+1] ;

                for (pj = Tp [j] ; pj < pj2 ; )
                {
                i = Ai [pj] ;

                if (i < k)
                {
                        /* A (i,j) is only in the lower part, not in upper.
                        * add both A (i,j) and A (j,i) to the matrix A+A' */
                        Len [i]++ ;
                        Len [j]++ ;
                        AMD_DEBUG3 (("    lower ("ID","ID") ("ID","ID")\n", i,j, j,i)) ;
                        pj++ ;
                }
                else if (i == k)
                {
                        /* entry A (k,j) in lower part and A (j,k) in upper */
                        pj++ ;
                        nzboth++ ;
                        break ;
                }
                else /* i > k */
                {
                        /* consider this entry later, when k advances to i */
                        break ;
                }
                }
                Tp [j] = pj ;
        }

        /* Tp [k] points to the entry just below the diagonal in column k */
        Tp [k] = p ;
        }

        /* clean up, for remaining mismatched entries */
        for (j = 0 ; j < n ; j++)
        {
        for (pj = Tp [j] ; pj < Ap [j+1] ; pj++)
        {
                i = Ai [pj] ;

                /* A (i,j) is only in the lower part, not in upper.
                * add both A (i,j) and A (j,i) to the matrix A+A' */
                Len [i]++ ;
                Len [j]++ ;
                AMD_DEBUG3 (("    lower cleanup ("ID","ID") ("ID","ID")\n", i,j, j,i)) ;
        }
        }

        /* --------------------------------------------------------------------- */
        /* compute the symmetry of the nonzero pattern of A */
        /* --------------------------------------------------------------------- */

        /* Given a matrix A, the symmetry of A is:
        *   B = tril (spones (A), -1) + triu (spones (A), 1) ;
        *  sym = nnz (B & B') / nnz (B) ;
        *  or 1 if nnz (B) is zero. */
```

```
    if (nz == nzdiag)
    {
    sym = 1 ;
    }
    else
    {
    sym = (2 * (abip_float) nzboth) / ((abip_float) (nz - nzdiag)) ;
    }

    nzaat = 0 ;
    for (k = 0 ; k < n ; k++)
    {
    nzaat += Len [k] ;
    }

    AMD_DEBUG1 (("AMD nz in A+A', excluding diagonal (nzaat) = %g\n", (abip_float) nzaat)) ;
    AMD_DEBUG1 (("   nzboth: "ID" nz: "ID" nzdiag: "ID" symmetry: %g\n", nzboth, nz, nzdiag, sym))
;

    if (ABIPInfo != (abip_float *) ABIP_NULL)
    {
    ABIPInfo [AMD_STATUS] = AMD_OK ;
    ABIPInfo [AMD_N] = n ;
    ABIPInfo [AMD_NZ] = nz ;
    ABIPInfo [AMD_SYMMETRY] = sym ;          /* symmetry of pattern of A */
    ABIPInfo [AMD_NZDIAG] = nzdiag ;          /* nonzeros on diagonal of A */
    ABIPInfo [AMD_NZ_A_PLUS_AT] = nzaat ;        /* nonzeros in A+A' */
    }

    return (nzaat) ;
}
```

4.9 amd/amd_control.c File Reference

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

Functions

- GLOBAL void AMD_control (abip_float Control[ ])

4.9.1 Function Documentation

4.9.1.1 AMD_control()

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

Definition at line 18 of file amd_control.c.

4.10 amd_control.c

Go to the documentation of this file.

```
/* ========================================================================= */
/* === AMD_control ========================================================= */
/* ========================================================================= */

/* ------------------------------------------------------------------------- */
/* AMD, Copyright (c) Timothy A. Davis,                                   */
/* Patrick R. Amestoy, and Iain S. Duff.  See ../README.txt for License.     */
/* email: DrTimothyAldenDavis@gmail.com                                      */
/* ------------------------------------------------------------------------- */

/* User-callable.  Prints the control parameters for AMD.  See amd.h
 * for details.  If the Control array is not present, the defaults are
 * printed instead.
 */

#include "amd_internal.h"

GLOBAL void AMD_control
(
            abip_float Control [ ]
)
{
            abip_float alpha ;
            Int aggressive ;

            if (Control != (abip_float *) ABIP_NULL)
            {
               alpha = Control [AMD_DENSE] ;
               aggressive = Control [AMD_AGGRESSIVE] != 0 ;
            }
            else
            {
               alpha = AMD_DEFAULT_DENSE ;
               aggressive = AMD_DEFAULT_AGGRESSIVE ;
            }

            PRINTF (("\nAMD version %d.%d.%d, %s: approximate minimum degree ordering\n"
               "    dense row parameter: %g\n", AMD_MAIN_VERSION, AMD_SUB_VERSION, AMD_SUBSUB_VERSION, AMD_DATE, alpha)) ;

            if (alpha < 0)
            {
               PRINTF (("    no rows treated as dense\n")) ;
            }
            else
            {
               PRINTF ((
                           "    (rows with more than max (%g * sqrt (n), 16) entries are\n"
                           "    considered \"dense\", and placed last in output permutation)\n", alpha)) ;
            }

            if (aggressive)
            {
               PRINTF (("    aggressive absorption:  yes\n")) ;
            }
            else
            {
               PRINTF (("    aggressive absorption:  no\n")) ;
            }

            PRINTF (("    size of AMD integer: %d\n\n", sizeof (Int))) ;
}
```

4.11 amd/amd_defaults.c File Reference

`#include "amd_internal.h"`

Functions

- GLOBAL void AMD_defaults (abip_float Control[ ])

4.11.1 Function Documentation

4.11.1.1 AMD_defaults()

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

Definition at line 21 of file amd_defaults.c.

4.12 amd_defaults.c

Go to the documentation of this file.

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

4.13 amd/amd_dump.c File Reference

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

Functions

- GLOBAL void AMD_debug_init (char *s)
- GLOBAL void AMD_dump (Int n, Int Pe[ ], Int Iw[ ], Int Len[ ], Int iwlen, Int pfree, Int Nv[ ], Int Next[ ], Int Last[ ], Int Head[ ], Int Elen[ ], Int Degree[ ], Int W[ ], Int nel)

Variables

- GLOBAL Int AMD_debug = -999

4.13.1 Function Documentation

4.13.1.1 AMD_debug_init()

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

Definition at line 29 of file amd_dump.c.

4.13.1.2 AMD_dump()

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

Definition at line 58 of file amd_dump.c.

4.13.2 Variable Documentation

4.13.2.1 AMD_debug

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

Definition at line 21 of file amd_dump.c.

4.14 amd_dump.c

Go to the documentation of this file.

```
/* ========================================================================= */
/* === AMD_dump ============================================================ */
/* ========================================================================= */

/* ------------------------------------------------------------------------- */
/* AMD, Copyright (c) Timothy A. Davis,                                     */
/* Patrick R. Amestoy, and Iain S. Duff.  See ../README.txt for License.     */
/* email: DrTimothyAldenDavis@gmail.com                                      */
/* ------------------------------------------------------------------------- */

/* Debugging routines for AMD.  Not used if NDEBUG is not defined at compile-
 * time (the default).  See comments in amd_internal.h on how to enable
 * debugging.  Not user-callable.
 */

#include "amd_internal.h"

#ifndef NDEBUG

/* This global variable is present only when debugging */
GLOBAL Int AMD_debug = -999 ;      /* default is no debug printing */

/* ========================================================================= */
/* === AMD_debug_init ====================================================== */
/* ========================================================================= */

/* Sets the debug print level, by reading the file debug.amd (if it exists) */

GLOBAL void AMD_debug_init ( char *s )
{
        FILE *f ;
        f = fopen ("debug.amd", "r") ;

        if (f == (FILE *) ABIP_NULL)
        {
        AMD_debug = -999 ;
        }
        else
        {
        fscanf (f, ID, &AMD_debug) ;
        fclose (f) ;
        }

        if (AMD_debug >= 0)
        {
        printf ("%s: AMD_debug_init, D= "ID"\n", s, AMD_debug) ;
        }
}

/* ========================================================================= */
/* === AMD_dump ============================================================ */
/* ========================================================================= */

/* Dump AMD's data structure, except for the hash buckets.  This routine
 * cannot be called when the hash buckets are non-empty.
 */

GLOBAL void AMD_dump (
        Int n,              /* A is n-by-n */
        Int Pe [ ],         /* pe [0..n-1]: index in iw of start of row i */
        Int Iw [ ],         /* workspace of size iwlen, iwlen [0..pfree-1]
                  * holds the matrix on input */
        Int Len [ ],        /* len [0..n-1]: length for row i */
        Int iwlen,          /* length of iw */
        Int pfree,          /* iw [pfree ... iwlen-1] is empty on input */
        Int Nv [ ],         /* nv [0..n-1] */
        Int Next [ ],       /* next [0..n-1] */
        Int Last [ ],       /* last [0..n-1] */
        Int Head [ ],       /* head [0..n-1] */
        Int Elen [ ],       /* size n */
        Int Degree [ ],     /* size n */
        Int W [ ],          /* size n */
        Int nel
)
{
        Int i;
        Int pe;
        Int elen;
        Int nv;
        Int len;
        Int e;
        Int p;
```

```
Int k;
Int j;
Int deg;
Int w;
Int cnt;
Int ilast;

if (AMD_debug < 0) return ;
ASSERT (pfree <= iwlen) ;
AMD_DEBUG3 (("\nAMD dump, pfree: "ID"\n", pfree)) ;
for (i = 0 ; i < n ; i++)
{
pe = Pe [i] ;
elen = Elen [i] ;
nv = Nv [i] ;
len = Len [i] ;
w = W [i] ;

if (elen >= EMPTY)
{
        if (nv == 0)
        {
        AMD_DEBUG3 (("\nI "ID": nonprincipal:    ", i)) ;
        ASSERT (elen == EMPTY) ;

        if (pe == EMPTY)
        {
                AMD_DEBUG3 ((" dense node\n")) ;
                ASSERT (w == 1) ;
        }
        else
        {
                ASSERT (pe < EMPTY) ;
                AMD_DEBUG3 ((" i "ID" -> parent "ID"\n", i, FLIP (Pe[i])));
        }
        }
        else
        {
        AMD_DEBUG3 (("\nI "ID": active principal supervariable:\n",i));
        AMD_DEBUG3 (("   nv(i): "ID"  Flag: %d\n", nv, (nv < 0))) ;

        ASSERT (elen >= 0) ;
        ASSERT (nv > 0 && pe >= 0) ;
        p = pe ;
        AMD_DEBUG3 (("   e/s: ")) ;

        if (elen == 0) AMD_DEBUG3 ((" : ")) ;
        ASSERT (pe + len <= pfree) ;

        for (k = 0 ; k < len ; k++)
        {
                j = Iw [p] ;
                AMD_DEBUG3 (("  "ID"", j)) ;
                ASSERT (j >= 0 && j < n) ;

                if (k == elen-1) AMD_DEBUG3 ((" : ")) ;
                p++ ;
        }

        AMD_DEBUG3 (("\n")) ;
        }
}
else
{
        e = i ;

        if (w == 0)
        {
        AMD_DEBUG3 (("\nE "ID": absorbed element: w "ID"\n", e, w)) ;
        ASSERT (nv > 0 && pe < 0) ;
        AMD_DEBUG3 ((" e "ID" -> parent "ID"\n", e, FLIP (Pe [e]))) ;
        }
        else
        {
        AMD_DEBUG3 (("\nE "ID": unabsorbed element: w "ID"\n", e, w)) ;
        ASSERT (nv > 0 && pe >= 0) ;
        p = pe ;
        AMD_DEBUG3 ((" : ")) ;
        ASSERT (pe + len <= pfree) ;

        for (k = 0 ; k < len ; k++)
        {
                j = Iw [p] ;
                AMD_DEBUG3 (("  "ID"", j)) ;
                ASSERT (j >= 0 && j < n) ;
                p++ ;
        }
```

```
                AMD_DEBUG3 (("\n")) ;
                }
        }
        }

        /* this routine cannot be called when the hash buckets are non-empty */
        AMD_DEBUG3 (("\nDegree lists:\n")) ;
        if (nel >= 0)
        {
        cnt = 0 ;

        for (deg = 0 ; deg < n ; deg++)
        {
                if (Head [deg] == EMPTY) continue ;
                ilast = EMPTY ;
                AMD_DEBUG3 ((ID": \n", deg)) ;

                for (i = Head [deg] ; i != EMPTY ; i = Next [i])
                {
                AMD_DEBUG3 (("   "ID" : next "ID" last "ID" deg "ID"\n", i, Next [i], Last [i], Degree [i])) ;
                ASSERT (i >= 0 && i < n && ilast == Last [i] && deg == Degree [i]) ;
                cnt += Nv [i] ;
                ilast = i ;
                }

                AMD_DEBUG3 (("\n")) ;
        }

        ASSERT (cnt == n - nel) ;
        }
}

#endif
```

4.15 amd/amd_global.c File Reference

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

Macros

- #define ABIP_NULL 0

Variables

- void ∗(∗ amd_malloc)(size_t) = malloc
- void(∗ amd_free)(void ∗) = free
- void ∗(∗ amd_realloc)(void ∗, size_t) = realloc
- void ∗(∗ amd_calloc)(size_t, size_t) = calloc
- int(∗ amd_printf)(const char ∗,...) = ABIP_NULL

4.15.1 Macro Definition Documentation

4.15.1.1 ABIP_NULL

```
#define ABIP_NULL 0
```

Definition at line 20 of file amd_global.c.

4.15.2 Variable Documentation

4.15.2.1 amd_calloc

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

Definition at line 54 of file amd_global.c.

4.15.2.2 amd_free

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

Definition at line 52 of file amd_global.c.

4.15.2.3 amd_malloc

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

Definition at line 51 of file amd_global.c.

4.15.2.4 amd_printf

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

Definition at line 75 of file amd_global.c.

4.15.2.5 amd_realloc

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

Definition at line 53 of file amd_global.c.

4.16 amd_global.c

Go to the documentation of this file.

```
/* ========================================================================= */
/* === amd_global ========================================================== */
/* ========================================================================= */

/* ------------------------------------------------------------------------- */
/* AMD, Copyright (c) Timothy A. Davis,                          */
/* Patrick R. Amestoy, and Iain S. Duff.  See ../README.txt for License.     */
/* email: DrTimothyAldenDavis@gmail.com                                      */
/* ------------------------------------------------------------------------- */

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

#ifdef MATLAB_MEX_FILE
#include "mex.h"
#include "matrix.h"
#endif

#ifndef ABIP_NULL
#define ABIP_NULL 0
#endif

/* ========================================================================= */
/* === Default AMD memory manager ========================================== */
/* ========================================================================= */

/* The user can redefine these global pointers at run-time to change the memory
 * manager used by AMD.  AMD only uses malloc and free; realloc and calloc are
 * include for completeness, in case another package wants to use the same
 * memory manager as AMD.
 *
 * If compiling as a MATLAB mexFunction, the default memory manager is mxMalloc.
 * You can also compile AMD as a standard ANSI-C library and link a mexFunction
 * against it, and then redefine these pointers at run-time, in your
 * mexFunction.
 *
 * If -DNMALLOC is defined at compile-time, no memory manager is specified at
 * compile-time.  You must then define these functions at run-time, before
 * calling AMD, for AMD to work properly.
 */

#ifndef NMALLOC
#ifdef MATLAB_MEX_FILE
/* MATLAB mexFunction: */
void *(*amd_malloc) (size_t) = mxMalloc ;
void (*amd_free) (void *) = mxFree ;
void *(*amd_realloc) (void *, size_t) = mxRealloc ;
void *(*amd_calloc) (size_t, size_t) = mxCalloc ;
#else
/* standard ANSI-C: */
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 ;
#endif
#else
/* no memory manager defined at compile-time; you MUST define one at run-time */
void *(*amd_malloc) (size_t) = ABIP_NULL ;
void (*amd_free) (void *) = ABIP_NULL ;
void *(*amd_realloc) (void *, size_t) = ABIP_NULL ;
void *(*amd_calloc) (size_t, size_t) = ABIP_NULL ;
#endif

/* ========================================================================= */
/* === Default AMD printf routine ========================================== */
/* ========================================================================= */

/* The user can redefine this global pointer at run-time to change the printf
 * routine used by AMD.  If ABIP_NULL, no printing occurs.
 *
 * If -DNPRINT is defined at compile-time, stdio.h is not included.  Printing
 * can then be enabled at run-time by setting amd_printf to a non-ABIP_NULL function.
 */

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

4.17 amd/amd_info.c File Reference

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

Macros

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

Functions

- GLOBAL void AMD_info (abip_float ABIPInfo[ ])

4.17.1 Macro Definition Documentation

4.17.1.1 PRI

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

Definition at line 17 of file amd_info.c.

4.17.2 Function Documentation

4.17.2.1 AMD_info()

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

Definition at line 19 of file amd_info.c.

4.18 amd_info.c

Go to the documentation of this file.

```c
/* ========================================================================= */
/* === AMD_info ============================================================ */
/* ========================================================================= */

/* ------------------------------------------------------------------------- */
/* AMD, Copyright (c) Timothy A. Davis,                                      */
/* Patrick R. Amestoy, and Iain S. Duff.  See ../README.txt for License.     */
/* email: DrTimothyAldenDavis@gmail.com                                      */
/* ------------------------------------------------------------------------- */

/* User-callable.  Prints the output statistics for AMD.  See amd.h
 * for details.  If the ABIPInfo array is not present, nothing is printed.
 */

#include "amd_internal.h"

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

GLOBAL void AMD_info
(
            abip_float ABIPInfo [ ]
)
{
            abip_float n;
            abip_float ndiv;
            abip_float nmultsubs_ldl;
            abip_float nmultsubs_lu;
            abip_float lnz;
            abip_float lnzd;

            PRINTF (("\nAMD version %d.%d.%d, %s, results:\n", AMD_MAIN_VERSION, AMD_SUB_VERSION, AMD_SUBSUB_VERSION, AMD_DATE)) ;

            if (!ABIPInfo)
            {
               return ;
            }

            n = ABIPInfo [AMD_N] ;
            ndiv = ABIPInfo [AMD_NDIV] ;
            nmultsubs_ldl = ABIPInfo [AMD_NMULTSUBS_LDL] ;
            nmultsubs_lu = ABIPInfo [AMD_NMULTSUBS_LU] ;
            lnz = ABIPInfo [AMD_LNZ] ;
            lnzd = (n >= 0 && lnz >= 0) ? (n + lnz) : (-1) ;

            /* AMD return status */
            PRINTF (("    status: ")) ;
            if (ABIPInfo [AMD_STATUS] == AMD_OK)
            {
               PRINTF (("OK\n")) ;
            }
            else if (ABIPInfo [AMD_STATUS] == AMD_OUT_OF_MEMORY)
            {
               PRINTF (("out of memory\n")) ;
            }
            else if (ABIPInfo [AMD_STATUS] == AMD_INVALID)
            {
               PRINTF (("invalid matrix\n")) ;
            }
            else if (ABIPInfo [AMD_STATUS] == AMD_OK_BUT_JUMBLED)
            {
               PRINTF (("OK, but jumbled\n")) ;
            }
            else
            {
               PRINTF (("unknown\n")) ;
            }

            /* statistics about the input matrix */
            PRI ("    n, dimension of A:                                  %.20g\n", n);
            PRI ("    nz, number of nonzeros in A:                        %.20g\n", ABIPInfo [AMD_NZ]) ;
            PRI ("    symmetry of A:                                      %.4f\n", ABIPInfo [AMD_SYMMETRY]) ;
            PRI ("    number of nonzeros on diagonal:                     %.20g\n", ABIPInfo [AMD_NZDIAG]) ;
            PRI ("    nonzeros in pattern of A+A' (excl. diagonal):       %.20g\n", ABIPInfo [AMD_NZ_A_PLUS_AT]) ;
            PRI ("    # dense rows/columns of A+A':                       %.20g\n", ABIPInfo [AMD_NDENSE]) ;

            /* statistics about AMD's behavior  */
```

```
            PRI ("    memory used, in bytes:                          %.20g\n", ABIPInfo
    [AMD_MEMORY]) ;
            PRI ("    # of memory compactions:                        %.20g\n", ABIPInfo
    [AMD_NCMPA]) ;

            /* statistics about the ordering quality */
            PRINTF (("\n"
           "    The following approximate statistics are for a subsequent\n"
           "    factorization of A(P,P) + A(P,P)'.  They are slight upper\n"
           "    bounds if there are no dense rows/columns in A+A', and become\n"
           "    looser if dense rows/columns exist.\n\n")) ;

            PRI ("    nonzeros in L (excluding diagonal):             %.20g\n", lnz) ;
            PRI ("    nonzeros in L (including diagonal):             %.20g\n", lnzd) ;
            PRI ("    # divide operations for LDL' or LU:             %.20g\n", ndiv) ;
            PRI ("    # multiply-subtract operations for LDL':        %.20g\n", nmultsubs_ldl) ;
            PRI ("    # multiply-subtract operations for LU:          %.20g\n", nmultsubs_lu) ;
            PRI ("    max nz. in any column of L (incl. diagonal):    %.20g\n", ABIPInfo
    [AMD_DMAX]) ;

            /* total flop counts for various factorizations */

            if (n >= 0 && ndiv >= 0 && nmultsubs_ldl >= 0 && nmultsubs_lu >= 0)
            {
               PRINTF (("\n"
                        "    chol flop count for real A, sqrt counted as 1 flop: %.20g\n"
                        "    LDL' flop count for real A:                         %.20g\n"
                        "    LDL' flop count for complex A:                      %.20g\n"
                        "    LU flop count for real A (with no pivoting):        %.20g\n"
                        "    LU flop count for complex A (with no pivoting):     %.20g\n\n",
                        n + ndiv + 2*nmultsubs_ldl,
                        ndiv + 2*nmultsubs_ldl,
                        9*ndiv + 8*nmultsubs_ldl,
                        ndiv + 2*nmultsubs_lu,
                        9*ndiv + 8*nmultsubs_lu)) ;
            }
}
```

4.19 amd/amd_internal.h File Reference

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

Macros

- #define EMPTY (-1)
- #define FLIP(i) (-(i)-2)
- #define UNFLIP(i) ((i < EMPTY) ? FLIP (i) : (i))
- #define MAX(a, b) (((a) > (b)) ? (a) : (b))
- #define MIN(a, b) (((a) < (b)) ? (a) : (b))
- #define IMPLIES(p, q) (!(p) || (q))
- #define TRUE (1)
- #define FALSE (0)
- #define PRIVATE static
- #define GLOBAL
- #define EMPTY (-1)
- #define ABIP_NULL 0
- #define SIZE_T_MAX ((size_t) (-1))
- #define Int int
- #define ID "%d"
- #define Int_MAX INT_MAX

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

Functions

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

4.19.1 Macro Definition Documentation

4.19.1.1 ABIP_NULL

```
#define ABIP_NULL 0
```

Definition at line 138 of file amd_internal.h.

4.19.1.2 AMD_1

```
#define AMD_1 amd_1
```

Definition at line 187 of file amd_internal.h.

4.19.1.3 AMD_2

```
#define AMD_2 amd_2
```

Definition at line 188 of file amd_internal.h.

4.19.1.4 AMD_aat

```
#define AMD_aat amd_aat
```

Definition at line 190 of file amd_internal.h.

4.19.1.5 AMD_control

```
#define AMD_control amd_control
```

Definition at line 185 of file amd_internal.h.

4.19.1.6 AMD_debug

```
#define AMD_debug amd_debug
```

Definition at line 194 of file amd_internal.h.

4.19.1.7 AMD_DEBUG0

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

Definition at line 328 of file amd_internal.h.

4.19.1.8 AMD_DEBUG1

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

Definition at line 329 of file amd_internal.h.

4.19.1.9 AMD_DEBUG2

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

Definition at line 330 of file amd_internal.h.

4.19.1.10 AMD_DEBUG3

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

Definition at line 331 of file amd_internal.h.

4.19.1.11 AMD_DEBUG4

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

Definition at line 332 of file amd_internal.h.

4.19.1.12 AMD_debug_init

```
#define AMD_debug_init amd_debug_init
```

Definition at line 195 of file amd_internal.h.

4.19.1.13 AMD_defaults

```
#define AMD_defaults amd_defaults
```

Definition at line 184 of file amd_internal.h.

4.19.1.14 AMD_dump

```
#define AMD_dump amd_dump
```

Definition at line 193 of file amd_internal.h.

4.19.1.15 AMD_info

```
#define AMD_info amd_info
```

Definition at line 186 of file amd_internal.h.

4.19.1.16 AMD_order

```
#define AMD_order amd_order
```

Definition at line 183 of file amd_internal.h.

4.19.1.17 AMD_post_tree

```
#define AMD_post_tree amd_post_tree
```

Definition at line 192 of file amd_internal.h.

4.19.1.18 AMD_postorder

```
#define AMD_postorder amd_postorder
```

Definition at line 191 of file amd_internal.h.

4.19.1.19 AMD_preprocess

```
#define AMD_preprocess amd_preprocess
```

Definition at line 196 of file amd_internal.h.

4.19.1.20 AMD_valid

```
#define AMD_valid amd_valid
```

Definition at line 189 of file amd_internal.h.

4.19.1.21 ASSERT

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

Definition at line 327 of file amd_internal.h.

4.19.1.22 EMPTY [1/2]

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

Definition at line 129 of file amd_internal.h.

4.19.1.23 EMPTY [2/2]

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

Definition at line 129 of file amd_internal.h.

4.19.1.24 FALSE

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

Definition at line 126 of file amd_internal.h.

4.19.1.25 FLIP

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

Definition at line 106 of file amd_internal.h.

4.19.1.26 GLOBAL

```
#define GLOBAL
```

Definition at line 128 of file amd_internal.h.

4.19.1.27 ID

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

Definition at line 180 of file amd_internal.h.

4.19.1.28 IMPLIES

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

Definition at line 114 of file amd_internal.h.

4.19.1.29 Int

```
#define Int int
```

Definition at line 179 of file amd_internal.h.

4.19.1.30 Int_MAX

```
#define Int_MAX INT_MAX
```

Definition at line 181 of file amd_internal.h.

4.19.1.31 MAX

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

Definition at line 110 of file amd_internal.h.

4.19.1.32 MIN

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

Definition at line 111 of file amd_internal.h.

4.19.1.33 PRINTF

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

Definition at line 205 of file amd_internal.h.

4.19.1.34 PRIVATE

```
#define PRIVATE static
```

Definition at line 127 of file amd_internal.h.

4.19.1.35 SIZE_T_MAX

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

Definition at line 146 of file amd_internal.h.

4.19.1.36 TRUE

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

Definition at line 125 of file amd_internal.h.

4.19.1.37 UNFLIP

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

Definition at line 107 of file amd_internal.h.

4.19.2 Function Documentation

4.19.2.1 AMD_1()

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

Definition at line 29 of file amd_1.c.

4.19.2.2 AMD_aat()

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

Definition at line 20 of file amd_aat.c.

4.19.2.3 AMD_post_tree()

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

4.19.2.4 AMD_postorder()

```
GLOBAL void AMD_postorder (
            Int nn,
            Int Parent[ ],
            Int Npiv[ ],
            Int Fsize[ ],
            Int Order[ ],
            Int Child[ ],
            Int Sibling[ ],
            Int Stack[ ] )
```

Definition at line 15 of file amd_postorder.c.

4.19.2.5 AMD_preprocess()

```
GLOBAL void AMD_preprocess (
            Int n,
            const Int Ap[],
            const Int Ai[],
            Int Rp[],
            Int Ri[],
            Int W[],
            Int Flag[] )
```

Definition at line 29 of file amd_preprocess.c.

4.20 amd_internal.h

Go to the documentation of this file.

```
/* ========================================================================= */
/* === amd_internal.h ====================================================== */
/* ========================================================================= */

/* ------------------------------------------------------------------------- */
/* AMD, Copyright (c) Timothy A. Davis,                                      */
/* Patrick R. Amestoy, and Iain S. Duff.  See ../README.txt for License.     */
/* email: DrTimothyAldenDavis@gmail.com                                      */
/* ------------------------------------------------------------------------- */

/* This file is for internal use in AMD itself, and does not normally need to
 * be included in user code (it is included in UMFPACK, however).   All others
 * should use amd.h instead.
 */

/* ========================================================================= */
/* === NDEBUG ============================================================== */
/* ========================================================================= */

/*
 * Turning on debugging takes some work (see below).   If you do not edit this
 * file, then debugging is always turned off, regardless of whether or not
 * -DNDEBUG is specified in your compiler options.
 *
 * If AMD is being compiled as a mexFunction, then MATLAB_MEX_FILE is defined,
 * and mxAssert is used instead of assert.  If debugging is not enabled, no
 * MATLAB include files or functions are used.  Thus, the AMD library libamd.a
 * can be safely used in either a stand-alone C program or in another
 * mexFunction, without any change.
 */

/*
    AMD will be exceedingly slow when running in debug mode.  The next three
    lines ensure that debugging is turned off.
*/
#ifndef NDEBUG
#define NDEBUG
#endif

/*
    To enable debugging, uncomment the following line:
#undef NDEBUG
*/

/* ------------------------------------------------------------------------- */
/* ANSI include files */
/* ------------------------------------------------------------------------- */

/* from stdlib.h:  size_t, malloc, free, realloc, and calloc */
#include <stdlib.h>

#if !defined(NPRINT) || !defined(NDEBUG)
/* from stdio.h:  printf.  Not included if NPRINT is defined at compile time.
 * fopen and fscanf are used when debugging. */
#include <stdio.h>
#endif

/* from limits.h:  INT_MAX and LONG_MAX */
#include <limits.h>
```

```c

/* from math.h: sqrt */
#include <math.h>

/* ------------------------------------------------------------------------- */
/* MATLAB include files (only if being used in or via MATLAB) */
/* ------------------------------------------------------------------------- */

#ifdef MATLAB_MEX_FILE
#include "matrix.h"
#include "mex.h"
#endif

/* ------------------------------------------------------------------------- */
/* basic definitions */
/* ------------------------------------------------------------------------- */

#ifdef FLIP
#undef FLIP
#endif

#ifdef MAX
#undef MAX
#endif

#ifdef MIN
#undef MIN
#endif

#ifdef EMPTY
#undef EMPTY
#endif

#ifdef GLOBAL
#undef GLOBAL
#endif

#ifdef PRIVATE
#undef PRIVATE
#endif

/* FLIP is a "negation about -1", and is used to mark an integer i that is
 * normally non-negative.  FLIP (EMPTY) is EMPTY.  FLIP of a number > EMPTY
 * is negative, and FLIP of a number < EMTPY is positive.  FLIP (FLIP (i)) = i
 * for all integers i.  UNFLIP (i) is >= EMPTY. */
#define EMPTY (-1)
#define FLIP(i) (-(i)-2)
#define UNFLIP(i) ((i < EMPTY) ? FLIP (i) : (i))

/* for integer MAX/MIN, or for doubles when we don't care how NaN's behave: */
#define MAX(a,b) (((a) > (b)) ? (a) : (b))
#define MIN(a,b) (((a) < (b)) ? (a) : (b))

/* logical expression of p implies q: */
#define IMPLIES(p,q) (!(p) || (q))

/* Note that the IBM RS 6000 xlc predefines TRUE and FALSE in <types.h>. */
/* The Compaq Alpha also predefines TRUE and FALSE. */
#ifdef TRUE
#undef TRUE
#endif
#ifdef FALSE
#undef FALSE
#endif

#define TRUE (1)
#define FALSE (0)
#define PRIVATE static
#define GLOBAL
#define EMPTY (-1)

/* Note that Linux's gcc 2.96 defines NULL as ((void *) 0), but other */
/* compilers (even gcc 2.95.2 on Solaris) define NULL as 0 or (0).  We */
/* need to use the ANSI standard value of 0. */
#ifdef ABIP_NULL
#undef ABIP_NULL
#endif

#define ABIP_NULL 0

/* largest value of size_t */
#ifndef SIZE_T_MAX
#ifdef SIZE_MAX
/* C99 only */
#define SIZE_T_MAX SIZE_MAX
#else
#define SIZE_T_MAX ((size_t) (-1))
```

```
#endif
#endif

/* ------------------------------------------------------------------------- */
/* integer type for AMD: int or SuiteSparse_long */
/* ------------------------------------------------------------------------- */

#include "amd.h"

#if defined (DLONG) || defined (ZLONG)

#define Int SuiteSparse_long
#define ID  SuiteSparse_long_id
#define Int_MAX SuiteSparse_long_max

#define AMD_order amd_l_order
#define AMD_defaults amd_l_defaults
#define AMD_control amd_l_control
#define AMD_info amd_l_info
#define AMD_1 amd_l1
#define AMD_2 amd_l2
#define AMD_valid amd_l_valid
#define AMD_aat amd_l_aat
#define AMD_postorder amd_l_postorder
#define AMD_post_tree amd_l_post_tree
#define AMD_dump amd_l_dump
#define AMD_debug amd_l_debug
#define AMD_debug_init amd_l_debug_init
#define AMD_preprocess amd_l_preprocess

#else

#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

#endif

/* ========================================================================= */
/* === PRINTF macro ======================================================== */
/* ========================================================================= */

/* All output goes through the PRINTF macro.  */
#define PRINTF(params) { if (amd_printf != ABIP_NULL) (void) amd_printf params ; }

/* ------------------------------------------------------------------------- */
/* AMD routine definitions (not user-callable) */
/* ------------------------------------------------------------------------- */

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 [ ]

            #ifndef NDEBUG
            , Int nn
            #endif
) ;

GLOBAL void AMD_preprocess
(
            Int n,
            const Int Ap [ ],
            const Int Ai [ ],
            Int Rp [ ],
            Int Ri [ ],
            Int W [ ],
            Int Flag [ ]
) ;

/* ------------------------------------------------------------------------- */
/* debugging definitions */
/* ------------------------------------------------------------------------- */

#ifndef NDEBUG

/* from assert.h:  assert macro */
#include <assert.h>

#ifndef EXTERN
#define EXTERN extern
#endif

EXTERN Int AMD_debug ;

GLOBAL void AMD_debug_init ( char *s ) ;

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
) ;

#ifdef ASSERT
#undef ASSERT
#endif

/* Use mxAssert if AMD is compiled into a mexFunction */
#ifdef MATLAB_MEX_FILE
#define ASSERT(expression) (mxAssert ((expression), ""))
#else
#define ASSERT(expression) (assert (expression))
#endif

#define AMD_DEBUG0(params) { PRINTF (params) ; }
#define AMD_DEBUG1(params) { if (AMD_debug >= 1) PRINTF (params) ; }
#define AMD_DEBUG2(params) { if (AMD_debug >= 2) PRINTF (params) ; }
```

```
#define AMD_DEBUG3(params) { if (AMD_debug >= 3) PRINTF (params) ; }
#define AMD_DEBUG4(params) { if (AMD_debug >= 4) PRINTF (params) ; }

#else

/* no debugging */
#define ASSERT(expression)
#define AMD_DEBUG0(params)
#define AMD_DEBUG1(params)
#define AMD_DEBUG2(params)
#define AMD_DEBUG3(params)
#define AMD_DEBUG4(params)

#endif
```

4.21 amd/amd_order.c File Reference

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

Functions

- GLOBAL Int AMD_order (Int n, const Int Ap[ ], const Int Ai[ ], Int P[ ], abip_float Control[ ], abip_float ABIPInfo[ ])

4.21.1 Function Documentation

4.21.1.1 AMD_order()

```
GLOBAL Int AMD_order (
            Int n,
            const Int Ap[ ],
            const Int Ai[ ],
            Int P[ ],
            abip_float Control[ ],
            abip_float ABIPInfo[ ] )
```

Definition at line 21 of file amd_order.c.

4.22 amd_order.c

Go to the documentation of this file.

```
/* ========================================================================= */
/* === AMD_order =========================================================== */
/* ========================================================================= */

/* ------------------------------------------------------------------------- */
/* AMD, Copyright (c) Timothy A. Davis,                                     */
/* Patrick R. Amestoy, and Iain S. Duff.  See ../README.txt for License.     */
/* email: DrTimothyAldenDavis@gmail.com                                      */
/* ------------------------------------------------------------------------- */

/* User-callable AMD minimum degree ordering routine.  See amd.h for
 * documentation.
 */
```

```c

#include "amd_internal.h"

/* ========================================================================= */
/* === AMD_order =========================================================== */
/* ========================================================================= */

GLOBAL Int AMD_order
(
            Int n,
            const Int Ap [ ],
            const Int Ai [ ],
            Int P [ ],
            abip_float Control [ ],
            abip_float ABIPInfo [ ]
)
{
            Int *Len;
            Int *S;
            Int nz;
            Int i;
            Int *Pinv;
            Int info;
            Int status;

            Int *Rp;
            Int *Ri;
            Int *Cp;
            Int *Ci;
            Int ok;

            size_t nzaat;
            size_t slen;

            abip_float mem = 0 ;

            #ifndef NDEBUG
            AMD_debug_init ("amd") ;
            #endif

            /* clear the ABIPInfo array, if it exists */
            info = ABIPInfo != (abip_float *) ABIP_NULL ;

            if (info)
            {
               for (i = 0 ; i < AMD_INFO ; i++)
               {
                           ABIPInfo [i] = EMPTY ;
               }
               ABIPInfo [AMD_N] = n ;
               ABIPInfo [AMD_STATUS] = AMD_OK ;
            }

            /* make sure inputs exist and n is >= 0 */
            if (Ai == (Int *) ABIP_NULL || Ap == (Int *) ABIP_NULL || P == (Int *) ABIP_NULL || n < 0)
            {
               if (info) ABIPInfo [AMD_STATUS] = AMD_INVALID ;
               return (AMD_INVALID) ;        /* arguments are invalid */
            }

            if (n == 0)
            {
               return (AMD_OK) ;      /* n is 0 so there's nothing to do */
            }

            nz = Ap [n] ;

            if (info)
            {
               ABIPInfo [AMD_NZ] = nz ;
            }

            if (nz < 0)
            {
               if (info) ABIPInfo [AMD_STATUS] = AMD_INVALID ;
               return (AMD_INVALID) ;
            }

            /* check if n or nz will cause size_t overflow */
            if (((size_t) n) >= SIZE_T_MAX / sizeof (Int) || ((size_t) nz) >= SIZE_T_MAX / sizeof (Int))
            {
               if (info) ABIPInfo [AMD_STATUS] = AMD_OUT_OF_MEMORY ;
               return (AMD_OUT_OF_MEMORY) ;        /* problem too large */
            }

            /* check the input matrix:  AMD_OK, AMD_INVALID, or AMD_OK_BUT_JUMBLED */
```

```
            status = AMD_valid (n, n, Ap, Ai) ;

            if (status == AMD_INVALID)
            {
               if (info) ABIPInfo [AMD_STATUS] = AMD_INVALID ;
               return (AMD_INVALID) ;          /* matrix is invalid */
            }

            /* allocate two size-n integer workspaces */
            Len  = amd_malloc (n * sizeof (Int)) ;
            Pinv = amd_malloc (n * sizeof (Int)) ;
            mem += n ;
            mem += n ;

            if (!Len || !Pinv)
            {
               /* :: out of memory :: */
               amd_free (Len) ;
               amd_free (Pinv) ;
               if (info) ABIPInfo [AMD_STATUS] = AMD_OUT_OF_MEMORY ;
               return (AMD_OUT_OF_MEMORY) ;
            }

            if (status == AMD_OK_BUT_JUMBLED)
            {
               /* sort the input matrix and remove duplicate entries */
               AMD_DEBUG1 (("Matrix is jumbled\n")) ;
               Rp = amd_malloc ((n+1) * sizeof (Int));
               Ri = amd_malloc (nz * sizeof (Int));
               mem += (n+1) ;
               mem += MAX (nz,1) ;

                        if (!Rp || !Ri)
               {
                           /* :: out of memory :: */
                           amd_free (Rp) ;
                           amd_free (Ri) ;
                           amd_free (Len) ;
                           amd_free (Pinv) ;

                                    if (info) ABIPInfo [AMD_STATUS] = AMD_OUT_OF_MEMORY ;
                           return (AMD_OUT_OF_MEMORY) ;
               }

               /* use Len and Pinv as workspace to create R = A' */
               AMD_preprocess (n, Ap, Ai, Rp, Ri, Len, Pinv) ;
               Cp = Rp ;
               Ci = Ri ;
            }
            else
            {
               /* order the input matrix as-is.  No need to compute R = A' first */
               Rp = ABIP_NULL ;
               Ri = ABIP_NULL ;
               Cp = (Int *) Ap ;
               Ci = (Int *) Ai ;
            }

            /* --------------------------------------------------------------------- */
            /* determine the symmetry and count off-diagonal nonzeros in A+A' */
            /* --------------------------------------------------------------------- */

            nzaat = AMD_aat (n, Cp, Ci, Len, P, ABIPInfo) ;
            AMD_DEBUG1 (("nzaat: %g\n", (abip_float) nzaat)) ;
            ASSERT ((MAX (nz-n, 0) <= nzaat) && (nzaat <= 2 * (size_t) nz)) ;

            /* --------------------------------------------------------------------- */
            /* allocate workspace for matrix, elbow room, and 6 size-n vectors */
            /* --------------------------------------------------------------------- */

            S = ABIP_NULL ;
            slen = nzaat ;                          /* space for matrix */
            ok = ((slen + nzaat/5) >= slen) ;     /* check for size_t overflow */
            slen += nzaat/5 ;               /* add elbow room */

            for (i = 0 ; ok && i < 7 ; i++)
            {
               ok = ((slen + n) > slen) ;    /* check for size_t overflow */
               slen += n ;            /* size-n elbow room, 6 size-n work */
            }

            mem += slen ;
            ok = ok && (slen < SIZE_T_MAX / sizeof (Int)) ;      /* check for overflow */
            ok = ok && (slen < Int_MAX) ;                         /* S[i] for Int i must be OK */

            if (ok)
```

```
            {
               S = amd_malloc (slen * sizeof (Int)) ;
            }

            AMD_DEBUG1 (("slen %g\n", (abip_float) slen)) ;

            if (!S)
            {
               /* :: out of memory :: (or problem too large) */
               amd_free (Rp) ;
               amd_free (Ri) ;
               amd_free (Len) ;
               amd_free (Pinv) ;
               if (info) ABIPInfo [AMD_STATUS] = AMD_OUT_OF_MEMORY ;
               return (AMD_OUT_OF_MEMORY) ;
            }

            if (info)
            {
               /* memory usage, in bytes. */
               ABIPInfo [AMD_MEMORY] = mem * sizeof (Int) ;
            }

            /* --------------------------------------------------------------------- */
            /* order the matrix */
            /* --------------------------------------------------------------------- */

            AMD_1 (n, Cp, Ci, P, Pinv, Len, slen, S, Control, ABIPInfo) ;

            /* --------------------------------------------------------------------- */
            /* free the workspace */
            /* --------------------------------------------------------------------- */

            amd_free (Rp) ;
            amd_free (Ri) ;
            amd_free (Len) ;
            amd_free (Pinv) ;
            amd_free (S) ;

            if (info) ABIPInfo [AMD_STATUS] = status ;
            return (status) ;       /* successful ordering */
}
```

4.23 amd/amd_post_tree.c File Reference

`#include "amd_internal.h"`

Functions

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

4.23.1 Function Documentation

4.23.1.1 AMD_post_tree()

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

Definition at line 15 of file amd_post_tree.c.

4.24 amd_post_tree.c

Go to the documentation of this file.

```
/* ========================================================================= */
/* === AMD_post_tree ======================================================= */
/* ========================================================================= */

/* ------------------------------------------------------------------------- */
/* AMD, Copyright (c) Timothy A. Davis,                                    */
/* Patrick R. Amestoy, and Iain S. Duff.  See ../README.txt for License.     */
/* email: DrTimothyAldenDavis@gmail.com                                      */
/* ------------------------------------------------------------------------- */

/* Post-ordering of a supernodal elimination tree.  */

#include "amd_internal.h"

GLOBAL Int AMD_post_tree
(
        Int root,            /* root of the tree */
        Int k,               /* start numbering at k */
        Int Child [ ],          /* input argument of size nn, undefined on
                 * output.  Child [i] is the head of a link
                 * list of all nodes that are children of node
                 * i in the tree. */
        const Int Sibling [ ],    /* input argument of size nn, not modified.
                 * If f is a node in the link list of the
                 * children of node i, then Sibling [f] is the
                 * next child of node i.
                 */
        Int Order [ ],          /* output order, of size nn.  Order [i] = k
                 * if node i is the kth node of the reordered
                 * tree. */
        Int Stack [ ]           /* workspace of size nn */
    #ifndef NDEBUG
        , Int nn             /* nodes are in the range 0..nn-1. */
    #endif
)
{
        Int f;
        Int head;
        Int h;
        Int i;

    #if 0
            /* --------------------------------------------------------------------- */
            /* recursive version (Stack [ ] is not used): */
            /* --------------------------------------------------------------------- */

            /* this is simple, but can caouse stack overflow if nn is large */
            i = root ;
            for (f = Child [i] ; f != EMPTY ; f = Sibling [f])
            {
            k = AMD_post_tree (f, k, Child, Sibling, Order, Stack, nn) ;
            }
            Order [i] = k++ ;
            return (k) ;
    #endif

        /* --------------------------------------------------------------------- */
        /* non-recursive version, using an explicit stack */
        /* --------------------------------------------------------------------- */

        /* push root on the stack */
        head = 0 ;
        Stack [0] = root ;

        while (head >= 0)
        {
        /* get head of stack */
        ASSERT (head < nn) ;
        i = Stack [head] ;
        AMD_DEBUG1 (("head of stack "ID" \n", i)) ;
        ASSERT (i >= 0 && i < nn) ;

        if (Child [i] != EMPTY)
        {
                /* the children of i are not yet ordered */
                /* push each child onto the stack in reverse order */
                /* so that small ones at the head of the list get popped first */
                /* and the biggest one at the end of the list gets popped last */
                for (f = Child [i] ; f != EMPTY ; f = Sibling [f])
                {
                head++ ;
                ASSERT (head < nn) ;
```

```
                ASSERT (f >= 0 && f < nn) ;
                }

                h = head ;
                ASSERT (head < nn) ;

                for (f = Child [i] ; f != EMPTY ; f = Sibling [f])
                {
                ASSERT (h > 0) ;
                Stack [h--] = f ;
                AMD_DEBUG1 (("push "ID" on stack\n", f)) ;
                ASSERT (f >= 0 && f < nn) ;
                }

                ASSERT (Stack [h] == i) ;

                /* delete child list so that i gets ordered next time we see it */
                Child [i] = EMPTY ;
        }
        else
        {
                /* the children of i (if there were any) are already ordered */
                /* remove i from the stack and order it.  Front i is kth front */
                head-- ;
                AMD_DEBUG1 (("pop "ID" order "ID"\n", i, k)) ;
                Order [i] = k++ ;
                ASSERT (k <= nn) ;
        }

        #ifndef NDEBUG
        AMD_DEBUG1 (("\nStack:")) ;

        for (h = head ; h >= 0 ; h--)
        {
                Int j = Stack [h] ;
                AMD_DEBUG1 ((" "ID, j)) ;
                ASSERT (j >= 0 && j < nn) ;
        }

        AMD_DEBUG1 (("\n\n")) ;
        ASSERT (head < nn) ;
        #endif
    }
        return (k) ;
}
```

4.25 amd/amd_postorder.c File Reference

`#include "amd_internal.h"`

Functions

- GLOBAL void AMD_postorder (Int nn, Int Parent[ ], Int Nv[ ], Int Fsize[ ], Int Order[ ], Int Child[ ], Int Sibling[ ], Int Stack[ ])

4.25.1 Function Documentation

4.25.1.1 AMD_postorder()

```
GLOBAL void AMD_postorder (
            Int nn,
            Int Parent[ ],
            Int Nv[ ],
            Int Fsize[ ],
            Int Order[ ],
            Int Child[ ],
            Int Sibling[ ],
            Int Stack[ ] )
```

Definition at line 15 of file amd_postorder.c.

4.26 amd_postorder.c

Go to the documentation of this file.

```
00001 /* ========================================================================= */
00002 /* === AMD_postorder ======================================================= */
00003 /* ========================================================================= */
00004
00005 /* ------------------------------------------------------------------------- */
00006 /* AMD, Copyright (c) Timothy A. Davis,                                     */
00007 /* Patrick R. Amestoy, and Iain S. Duff.  See ../README.txt for License.     */
00008 /* email: DrTimothyAldenDavis@gmail.com                                      */
00009 /* ------------------------------------------------------------------------- */
00010
00011 /* Perform a postordering (via depth-first search) of an assembly tree. */
00012
00013 #include "amd_internal.h"
00014
00015 GLOBAL void AMD_postorder
00016 (
00017         /* inputs, not modified on output: */
00018         Int nn,          /* nodes are in the range 0..nn-1 */
00019         Int Parent [ ],      /* Parent [j] is the parent of j, or EMPTY if root */
00020         Int Nv [ ],      /* Nv [j] > 0 number of pivots represented by node j,
00021                  * or zero if j is not a node. */
00022         Int Fsize [ ],       /* Fsize [j]: size of node j */
00023
00024         /* output, not defined on input: */
00025         Int Order [ ],       /* output post-order */
00026
00027         /* workspaces of size nn: */
00028         Int Child [ ],
00029         Int Sibling [ ],
00030         Int Stack [ ]
00031 )
00032 {
00033         Int i;
00034         Int j;
00035         Int k;
00036
00037         Int parent;
00038         Int frsize;
00039         Int f;
00040         Int fprev;
00041         Int maxfrsize;
00042         Int bigfprev;
00043         Int bigf;
00044         Int fnext;
00045
00046         for (j = 0 ; j < nn ; j++)
00047         {
00048         Child [j] = EMPTY ;
00049         Sibling [j] = EMPTY ;
00050         }
00051
00052         /* --------------------------------------------------------------------- */
00053         /* place the children in link lists - bigger elements tend to be last */
00054         /* --------------------------------------------------------------------- */
00055
00056         for (j = nn-1 ; j >= 0 ; j--)
00057         {
```

```
        if (Nv [j] > 0)
        {
                /* this is an element */
                parent = Parent [j] ;
                if (parent != EMPTY)
                {
                /* place the element in link list of the children its parent */
                /* bigger elements will tend to be at the end of the list */
                Sibling [j] = Child [parent] ;
                Child [parent] = j ;
                }
        }
        }

    #ifndef NDEBUG

        Int nels;
        Int ff;
        Int nchild;
    AMD_DEBUG1 (("\n\n================================ AMD_postorder:\n"));
    nels = 0;

    for (j = 0 ; j < nn ; j++)
    {
            if (Nv [j] > 0)
            {
            AMD_DEBUG1 (( ""ID" :  nels "ID" npiv "ID" size "ID" parent "ID" maxfr "ID"\n", j, nels, Nv [j], Fsize [j], Parent [j], Fsize [j])) ;

            /* this is an element */
            /* dump the link list of children */
            nchild = 0 ;
            AMD_DEBUG1 (("    Children: ")) ;

            for (ff = Child [j] ; ff != EMPTY ; ff = Sibling [ff])
            {
                    AMD_DEBUG1 ((ID" ", ff)) ;
                    ASSERT (Parent [ff] == j) ;
                    nchild++ ;
                    ASSERT (nchild < nn) ;
            }

            AMD_DEBUG1 (("\n")) ;
            parent = Parent [j] ;

            if (parent != EMPTY)
            {
                    ASSERT (Nv [parent] > 0) ;
            }

            nels++ ;
            }
    }

        AMD_DEBUG1 (("\n\nGo through the children of each node, and put\n" "the biggest child last in each list:\n")) ;

    #endif

        /* --------------------------------------------------------------------- */
        /* place the largest child last in the list of children for each node */
        /* --------------------------------------------------------------------- */

        for (i = 0 ; i < nn ; i++)
        {
        if (Nv [i] > 0 && Child [i] != EMPTY)
        {

            #ifndef NDEBUG

                Int nchild ;
                AMD_DEBUG1 (("Before partial sort, element "ID"\n", i)) ;
                nchild = 0 ;

                for (f = Child [i] ; f != EMPTY ; f = Sibling [f])
                {
                ASSERT (f >= 0 && f < nn) ;
                AMD_DEBUG1 (("      f: "ID"  size: "ID"\n", f, Fsize [f])) ;
                nchild++ ;
                ASSERT (nchild <= nn) ;
                }

            #endif

                /* find the biggest element in the child list */
                fprev = EMPTY ;
                maxfrsize = EMPTY ;
```

```
            bigfprev = EMPTY ;
            bigf = EMPTY ;

            for (f = Child [i] ; f != EMPTY ; f = Sibling [f])
            {
            ASSERT (f >= 0 && f < nn) ;
            frsize = Fsize [f] ;

            if (frsize >= maxfrsize)
            {
                    /* this is the biggest seen so far */
                    maxfrsize = frsize ;
                    bigfprev = fprev ;
                    bigf = f ;
            }

            fprev = f ;
            }

            ASSERT (bigf != EMPTY) ;

            fnext = Sibling [bigf] ;

            AMD_DEBUG1 (("bigf "ID" maxfrsize "ID" bigfprev "ID" fnext "ID" fprev " ID"\n", bigf, maxfrsize, bigfprev, fnext, fprev)) ;

            if (fnext != EMPTY)
            {
            /* if fnext is EMPTY then bigf is already at the end of list */

            if (bigfprev == EMPTY)
            {
                    /* delete bigf from the element of the list */
                    Child [i] = fnext ;
            }
            else
            {
                    /* delete bigf from the middle of the list */
                    Sibling [bigfprev] = fnext ;
            }

            /* put bigf at the end of the list */
            Sibling [bigf] = EMPTY ;
            ASSERT (Child [i] != EMPTY) ;
            ASSERT (fprev != bigf) ;
            ASSERT (fprev != EMPTY) ;
            Sibling [fprev] = bigf ;
            }

        #ifndef NDEBUG

            AMD_DEBUG1 (("After partial sort, element "ID"\n", i)) ;

            for (f = Child [i] ; f != EMPTY ; f = Sibling [f])
            {
            ASSERT (f >= 0 && f < nn) ;
            AMD_DEBUG1 (("        "ID"  "ID"\n", f, Fsize [f])) ;
            ASSERT (Nv [f] > 0) ;
            nchild-- ;
            }

            ASSERT (nchild == 0) ;

        #endif
    }
    }

    /* --------------------------------------------------------------------- */
    /* postorder the assembly tree */
    /* --------------------------------------------------------------------- */

    for (i = 0 ; i < nn ; i++)
    {
    Order [i] = EMPTY ;
    }

    k = 0 ;

    for (i = 0 ; i < nn ; i++)
    {
    if (Parent [i] == EMPTY && Nv [i] > 0)
    {
            AMD_DEBUG1 (("Root of assembly tree "ID"\n", i)) ;
            k = AMD_post_tree (i, k, Child, Sibling, Order, Stack
        #ifndef NDEBUG
        , nn
        #endif
```

```
            ) ;
        }
        }
}
```

4.27 amd/amd_preprocess.c File Reference

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

Functions

- GLOBAL void AMD_preprocess (Int n, const Int Ap[ ], const Int Ai[ ], Int Rp[ ], Int Ri[ ], Int W[ ], Int Flag[ ])

4.27.1 Function Documentation

4.27.1.1 AMD_preprocess()

```
GLOBAL void AMD_preprocess (
            Int n,
            const Int Ap[ ],
            const Int Ai[ ],
            Int Rp[ ],
            Int Ri[ ],
            Int W[ ],
            Int Flag[ ] )
```

Definition at line 29 of file amd_preprocess.c.

4.28 amd_preprocess.c

Go to the documentation of this file.

```
/* ========================================================================= */
/* === AMD_preprocess ====================================================== */
/* ========================================================================= */

/* ------------------------------------------------------------------------- */
/* AMD, Copyright (c) Timothy A. Davis,                                      */
/* Patrick R. Amestoy, and Iain S. Duff.  See ../README.txt for License.     */
/* email: DrTimothyAldenDavis@gmail.com                                      */
/* ------------------------------------------------------------------------- */

/* Sorts, removes duplicate entries, and transposes from the nonzero pattern of
 * a column-form matrix A, to obtain the matrix R.  The input matrix can have
 * duplicate entries and/or unsorted columns (AMD_valid (n,Ap,Ai) must not be
 * AMD_INVALID).
 *
 * This input condition is NOT checked.  This routine is not user-callable.
 */

#include "amd_internal.h"

/* ========================================================================= */
/* === AMD_preprocess ====================================================== */
/* ========================================================================= */
```

```

/* AMD_preprocess does not check its input for errors or allocate workspace.
 * On input, the condition (AMD_valid (n,n,Ap,Ai) != AMD_INVALID) must hold.
 */

GLOBAL void AMD_preprocess
(
            Int n,          /* input matrix: A is n-by-n */
            const Int Ap [ ],       /* size n+1 */
            const Int Ai [ ],       /* size nz = Ap [n] */

            /* output matrix R: */
            Int Rp [ ],         /* size n+1 */
            Int Ri [ ],         /* size nz (or less, if duplicates present) */

            Int W [ ],          /* workspace of size n */
            Int Flag [ ] /* workspace of size n */
)
{

            /* --------------------------------------------------------------------- */
            /* local variables */
            /* --------------------------------------------------------------------- */

            Int i;
            Int j;
            Int p;
            Int p2;

            ASSERT (AMD_valid (n, n, Ap, Ai) != AMD_INVALID) ;

            /* --------------------------------------------------------------------- */
            /* count the entries in each row of A (excluding duplicates) */
            /* --------------------------------------------------------------------- */

            for (i = 0 ; i < n ; i++)
            {
               W [i] = 0 ;      /* # of nonzeros in row i (excl duplicates) */
               Flag [i] = EMPTY ; /* Flag [i] = j if i appears in column j */
            }

            for (j = 0 ; j < n ; j++)
            {
               p2 = Ap [j+1] ;
               for (p = Ap [j] ; p < p2 ; p++)
               {
                            i = Ai [p] ;
                            if (Flag [i] != j)
                            {
                                   /* row index i has not yet appeared in column j */
                                   W [i]++ ;           /* one more entry in row i */
                                   Flag [i] = j ;        /* flag row index i as appearing in col j*/
                            }
               }
            }

            /* --------------------------------------------------------------------- */
            /* compute the row pointers for R */
            /* --------------------------------------------------------------------- */

            Rp [0] = 0 ;
            for (i = 0 ; i < n ; i++)
            {
               Rp [i+1] = Rp [i] + W [i] ;
            }

            for (i = 0 ; i < n ; i++)
            {
               W [i] = Rp [i] ;
               Flag [i] = EMPTY ;
            }

            /* --------------------------------------------------------------------- */
            /* construct the row form matrix R */
            /* --------------------------------------------------------------------- */

            /* R = row form of pattern of A */
            for (j = 0 ; j < n ; j++)
            {
               p2 = Ap [j+1] ;
               for (p = Ap [j] ; p < p2 ; p++)
               {
                            i = Ai [p] ;
                            if (Flag [i] != j)
                            {
                                   /* row index i has not yet appeared in column j */
                                   Ri [W [i]++] = j ;  /* put col j in row i */
```

```
                                Flag [i] = j ;          /* flag row index i as appearing in col j*/
                          }
                }
            }

            #ifndef NDEBUG
            ASSERT (AMD_valid (n, n, Rp, Ri) == AMD_OK) ;

            for (j = 0 ; j < n ; j++)
            {
                ASSERT (W [j] == Rp [j+1]) ;
            }

            #endif
}
```

4.29 amd/amd_valid.c File Reference

`#include "amd_internal.h"`

Functions

- GLOBAL Int AMD_valid (Int n_row, Int n_col, const Int Ap[ ], const Int Ai[ ])

4.29.1 Function Documentation

4.29.1.1 AMD_valid()

```
GLOBAL Int AMD_valid (
            Int n_row,
            Int n_col,
            const Int Ap[ ],
            const Int Ai[ ] )
```

Definition at line 38 of file amd_valid.c.

4.30 amd_valid.c

Go to the documentation of this file.

```
/* ========================================================================= */
/* === AMD_valid =========================================================== */
/* ========================================================================= */

/* ------------------------------------------------------------------------- */
/* AMD, Copyright (c) Timothy A. Davis,                                  */
/* Patrick R. Amestoy, and Iain S. Duff.  See ../README.txt for License.     */
/* email: DrTimothyAldenDavis@gmail.com                                      */
/* ------------------------------------------------------------------------- */

/* Check if a column-form matrix is valid or not.  The matrix A is
 * n_row-by-n_col.  The row indices of entries in column j are in
 * Ai [Ap [j] ... Ap [j+1]-1].  Required conditions are:
 *
 *  n_row >= 0
 *  n_col >= 0
```

```
 *  nz = Ap [n_col] >= 0        number of entries in the matrix
 *  Ap [0] == 0
 *  Ap [j] <= Ap [j+1] for all j in the range 0 to n_col.
 *      Ai [0 ... nz-1] must be in the range 0 to n_row-1.
 *
 * If any of the above conditions hold, AMD_INVALID is returned.  If the
 * following condition holds, AMD_OK_BUT_JUMBLED is returned (a warning,
 * not an error):
 *
 *  row indices in Ai [Ap [j] ... Ap [j+1]-1] are not sorted in ascending
 *      order, and/or duplicate entries exist.
 *
 * Otherwise, AMD_OK is returned.
 *
 * In v1.2 and earlier, this function returned TRUE if the matrix was valid
 * (now returns AMD_OK), or FALSE otherwise (now returns AMD_INVALID or
 * AMD_OK_BUT_JUMBLED).
 */

#include "amd_internal.h"

GLOBAL Int AMD_valid
(
        /* inputs, not modified on output: */
        Int n_row,       /* A is n_row-by-n_col */
        Int n_col,
        const Int Ap [ ],     /* column pointers of A, of size n_col+1 */
        const Int Ai [ ]      /* row indices of A, of size nz = Ap [n_col] */
)
{
        Int nz;
        Int j;
        Int p1;
        Int p2;
        Int ilast;
        Int i;
        Int p;
        Int result = AMD_OK ;

        if (n_row < 0 || n_col < 0 || Ap == ABIP_NULL || Ai == ABIP_NULL)
        {
        return (AMD_INVALID) ;
        }

        nz = Ap [n_col] ;
        if (Ap [0] != 0 || nz < 0)
        {
        /* column pointers must start at Ap [0] = 0, and Ap [n] must be >= 0 */
        AMD_DEBUG0 (("column 0 pointer bad or nz < 0\n")) ;
        return (AMD_INVALID) ;
        }

        for (j = 0 ; j < n_col ; j++)
        {
        p1 = Ap [j] ;
        p2 = Ap [j+1] ;
        AMD_DEBUG2 (("\nColumn: "ID" p1: "ID" p2: "ID"\n", j, p1, p2)) ;

        if (p1 > p2)
        {
                /* column pointers must be ascending */
                AMD_DEBUG0 (("column "ID" pointer bad\n", j)) ;
                return (AMD_INVALID) ;
        }

        ilast = EMPTY ;
        for (p = p1 ; p < p2 ; p++)
        {
                i = Ai [p] ;
                AMD_DEBUG3 (("row: "ID"\n", i)) ;

                if (i < 0 || i >= n_row)
                {
                /* row index out of range */
                AMD_DEBUG0 (("index out of range, col "ID" row "ID"\n", j, i));
                return (AMD_INVALID) ;
                }

                if (i <= ilast)
                {
                /* row index unsorted, or duplicate entry present */
                AMD_DEBUG1 (("index unsorted/dupl col "ID" row "ID"\n", j, i));
                result = AMD_OK_BUT_JUMBLED ;
                }

                ilast = i;
        }
```

```
        }

        return (result) ;
}
```

4.31 amd/SuiteSparse_config.c File Reference

```
#include <math.h>
#include <stdlib.h>
#include <stdio.h>
#include "SuiteSparse_config.h"
```

Functions

- void SuiteSparse_start (void)
- void SuiteSparse_finish (void)
- void ∗ SuiteSparse_malloc (size_t nitems, size_t size_of_item)
- void ∗ SuiteSparse_calloc (size_t nitems, size_t size_of_item)
- void ∗ SuiteSparse_realloc (size_t nitems_new, size_t nitems_old, size_t size_of_item, void ∗p, int ∗ok)
- void ∗ SuiteSparse_free (void ∗p)
- void SuiteSparse_tic (abip_float tic[2])
- abip_float SuiteSparse_toc (abip_float tic[2])
- abip_float SuiteSparse_time (void)
- int SuiteSparse_version (int version[3])
- abip_float SuiteSparse_hypot (abip_float x, abip_float y)
- int SuiteSparse_divcomplex (abip_float ar, abip_float ai, abip_float br, abip_float bi, abip_float ∗cr, abip_float ∗ci)

Variables

- struct SuiteSparse_config_struct SuiteSparse_config

4.31.1 Function Documentation

4.31.1.1 SuiteSparse_calloc()

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

Definition at line 211 of file SuiteSparse_config.c.

4.31.1.2 SuiteSparse_divcomplex()

```
int SuiteSparse_divcomplex (
            abip_float ar,
            abip_float ai,
            abip_float br,
            abip_float bi,
            abip_float * cr,
            abip_float * ci )
```

Definition at line 516 of file SuiteSparse_config.c.

4.31.1.3 SuiteSparse_finish()

```
void SuiteSparse_finish (
            void )
```

Definition at line 173 of file SuiteSparse_config.c.

4.31.1.4 SuiteSparse_free()

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

Definition at line 311 of file SuiteSparse_config.c.

4.31.1.5 SuiteSparse_hypot()

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

Definition at line 464 of file SuiteSparse_config.c.

4.31.1.6 SuiteSparse_malloc()

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

Definition at line 182 of file SuiteSparse_config.c.

4.31.1.7 SuiteSparse_realloc()

```
void * SuiteSparse_realloc (
            size_t nitems_new,
            size_t nitems_old,
            size_t size_of_item,
            void * p,
            int * ok )
```

Definition at line 247 of file SuiteSparse_config.c.

4.31.1.8 SuiteSparse_start()

```
void SuiteSparse_start (
            void )
```

Definition at line 109 of file SuiteSparse_config.c.

4.31.1.9 SuiteSparse_tic()

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

Definition at line 373 of file SuiteSparse_config.c.

4.31.1.10 SuiteSparse_time()

```
abip_float SuiteSparse_time (
            void )
```

Definition at line 414 of file SuiteSparse_config.c.

4.31.1.11 SuiteSparse_toc()

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

Definition at line 397 of file SuiteSparse_config.c.

4.31.1.12 SuiteSparse_version()

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

Definition at line 429 of file SuiteSparse_config.c.

4.31.2 Variable Documentation

4.31.2.1 SuiteSparse_config

```
struct SuiteSparse_config_struct SuiteSparse_config
```

Definition at line 53 of file SuiteSparse_config.c.

4.32 SuiteSparse_config.c

Go to the documentation of this file.

```
00001 /* ========================================================================== */
00002 /* === SuiteSparse_config =================================================== */
00003 /* ========================================================================== */
00004
00005 /* SuiteSparse configuration : memory manager and printf functions. */
00006
00007 /* Copyright (c) 2013, Timothy A. Davis.  No licensing restrictions
00008  * apply to this file or to the SuiteSparse_config directory.
00009  * Author: Timothy A. Davis.
00010  */
00011
00012 #include <math.h>
00013 #include <stdlib.h>
00014
00015 #ifndef NPRINT
00016 #include <stdio.h>
00017 #endif
00018
00019 #ifdef MATLAB_MEX_FILE
00020 #include "mex.h"
00021 #include "matrix.h"
00022 #endif
00023
00024 #ifdef ABIP_NULL
00025 #undef ABIP_NULL
00026 #define ABIP_NULL ((void *) 0)
00027 #endif
00028
00029 #include "SuiteSparse_config.h"
00030
00031 /* -------------------------------------------------------------------------- */
00032 /* SuiteSparse_config : a global extern struct */
00033 /* -------------------------------------------------------------------------- */
00034
00035 /* The SuiteSparse_config struct is available to all SuiteSparse functions and
00036     to all applications that use those functions.  It must be modified with
00037     care, particularly in a multithreaded context.  Normally, the application
00038     will initialize this object once, via SuiteSparse_start, possibily 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.
```

```

    If -DPRINT is defined a compile time, then printf is disabled, and
    SuiteSparse will not use printf.
 */

struct SuiteSparse_config_struct SuiteSparse_config =
{

            /* memory management functions */
            #ifndef NMALLOC

            #ifdef MATLAB_MEX_FILE
                        /* MATLAB mexFunction: */
                        mxMalloc, mxCalloc, mxRealloc, mxFree,
            #else
                        /* standard ANSI C: */
                        malloc, calloc, realloc, free,
            #endif

            #else
                        /* no memory manager defined; you must define one at run-time: */
                        ABIP_NULL, ABIP_NULL, ABIP_NULL, ABIP_NULL,
            #endif

            /* printf function */
            #ifndef NPRINT

            #ifdef MATLAB_MEX_FILE
                        /* MATLAB mexFunction: */
                        mexPrintf,
            #else
                        /* standard ANSI C: */
                        printf,
            #endif

            #else
                        /* printf is disabled */
                        ABIP_NULL,
            #endif

            SuiteSparse_hypot,
            SuiteSparse_divcomplex

} ;

/* -------------------------------------------------------------------------- */
/* SuiteSparse_start */
/* -------------------------------------------------------------------------- */

/* All applications that use SuiteSparse should call SuiteSparse_start prior
   to using any SuiteSparse function.  Only a single thread should call this
   function, in a multithreaded application.  Currently, this function is
   optional, since all this function currently does is to set the four memory
   function pointers to ABIP_NULL (which tells SuiteSparse to use the default
   functions).  In a multi- threaded application, only a single thread should
   call this function.

   Future releases of SuiteSparse might enforce a requirement that
   SuiteSparse_start be called prior to calling any SuiteSparse function.
 */

void SuiteSparse_start ( void )
{

            /* memory management functions */
            #ifndef NMALLOC

            #ifdef MATLAB_MEX_FILE
                        /* MATLAB mexFunction: */
                        SuiteSparse_config.malloc_func  = mxMalloc ;
                        SuiteSparse_config.calloc_func  = mxCalloc ;
                        SuiteSparse_config.realloc_func = mxRealloc ;
                        SuiteSparse_config.free_func    = mxFree ;
            #else
                        /* standard ANSI C: */
                        SuiteSparse_config.malloc_func  = malloc ;
                        SuiteSparse_config.calloc_func  = calloc ;
                        SuiteSparse_config.realloc_func = realloc ;
                        SuiteSparse_config.free_func    = free ;
            #endif

            #else
                        /* no memory manager defined; you must define one after calling
 SuiteSparse_start */
                        SuiteSparse_config.malloc_func  = ABIP_NULL ;
                        SuiteSparse_config.calloc_func  = ABIP_NULL ;
                        SuiteSparse_config.realloc_func = ABIP_NULL ;
```

```
                SuiteSparse_config.free_func = ABIP_NULL ;
        #endif

        /* printf function */
        #ifndef NPRINT

        #ifdef MATLAB_MEX_FILE
                /* MATLAB mexFunction: */
                SuiteSparse_config.printf_func = mexPrintf ;
        #else
                /* standard ANSI C: */
                SuiteSparse_config.printf_func = printf ;
        #endif

        #else
                /* printf is disabled */
                SuiteSparse_config.printf_func = ABIP_NULL ;

        #endif

        /* math functions */
        SuiteSparse_config.hypot_func = SuiteSparse_hypot ;
        SuiteSparse_config.divcomplex_func = SuiteSparse_divcomplex ;
}

/* -------------------------------------------------------------------------- */
/* SuiteSparse_finish */
/* -------------------------------------------------------------------------- */

/* This currently does nothing, but in the future, applications should call
   SuiteSparse_start before calling any SuiteSparse function, and then
   SuiteSparse_finish after calling the last SuiteSparse function, just before
   exiting.  In a multithreaded application, only a single thread should call
   this function.

   Future releases of SuiteSparse might use this function for any
   SuiteSparse-wide cleanup operations or finalization of statistics.
 */

void SuiteSparse_finish ( void )
{
    /* do nothing */ ;
}

/* -------------------------------------------------------------------------- */
/* SuiteSparse_malloc: malloc wrapper */
/* -------------------------------------------------------------------------- */

void *SuiteSparse_malloc      /* pointer to allocated block of memory */
(
        size_t nitems,             /* number of items to malloc */
        size_t size_of_item     /* sizeof each item */
)
{
        void *p ;
        size_t size ;
        if (nitems < 1) nitems = 1 ;
        if (size_of_item < 1) size_of_item = 1 ;
        size = nitems * size_of_item  ;

        if (size != ((abip_float) nitems) * size_of_item)
        {
                /* size_t overflow */
                p = ABIP_NULL ;
        }
        else
        {
                p = (void *) (SuiteSparse_config.malloc_func) (size) ;
        }
        return (p) ;
}


/* -------------------------------------------------------------------------- */
/* SuiteSparse_calloc: calloc wrapper */
/* -------------------------------------------------------------------------- */

void *SuiteSparse_calloc    /* pointer to allocated block of memory */
(
        size_t nitems,          /* number of items to calloc */
        size_t size_of_item     /* sizeof each item */
)
{
        void *p ;
        size_t size ;
        if (nitems < 1) nitems = 1 ;
        if (size_of_item < 1) size_of_item = 1 ;
```

```
            size = nitems * size_of_item  ;

            if (size != ((abip_float) nitems) * size_of_item)
            {
                    /* size_t overflow */
                    p = ABIP_NULL ;
            }
            else
            {
                    p = (void *) (SuiteSparse_config.calloc_func) (nitems, size_of_item) ;
            }
            return (p) ;
}

/* ------------------------------------------------------------------------- */
/* SuiteSparse_realloc: realloc wrapper */
/* ------------------------------------------------------------------------- */

/* If p is non-NULL on input, it points to a previously allocated object of
   size nitems_old * size_of_item.  The object is reallocated to be of size
   nitems_new * size_of_item.  If p is NULL on input, then a new object of that
   size is allocated.  On success, a pointer to the new object is returned,
   and ok is returned as 1.  If the allocation fails, ok is set to 0 and a
   pointer to the old (unmodified) object is returned.
 */

void *SuiteSparse_realloc   /* pointer to reallocated block of memory, or to original block if the realloc failed. */
(
            size_t nitems_new,      /* new number of items in the object */
            size_t nitems_old,      /* old number of items in the object */
            size_t size_of_item,    /* sizeof each item */
            void *p,                /* old object to reallocate */
            int *ok                 /* 1 if successful, 0 otherwise */
)
{
            size_t size ;
            if (nitems_old < 1) nitems_old = 1 ;
            if (nitems_new < 1) nitems_new = 1 ;
            if (size_of_item < 1) size_of_item = 1 ;
            size = nitems_new * size_of_item  ;

            if (size != ((abip_float) nitems_new) * size_of_item)
            {
                    /* size_t overflow */
                    (*ok) = 0 ;
            }
            else if (p == ABIP_NULL)
            {
                    /* a fresh object is being allocated */
                    p = SuiteSparse_malloc (nitems_new, size_of_item) ;
                    (*ok) = (p != ABIP_NULL) ;
            }
            else if (nitems_old == nitems_new)
            {
                    /* the object does not change; do nothing */
                    (*ok) = 1 ;
            }
            else
            {
                    /* change the size of the object from nitems_old to nitems_new */
                    void *pnew ;
                    pnew = (void *) (SuiteSparse_config.realloc_func) (p, size) ;

                    if (pnew == ABIP_NULL)
                    {
                            if (nitems_new < nitems_old)
                            {
                                    /* the attempt to reduce the size of the block failed, but the old block is unchanged.  So pretend to succeed. */
                                    (*ok) = 1 ;
                            }
                            else
                            {
                                    /* out of memory */
                                    (*ok) = 0 ;
                            }
                    }
                    else
                    {
                            /* success */
                            p = pnew ;
                            (*ok) = 1 ;
                    }
            }
            return (p) ;
}
```

```

/* ------------------------------------------------------------------------- */
/* SuiteSparse_free: free wrapper */
/* ------------------------------------------------------------------------- */

void *SuiteSparse_free      /* always returns ABIP_NULL */
(
            void *p                 /* block to free */
)
{
            if (p)
            {
                        (SuiteSparse_config.free_func) (p) ;
            }
            return (ABIP_NULL) ;
}


/* ------------------------------------------------------------------------- */
/* SuiteSparse_tic: return current wall clock time */
/* ------------------------------------------------------------------------- */

/* Returns the number of seconds (tic [0]) and nanoseconds (tic [1]) since some
 * unspecified but fixed time in the past.  If no timer is installed, zero is
 * returned.  A scalar abip_float precision value for 'tic' could be used, but this
 * might cause loss of precision because clock_getttime returns the time from
 * some distant time in the past.  Thus, an array of size 2 is used.
 *
 * The timer is enabled by default.  To disable the timer, compile with
 * -DNTIMER.  If enabled on a POSIX C 1993 system, the timer requires linking
 * with the -lrt library.
 *
 * example:
 *
 *      abip_float tic [2], r, s, t ;
 *      SuiteSparse_tic (tic) ;     // start the timer
 *      // do some work A
 *      t = SuiteSparse_toc (tic) ; // t is time for work A, in seconds
 *      // do some work B
 *      s = SuiteSparse_toc (tic) ; // s is time for work A and B, in seconds
 *      SuiteSparse_tic (tic) ;     // restart the timer
 *      // do some work C
 *      r = SuiteSparse_toc (tic) ; // s is time for work C, in seconds
 *
 * A abip_float array of size 2 is used so that this routine can be more easily
 * ported to non-POSIX systems.  The caller does not rely on the POSIX
 * <time.h> include file.
 */

#ifdef SUITESPARSE_TIMER_ENABLED

#include <time.h>

void SuiteSparse_tic
(
            abip_float tic [2]      /* output, contents undefined on input */
)
{
            /* POSIX C 1993 timer, requires -librt */
            struct timespec t ;
            clock_gettime (CLOCK_MONOTONIC, &t) ;
            tic [0] = (abip_float) (t.tv_sec) ;
            tic [1] = (abip_float) (t.tv_nsec) ;
}

#else

void SuiteSparse_tic
(
            abip_float tic [2]      /* output, contents undefined on input */
)
{
            /* no timer installed */
            tic [0] = 0 ;
            tic [1] = 0 ;
}

#endif


/* ------------------------------------------------------------------------- */
/* SuiteSparse_toc: return time since last tic */
/* ------------------------------------------------------------------------- */

/* Assuming SuiteSparse_tic is accurate to the nanosecond, this function is
 * accurate down to the nanosecond for 2^53 nanoseconds since the last call to
 * SuiteSparse_tic, which is sufficient for SuiteSparse (about 104 days).  If
```

```
 * additional accuracy is required, the caller can use two calls to
 * SuiteSparse_tic and do the calculations differently.
 */

abip_float SuiteSparse_toc  /* returns time in seconds since last tic */
(
            abip_float tic [2]  /* input, not modified from last call to SuiteSparse_tic */
)
{
            abip_float toc [2] ;
            SuiteSparse_tic (toc) ;
            return ((toc [0] - tic [0]) + 1e-9 * (toc [1] - tic [1])) ;
}


/* -------------------------------------------------------------------------- */
/* SuiteSparse_time: return current wallclock time in seconds */
/* -------------------------------------------------------------------------- */

/* This function might not be accurate down to the nanosecond. */

abip_float SuiteSparse_time  /* returns current wall clock time in seconds */
(
            void
)
{
            abip_float toc [2] ;
            SuiteSparse_tic (toc) ;
            return (toc [0] + 1e-9 * toc [1]) ;
}


/* -------------------------------------------------------------------------- */
/* SuiteSparse_version: return the current version of SuiteSparse */
/* -------------------------------------------------------------------------- */

int SuiteSparse_version
(
            int version [3]
)
{
            if (version != ABIP_NULL)
            {
                        version [0] = SUITESPARSE_MAIN_VERSION ;
                        version [1] = SUITESPARSE_SUB_VERSION ;
                        version [2] = SUITESPARSE_SUBSUB_VERSION ;
            }
            return (SUITESPARSE_VERSION) ;
}

/* -------------------------------------------------------------------------- */
/* SuiteSparse_hypot */
/* -------------------------------------------------------------------------- */

/* There is an equivalent routine called hypot in <math.h>, which conforms
 * to ANSI C99.  However, SuiteSparse does not assume that ANSI C99 is
 * available.  You can use the ANSI C99 hypot routine with:
 *
 *      #include <math.h>
 *i     SuiteSparse_config.hypot_func = hypot ;
 *
 * Default value of the SuiteSparse_config.hypot_func pointer is
 * SuiteSparse_hypot, defined below.
 *
 * s = hypot (x,y) computes s = sqrt (x*x + y*y) but does so more accurately.
 * The NaN cases for the abip_float relops x >= y and x+y == x are safely ignored.
 *
 * Source: Algorithm 312, "Absolute value and square root of a complex number,"
 * P. Friedland, Comm. ACM, vol 10, no 10, October 1967, page 665.
 */

abip_float SuiteSparse_hypot (abip_float x, abip_float y)
{
            abip_float s;
            abip_float r ;
            x = fabs (x) ;
            y = fabs (y) ;

            if (x >= y)
            {
                        if (x + y == x)
                        {
                                    s = x ;
                        }
                        else
                        {
                                    r = y / x ;
```

```
                                s = x * sqrt (1.0 + r*r) ;
                        }
        }
        else
        {
                if (y + x == y)
                {
                        s = y ;
                }
                else
                {
                        r = x / y ;
                        s = y * sqrt (1.0 + r*r) ;
                }
        }
        return (s) ;
}

/* ------------------------------------------------------------------------- */
/* SuiteSparse_divcomplex */
/* ------------------------------------------------------------------------- */

/* c = a/b where c, a, and b are complex.  The real and imaginary parts are
 * passed as separate arguments to this routine.  The NaN case is ignored
 * for the abip_float relop br >= bi.  Returns 1 if the denominator is zero,
 * 0 otherwise.
 *
 * This uses ACM Algo 116, by R. L. Smith, 1962, which tries to avoid
 * underflow and overflow.
 *
 * c can be the same variable as a or b.
 *
 * Default value of the SuiteSparse_config.divcomplex_func pointer is
 * SuiteSparse_divcomplex.
 */

int SuiteSparse_divcomplex
(
        abip_float ar,
        abip_float ai,      /* real and imaginary parts of a */
        abip_float br,
        abip_float bi,      /* real and imaginary parts of b */
        abip_float *cr,
        abip_float *ci     /* real and imaginary parts of c */
)
{
        abip_float tr;
        abip_float ti;
        abip_float r;
        abip_float den;

        if (fabs (br) >= fabs (bi))
        {
                r = bi / br ;
                den = br + r * bi ;
                tr = (ar + ai * r) / den ;
                ti = (ai - ar * r) / den ;
        }
        else
        {
                r = br / bi ;
                den = r * br + bi ;
                tr = (ar * r + ai) / den ;
                ti = (ai * r - ar) / den ;
        }

        *cr = tr ;
        *ci = ti ;
        return (den == 0.) ;
}
```

4.33 amd/SuiteSparse_config.h File Reference

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

Classes

- struct SuiteSparse_config_struct

Macros

- #define SuiteSparse_long long
- #define SuiteSparse_long_max LONG_MAX
- #define SuiteSparse_long_idd "ld"
- #define SuiteSparse_long_id "%" SuiteSparse_long_idd
- #define SUITESPARSE_PRINTF(params) {if (SuiteSparse_config.printf_func != ABIP_NULL){(void) (Suite↩ Sparse_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_VE

Functions

- void SuiteSparse_start (void)
- void SuiteSparse_finish (void)
- void ∗ SuiteSparse_malloc (size_t nitems, size_t size_of_item)
- void ∗ SuiteSparse_calloc (size_t nitems, size_t size_of_item)
- void ∗ SuiteSparse_realloc (size_t nitems_new, size_t nitems_old, size_t size_of_item, void ∗p, int ∗ok)
- void ∗ SuiteSparse_free (void ∗p)
- void SuiteSparse_tic (abip_float tic[2])
- abip_float SuiteSparse_toc (abip_float tic[2])
- abip_float SuiteSparse_time (void)
- abip_float SuiteSparse_hypot (abip_float x, abip_float y)
- int SuiteSparse_divcomplex (abip_float ar, abip_float ai, abip_float br, abip_float bi, abip_float ∗cr, abip_float ∗ci)
- int SuiteSparse_version (int version[3])

Variables

- struct SuiteSparse_config_struct SuiteSparse_config

4.33.1 Macro Definition Documentation

4.33.1.1 SUITESPARSE_DATE

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

Definition at line 237 of file SuiteSparse_config.h.

4.33.1.2 SUITESPARSE_HAS_VERSION_FUNCTION

```
#define SUITESPARSE_HAS_VERSION_FUNCTION
```

Definition at line 235 of file SuiteSparse_config.h.

4.33.1.3 SuiteSparse_long

```
#define SuiteSparse_long long
```

Definition at line 64 of file SuiteSparse_config.h.

4.33.1.4 SuiteSparse_long_id

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

Definition at line 69 of file SuiteSparse_config.h.

4.33.1.5 SuiteSparse_long_idd

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

Definition at line 66 of file SuiteSparse_config.h.

4.33.1.6 SuiteSparse_long_max

```
#define SuiteSparse_long_max LONG_MAX
```

Definition at line 65 of file SuiteSparse_config.h.

4.33.1.7 SUITESPARSE_MAIN_VERSION

```
#define SUITESPARSE_MAIN_VERSION 5
```

Definition at line 239 of file SuiteSparse_config.h.

4.33.1.8 SUITESPARSE_PRINTF

```
#define SUITESPARSE_PRINTF(
              params ) {if (SuiteSparse_config.printf_func != ABIP_NULL){(void) (SuiteSparse_←
config.printf_func) params;}}
```

Definition at line 170 of file SuiteSparse_config.h.

4.33.1.9 SUITESPARSE_SUB_VERSION

```
#define SUITESPARSE_SUB_VERSION 1
```

Definition at line 240 of file SuiteSparse_config.h.

4.33.1.10 SUITESPARSE_SUBSUB_VERSION

```
#define SUITESPARSE_SUBSUB_VERSION 2
```

Definition at line 241 of file SuiteSparse_config.h.

4.33.1.11 SUITESPARSE_VER_CODE

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

Definition at line 238 of file SuiteSparse_config.h.

4.33.1.12 SUITESPARSE_VERSION

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

Definition at line 242 of file SuiteSparse_config.h.

4.33.2 Function Documentation

4.33.2.1 SuiteSparse_calloc()

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

Definition at line 211 of file SuiteSparse_config.c.

4.33.2.2 SuiteSparse_divcomplex()

```
int SuiteSparse_divcomplex (
            abip_float ar,
            abip_float ai,
            abip_float br,
            abip_float bi,
            abip_float * cr,
            abip_float * ci )
```

Definition at line 516 of file SuiteSparse_config.c.

4.33.2.3 SuiteSparse_finish()

```
void SuiteSparse_finish (
            void )
```

Definition at line 173 of file SuiteSparse_config.c.

4.33.2.4 SuiteSparse_free()

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

Definition at line 311 of file SuiteSparse_config.c.

4.33.2.5 SuiteSparse_hypot()

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

Definition at line 464 of file SuiteSparse_config.c.

4.33.2.6 SuiteSparse_malloc()

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

Definition at line 182 of file SuiteSparse_config.c.

4.33.2.7 SuiteSparse_realloc()

```
void * SuiteSparse_realloc (
            size_t nitems_new,
            size_t nitems_old,
            size_t size_of_item,
            void * p,
            int * ok )
```

Definition at line 247 of file SuiteSparse_config.c.

4.33.2.8 SuiteSparse_start()

```
void SuiteSparse_start (
            void )
```

Definition at line 109 of file SuiteSparse_config.c.

4.33.2.9 SuiteSparse_tic()

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

Definition at line 373 of file SuiteSparse_config.c.

4.33.2.10 SuiteSparse_time()

```
abip_float SuiteSparse_time (
            void )
```

Definition at line 414 of file SuiteSparse_config.c.

4.33.2.11 SuiteSparse_toc()

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

Definition at line 397 of file SuiteSparse_config.c.

4.33.2.12 SuiteSparse_version()

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

Definition at line 429 of file SuiteSparse_config.c.

4.33.3 Variable Documentation

4.33.3.1 SuiteSparse_config

```
struct SuiteSparse_config_struct SuiteSparse_config  [extern]
```

Definition at line 53 of file SuiteSparse_config.c.

4.34 SuiteSparse_config.h

Go to the documentation of this file.

```
/* ========================================================================== */
/* === SuiteSparse_config =================================================== */
/* ========================================================================== */

/* Configuration file for SuiteSparse: a Suite of Sparse matrix packages
 * (AMD, COLAMD, CCOLAMD, CAMD, CHOLMOD, UMFPACK, CXSparse, and others).
 *
 * SuiteSparse_config.h provides the definition of the long integer.  On most
 * systems, a C program can be compiled in LP64 mode, in which long's and
 * pointers are both 64-bits, and int's are 32-bits.  Windows 64, however, uses
 * the LLP64 model, in which int's and long's are 32-bits, and long long's and
 * pointers are 64-bits.
 *
 * SuiteSparse packages that include long integer versions are
 * intended for the LP64 mode.  However, as a workaround for Windows 64
 * (and perhaps other systems), the long integer can be redefined.
 *
 * If _WIN64 is defined, then the __int64 type is used instead of long.
 *
 * The long integer can also be defined at compile time.  For example, this
 * could be added to SuiteSparse_config.mk:
 *
 * CFLAGS = -O -D'SuiteSparse_long=long long' \
 *  -D'SuiteSparse_long_max=9223372036854775801' -D'SuiteSparse_long_idd="lld"'
 *
 * This file defines SuiteSparse_long as either long (on all but _WIN64) or
 * __int64 on Windows 64.  The intent is that a SuiteSparse_long is always a
 * 64-bit integer in a 64-bit code.  ptrdiff_t might be a better choice than
 * long; it is always the same size as a pointer.
 *
 * This file also defines the SUITESPARSE_VERSION and related definitions.
```

```
 *
 * Copyright (c) 2012, Timothy A. Davis.  No licensing restrictions apply
 * to this file or to the SuiteSparse_config directory.
 * Author: Timothy A. Davis.
 */

#ifndef SUITESPARSE_CONFIG_H
#define SUITESPARSE_CONFIG_H

#ifdef __cplusplus
extern "C" {
#endif

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

/* ========================================================================== */
/* === SuiteSparse_long ===================================================== */
/* ========================================================================== */

#ifndef SuiteSparse_long

#ifdef _WIN64

#define SuiteSparse_long __int64
#define SuiteSparse_long_max _I64_MAX
#define SuiteSparse_long_idd "I64d"

#else

#define SuiteSparse_long long
#define SuiteSparse_long_max LONG_MAX
#define SuiteSparse_long_idd "ld"

#endif
#define SuiteSparse_long_id "%" SuiteSparse_long_idd
#endif

/* ========================================================================== */
/* === SuiteSparse_config parameters and functions ========================== */
/* ========================================================================== */

/* SuiteSparse-wide parameters are placed in this struct.  It is meant to be
   an extern, globally-accessible struct.  It is not meant to be updated
   frequently by multiple threads.  Rather, if an application needs to modify
   SuiteSparse_config, it should do it once at the beginning of the application,
   before multiple threads are launched.

   The intent of these function pointers is that they not be used in your
   application directly, except to assign them to the desired user-provided
   functions.  Rather, you should use the
 */

struct SuiteSparse_config_struct
{
    void *(*malloc_func) (size_t) ;             /* pointer to malloc */
    void *(*calloc_func) (size_t, size_t) ;     /* pointer to calloc */
    void *(*realloc_func) (void *, size_t) ;    /* pointer to realloc */
    void (*free_func) (void *) ;                /* pointer to free */
    int (*printf_func) (const char *, ...) ;    /* pointer to printf */
    abip_float (*hypot_func) (abip_float, abip_float) ;     /* pointer to hypot */
    int (*divcomplex_func) (abip_float, abip_float, abip_float, abip_float, abip_float *, abip_float
 *);
} ;

extern struct SuiteSparse_config_struct SuiteSparse_config ;

void SuiteSparse_start ( void ) ;   /* called to start SuiteSparse */

void SuiteSparse_finish ( void ) ;  /* called to finish SuiteSparse */

void *SuiteSparse_malloc    /* pointer to allocated block of memory */
(
      size_t nitems,          /* number of items to malloc (>=1 is enforced) */
      size_t size_of_item     /* sizeof each item */
) ;

void *SuiteSparse_calloc    /* pointer to allocated block of memory */
(
      size_t nitems,          /* number of items to calloc (>=1 is enforced) */
      size_t size_of_item     /* sizeof each item */
) ;

void *SuiteSparse_realloc   /* pointer to reallocated block of memory, or
                               to original block if the realloc failed. */
```

```
(
      size_t nitems_new,      /* new number of items in the object */
      size_t nitems_old,      /* old number of items in the object */
      size_t size_of_item,    /* sizeof each item */
      void *p,                /* old object to reallocate */
      int *ok                 /* 1 if successful, 0 otherwise */
) ;

void *SuiteSparse_free      /* always returns NULL */
(
      void *p                 /* block to free */
) ;

void SuiteSparse_tic    /* start the timer */
(
      abip_float tic [2]      /* output, contents undefined on input */
) ;

abip_float SuiteSparse_toc  /* return time in seconds since last tic */
(
      abip_float tic [2]      /* input: from last call to SuiteSparse_tic */
) ;

abip_float SuiteSparse_time  /* returns current wall clock time in seconds */
(
    void
) ;

/* returns sqrt (x^2 + y^2), computed reliably */
abip_float SuiteSparse_hypot (abip_float x, abip_float y) ;

/* complex division of c = a/b */
int SuiteSparse_divcomplex
(
      abip_float ar,
      abip_float ai,     /* real and imaginary parts of a */
      abip_float br,
      abip_float bi,  /* real and imaginary parts of b */
      abip_float *cr,
      abip_float *ci  /* real and imaginary parts of c */
) ;

/* determine which timer to use, if any */
#ifndef NTIMER
#ifdef _POSIX_C_SOURCE
#if    _POSIX_C_SOURCE >= 199309L
#define SUITESPARSE_TIMER_ENABLED
#endif
#endif
#endif

/* SuiteSparse printf macro */
#define SUITESPARSE_PRINTF(params) {if (SuiteSparse_config.printf_func != ABIP_NULL){(void)
 (SuiteSparse_config.printf_func) params;}}

/* ========================================================================== */
/* === SuiteSparse version ================================================== */
/* ========================================================================== */

/* SuiteSparse is not a package itself, but a collection of packages, some of
 * which must be used together (UMFPACK requires AMD, CHOLMOD requires AMD,
 * COLAMD, CAMD, and CCOLAMD, etc).  A version number is provided here for the
 * collection itself.  The versions of packages within each version of
 * SuiteSparse are meant to work together.  Combining one package from one
 * version of SuiteSparse, with another package from another version of
 * SuiteSparse, may or may not work.
 *
 * SuiteSparse contains the following packages:
 *
 *  SuiteSparse_config version 5.1.2 (version always the same as SuiteSparse)
 *  GraphBLAS         version 1.1.2
 *  ssget             version 2.0.0
 *  AMD               version 2.4.6
 *  BTF               version 1.2.6
 *  CAMD              version 2.4.6
 *  CCOLAMD           version 2.9.6
 *  CHOLMOD           version 3.0.11
 *  COLAMD            version 2.9.6
 *  CSparse           version 3.2.0
 *  CXSparse          version 3.2.0
 *  GPUQREngine       version 1.0.5
 *  KLU               version 1.3.8
 *  LDL               version 2.2.6
 *  RBio              version 2.2.6
 *  SPQR              version 2.0.8
 *  SuiteSparse_GPURuntime  version 1.0.5
 *  UMFPACK           version 5.7.6
```

```
 *  MATLAB_Tools    various packages & M-files
 *  xerbla          version 1.0.3
 *
 * Other package dependencies:
 *  BLAS            required by CHOLMOD and UMFPACK
 *  LAPACK          required by CHOLMOD
 *  METIS 5.1.0     required by CHOLMOD (optional) and KLU (optional)
 *  CUBLAS, CUDART  NVIDIA libraries required by CHOLMOD and SPQR when
 *                  they are compiled with GPU acceleration.
 */

int SuiteSparse_version     /* returns SUITESPARSE_VERSION */
(
    /* output, not defined on input.  Not used if NULL.  Returns
       the three version codes in version [0..2]:
       version [0] is SUITESPARSE_MAIN_VERSION
       version [1] is SUITESPARSE_SUB_VERSION
       version [2] is SUITESPARSE_SUBSUB_VERSION
       */
    int version [3]
) ;

/* Versions prior to 4.2.0 do not have the above function.  The following
   code fragment will work with any version of SuiteSparse:

   #ifdef SUITESPARSE_HAS_VERSION_FUNCTION
   v = SuiteSparse_version (NULL) ;
   #else
   v = SUITESPARSE_VERSION ;
   #endif
*/
#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)

#ifdef __cplusplus
}
#endif
#endif
```

4.35 csparse/Include/cs.h File Reference

```
#include <stdlib.h>
#include <limits.h>
#include <math.h>
#include <stdio.h>
#include <stddef.h>
#include "glbopts.h"
```

Classes

- struct cs_sparse
- struct cs_symbolic
- struct cs_numeric
- struct cs_dmperm_results

Macros

- #define CS_VER 3 /* CSparse Version */
- #define CS_SUBVER 2

- #define CS_SUBSUB 0
- #define CS_DATE "Sept 12, 2017" /∗ CSparse release date ∗/
- #define CS_COPYRIGHT "Copyright (c) Timothy A. Davis, 2006-2016"
- #define csi abip_int
- #define CS_MAX(a, b) (((a) > (b)) ? (a) : (b))
- #define CS_MIN(a, b) (((a) < (b)) ? (a) : (b))
- #define CS_FLIP(i) (-(i)-2)
- #define CS_UNFLIP(i) (((i) < 0) ? CS_FLIP(i) : (i))
- #define CS_MARKED(w, j) (w [j] < 0)
- #define CS_MARK(w, j) { w [j] = CS_FLIP (w [j]) ; }
- #define CS_CSC(A) (A && (A->nz == -1))
- #define CS_TRIPLET(A) (A && (A->nz >= 0))

Typedefs

- typedef struct cs_sparse cs
- typedef struct cs_symbolic css
- typedef struct cs_numeric csn
- typedef struct cs_dmperm_results csd

Functions

- cs ∗ cs_add (const cs ∗A, const cs ∗B, double alpha, double beta)
- csi cs_cholsol (csi order, const cs ∗A, double ∗b)
- cs ∗ cs_compress (const cs ∗T)
- csi cs_dupl (cs ∗A)
- csi cs_entry (cs ∗T, csi i, csi j, double x)
- csi cs_gaxpy (const cs ∗A, const double ∗x, double ∗y)
- cs ∗ cs_load (FILE ∗f)
- csi cs_lusol (csi order, const cs ∗A, double ∗b, double tol)
- cs ∗ cs_multiply (const cs ∗A, const cs ∗B)
- double cs_norm (const cs ∗A)
- csi cs_print (const cs ∗A, csi brief)
- csi cs_qrsol (csi order, const cs ∗A, double ∗b)
- cs ∗ cs_transpose (const cs ∗A, csi values)
- void ∗ cs_calloc (csi n, size_t size)
- void ∗ cs_free (void ∗p)
- void ∗ cs_realloc (void ∗p, csi n, size_t size, csi ∗ok)
- cs ∗ cs_spalloc (csi m, csi n, csi nzmax, csi values, csi triplet)
- cs ∗ cs_spfree (cs ∗A)
- csi cs_sprealloc (cs ∗A, csi nzmax)
- void ∗ cs_malloc (csi n, size_t size)
- csi ∗ cs_amd (csi order, const cs ∗A)
- csn ∗ cs_chol (const cs ∗A, const css ∗S)
- csd ∗ cs_dmperm (const cs ∗A, csi seed)
- csi cs_droptol (cs ∗A, double tol)
- csi cs_dropzeros (cs ∗A)
- csi cs_happly (const cs ∗V, csi i, double beta, double ∗x)
- csi cs_ipvec (const csi ∗p, const double ∗b, double ∗x, csi n)
- csi cs_lsolve (const cs ∗L, double ∗x)
- csi cs_ltsolve (const cs ∗L, double ∗x)
- csn ∗ cs_lu (const cs ∗A, const css ∗S, double tol)
- cs ∗ cs_permute (const cs ∗A, const csi ∗pinv, const csi ∗q, csi values)

- csi * cs_pinv (const csi *p, csi n)
- csi cs_pvec (const csi *p, const double *b, double *x, csi n)
- csn * cs_qr (const cs *A, const css *S)
- css * cs_schol (csi order, const cs *A)
- css * cs_sqr (csi order, const cs *A, csi qr)
- cs * cs_symperm (const cs *A, const csi *pinv, csi values)
- csi cs_updown (cs *L, csi sigma, const cs *C, const csi *parent)
- csi cs_usolve (const cs *U, double *x)
- csi cs_utsolve (const cs *U, double *x)
- css * cs_sfree (css *S)
- csn * cs_nfree (csn *N)
- csd * cs_dfree (csd *D)
- csi * cs_counts (const cs *A, const csi *parent, const csi *post, csi ata)
- double cs_cumsum (csi *p, csi *c, csi n)
- csi cs_dfs (csi j, cs *G, csi top, csi *xi, csi *pstack, const csi *pinv)
- csi cs_ereach (const cs *A, csi k, const csi *parent, csi *s, csi *w)
- csi * cs_etree (const cs *A, csi ata)
- csi cs_fkeep (cs *A, csi(*fkeep)(csi, csi, double, void *), void *other)
- double cs_house (double *x, double *beta, csi n)
- csi cs_leaf (csi i, csi j, const csi *first, csi *maxfirst, csi *prevleaf, csi *ancestor, csi *jleaf)
- csi * cs_maxtrans (const cs *A, csi seed)
- csi * cs_post (const csi *parent, csi n)
- csi * cs_randperm (csi n, csi seed)
- csi cs_reach (cs *G, const cs *B, csi k, csi *xi, const csi *pinv)
- csi cs_scatter (const cs *A, csi j, double beta, csi *w, double *x, csi mark, cs *C, csi nz)
- csd * cs_scc (cs *A)
- csi cs_spsolve (cs *G, const cs *B, csi k, csi *xi, double *x, const csi *pinv, csi lo)
- csi cs_tdfs (csi j, csi k, csi *head, const csi *next, csi *post, csi *stack)
- csd * cs_dalloc (csi m, csi n)
- csd * cs_ddone (csd *D, cs *C, void *w, csi ok)
- cs * cs_done (cs *C, void *w, void *x, csi ok)
- csi * cs_idone (csi *p, cs *C, void *w, csi ok)
- csn * cs_ndone (csn *N, cs *C, void *w, void *x, csi ok)

4.35.1 Macro Definition Documentation

4.35.1.1 CS_COPYRIGHT

```
#define CS_COPYRIGHT "Copyright (c) Timothy A. Davis, 2006-2016"
```

Definition at line 16 of file cs.h.

4.35.1.2 CS_CSC

```
#define CS_CSC(
            A ) (A && (A->nz == -1))
```

Definition at line 152 of file cs.h.

4.35.1.3 CS_DATE

```
#define CS_DATE "Sept 12, 2017" /* CSparse release date */
```

Definition at line 15 of file cs.h.

4.35.1.4 CS_FLIP

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

Definition at line 148 of file cs.h.

4.35.1.5 CS_MARK

```
#define CS_MARK(
              w,
              j ) { w [j] = CS_FLIP (w [j]) ; }
```

Definition at line 151 of file cs.h.

4.35.1.6 CS_MARKED

```
#define CS_MARKED(
              w,
              j ) (w [j] < 0)
```

Definition at line 150 of file cs.h.

4.35.1.7 CS_MAX

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

Definition at line 146 of file cs.h.

4.35.1.8 CS_MIN

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

Definition at line 147 of file cs.h.

4.35.1.9 CS_SUBSUB

```
#define CS_SUBSUB 0
```

Definition at line 14 of file cs.h.

4.35.1.10 CS_SUBVER

```
#define CS_SUBVER 2
```

Definition at line 13 of file cs.h.

4.35.1.11 CS_TRIPLET

```
#define CS_TRIPLET(
            A ) (A && (A->nz >= 0))
```

Definition at line 153 of file cs.h.

4.35.1.12 CS_UNFLIP

```
#define CS_UNFLIP(
            i ) (((i) < 0) ? CS_FLIP(i) : (i))
```

Definition at line 149 of file cs.h.

4.35.1.13 CS_VER

```
#define CS_VER 3 /* CSparse Version */
```

Definition at line 12 of file cs.h.

4.35.1.14 csi

```
#define csi abip_int
```

Definition at line 24 of file cs.h.

4.35.2 Typedef Documentation

4.35.2.1 cs

```
typedef struct cs_sparse cs
```

4.35.2.2 csd

```
typedef struct cs_dmperm_results csd
```

4.35.2.3 csn

```
typedef struct cs_numeric csn
```

4.35.2.4 css

```
typedef struct cs_symbolic css
```

4.35.3 Function Documentation

4.35.3.1 cs_add()

```
cs * cs_add (
            const cs * A,
            const cs * B,
            double alpha,
            double beta )
```

Definition at line 3 of file cs_add.c.

4.35.3.2 cs_amd()

```
csi * cs_amd (
            csi order,
            const cs * A )
```

Definition at line 18 of file cs_amd.c.

4.35.3.3 cs_calloc()

```
void * cs_calloc (
            csi n,
            size_t size )
```

Definition at line 16 of file cs_malloc.c.

4.35.3.4 cs_chol()

```
csn * cs_chol (
            const cs * A,
            const css * S )
```

Definition at line 3 of file cs_chol.c.

4.35.3.5 cs_cholsol()

```
csi cs_cholsol (
            csi order,
            const cs * A,
            double * b )
```

Definition at line 3 of file cs_cholsol.c.

4.35.3.6 cs_compress()

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

Definition at line 3 of file cs_compress.c.

4.35.3.7 cs_counts()

```
csi * cs_counts (
            const cs * A,
            const csi * parent,
            const csi * post,
            csi ata )
```

Definition at line 17 of file cs_counts.c.

4.35.3.8 cs_cumsum()

```
double cs_cumsum (
            csi * p,
            csi * c,
            csi n )
```

Definition at line 3 of file cs_cumsum.c.

4.35.3.9 cs_dalloc()

```
csd * cs_dalloc (
            csi m,
            csi n )
```

Definition at line 66 of file cs_util.c.

4.35.3.10 cs_ddone()

```
csd * cs_ddone (
            csd * D,
            cs * C,
            void * w,
            csi ok )
```

Definition at line 115 of file cs_util.c.

4.35.3.11 cs_dfree()

```
csd * cs_dfree (
            csd * D )
```

Definition at line 79 of file cs_util.c.

4.35.3.12 cs_dfs()

```
csi cs_dfs (
            csi j,
            cs * G,
            csi top,
            csi * xi,
            csi * pstack,
            const csi * pinv )
```

Definition at line 3 of file cs_dfs.c.

4.35.3.13 cs_dmperm()

```
csd * cs_dmperm (
            const cs * A,
            csi seed )
```

Definition at line 68 of file cs_dmperm.c.

4.35.3.14 cs_done()

```
cs * cs_done (
            cs * C,
            void * w,
            void * x,
            csi ok )
```

Definition at line 90 of file cs_util.c.

4.35.3.15 cs_droptol()

```
csi cs_droptol (
            cs * A,
            double tol )
```

Definition at line 6 of file cs_droptol.c.

4.35.3.16 cs_dropzeros()

```
csi cs_dropzeros (
            cs * A )
```

Definition at line 6 of file cs_dropzeros.c.

4.35.3.17 cs_dupl()

```
csi cs_dupl (
            cs * A )
```

Definition at line 3 of file cs_dupl.c.

4.35.3.18 cs_entry()

```
csi cs_entry (
            cs * T,
            csi i,
            csi j,
            double x )
```

Definition at line 3 of file cs_entry.c.

4.35.3.19 cs_ereach()

```
csi cs_ereach (
            const cs * A,
            csi k,
            const csi * parent,
            csi * s,
            csi * w )
```

Definition at line 3 of file cs_ereach.c.

4.35.3.20 cs_etree()

```
csi * cs_etree (
            const cs * A,
            csi ata )
```

Definition at line 3 of file cs_etree.c.

4.35.3.21 cs_fkeep()

```
csi cs_fkeep (
            cs * A,
            csi(*)(csi, csi, double, void *) fkeep,
            void * other )
```

Definition at line 3 of file cs_fkeep.c.

4.35.3.22 cs_free()

```
void * cs_free (
            void * p )
```

Definition at line 22 of file cs_malloc.c.

4.35.3.23 cs_gaxpy()

```
csi cs_gaxpy (
            const cs * A,
            const double * x,
            double * y )
```

Definition at line 3 of file cs_gaxpy.c.

4.35.3.24 cs_happly()

```
csi cs_happly (
            const cs * V,
            csi i,
            double beta,
            double * x )
```

Definition at line 3 of file cs_happly.c.

4.35.3.25 cs_house()

```
double cs_house (
            double * x,
            double * beta,
            csi n )
```

Definition at line 4 of file cs_house.c.

4.35.3.26 cs_idone()

```
csi * cs_idone (
            csi * p,
            cs * C,
            void * w,
            csi ok )
```

Definition at line 98 of file cs_util.c.

4.35.3.27 cs_ipvec()

```
csi cs_ipvec (
            const csi * p,
            const double * b,
            double * x,
            csi n )
```

Definition at line 3 of file cs_ipvec.c.

4.35.3.28 cs_leaf()

```
csi cs_leaf (
            csi i,
            csi j,
            const csi * first,
            csi * maxfirst,
            csi * prevleaf,
            csi * ancestor,
            csi * jleaf )
```

Definition at line 3 of file cs_leaf.c.

4.35.3.29 cs_load()

```
cs * cs_load (
            FILE * f )
```

Definition at line 3 of file cs_load.c.

4.35.3.30 cs_lsolve()

```
csi cs_lsolve (
            const cs * L,
            double * x )
```

Definition at line 3 of file cs_lsolve.c.

4.35.3.31 cs_ltsolve()

```
csi cs_ltsolve (
            const cs * L,
            double * x )
```

Definition at line 3 of file cs_ltsolve.c.

4.35.3.32 cs_lu()

```
csn * cs_lu (
            const cs * A,
            const css * S,
            double tol )
```

Definition at line 3 of file cs_lu.c.

4.35.3.33 cs_lusol()

```
csi cs_lusol (
            csi order,
            const cs * A,
            double * b,
            double tol )
```

Definition at line 3 of file cs_lusol.c.

4.35.3.34 cs_malloc()

```
void * cs_malloc (
            csi n,
            size_t size )
```

Definition at line 10 of file cs_malloc.c.

4.35.3.35 cs_maxtrans()

```
csi * cs_maxtrans (
            const cs * A,
            csi seed )
```

Definition at line 44 of file cs_maxtrans.c.

4.35.3.36 cs_multiply()

```
cs * cs_multiply (
            const cs * A,
            const cs * B )
```

Definition at line 3 of file cs_multiply.c.

4.35.3.37 cs_ndone()

```
csn * cs_ndone (
            csn * N,
            cs * C,
            void * w,
            void * x,
            csi ok )
```

Definition at line 106 of file cs_util.c.

4.35.3.38 cs_nfree()

```
csn * cs_nfree (
            csn * N )
```

Definition at line 43 of file cs_util.c.

4.35.3.39 cs_norm()

```
double cs_norm (
            const cs * A )
```

Definition at line 3 of file cs_norm.c.

4.35.3.40 cs_permute()

```
cs * cs_permute (
            const cs * A,
            const csi * pinv,
            const csi * q,
            csi values )
```

Definition at line 3 of file cs_permute.c.

4.35.3.41 cs_pinv()

```
csi * cs_pinv (
            const csi * p,
            csi n )
```

Definition at line 3 of file cs_pinv.c.

4.35.3.42 cs_post()

```
csi * cs_post (
            const csi * parent,
            csi n )
```

Definition at line 3 of file cs_post.c.

4.35.3.43 cs_print()

```
csi cs_print (
            const cs * A,
            csi brief )
```

Definition at line 3 of file cs_print.c.

4.35.3.44 cs_pvec()

```
csi cs_pvec (
            const csi * p,
            const double * b,
            double * x,
            csi n )
```

Definition at line 3 of file cs_pvec.c.

4.35.3.45 cs_qr()

```
csn * cs_qr (
            const cs * A,
            const css * S )
```

Definition at line 3 of file cs_qr.c.

4.35.3.46 cs_qrsol()

```
csi cs_qrsol (
            csi order,
            const cs * A,
            double * b )
```

Definition at line 3 of file cs_qrsol.c.

4.35.3.47 cs_randperm()

```
csi * cs_randperm (
              csi n,
              csi seed )
```

Definition at line 5 of file cs_randperm.c.

4.35.3.48 cs_reach()

```
csi cs_reach (
              cs * G,
              const cs * B,
              csi k,
              csi * xi,
              const csi * pinv )
```

Definition at line 4 of file cs_reach.c.

4.35.3.49 cs_realloc()

```
void * cs_realloc (
              void * p,
              csi n,
              size_t size,
              csi * ok )
```

Definition at line 29 of file cs_malloc.c.

4.35.3.50 cs_scatter()

```
csi cs_scatter (
              const cs * A,
              csi j,
              double beta,
              csi * w,
              double * x,
              csi mark,
              cs * C,
              csi nz )
```

Definition at line 3 of file cs_scatter.c.

4.35.3.51 cs_scc()

```
csd * cs_scc (
            cs * A )
```

Definition at line 3 of file cs_scc.c.

4.35.3.52 cs_schol()

```
css * cs_schol (
            csi order,
            const cs * A )
```

Definition at line 3 of file cs_schol.c.

4.35.3.53 cs_sfree()

```
css * cs_sfree (
            css * S )
```

Definition at line 54 of file cs_util.c.

4.35.3.54 cs_spalloc()

```
cs * cs_spalloc (
            csi m,
            csi n,
            csi nzmax,
            csi values,
            csi triplet )
```

Definition at line 3 of file cs_util.c.

4.35.3.55 cs_spfree()

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

Definition at line 33 of file cs_util.c.

4.35.3.56 cs_sprealloc()

```
csi cs_sprealloc (
            cs * A,
            csi nzmax )
```

Definition at line 18 of file cs_util.c.

4.35.3.57 cs_spsolve()

```
csi cs_spsolve (
            cs * G,
            const cs * B,
            csi k,
            csi * xi,
            double * x,
            const csi * pinv,
            csi lo )
```

Definition at line 3 of file cs_spsolve.c.

4.35.3.58 cs_sqr()

```
css * cs_sqr (
            csi order,
            const cs * A,
            csi qr )
```

Definition at line 60 of file cs_sqr.c.

4.35.3.59 cs_symperm()

```
cs * cs_symperm (
            const cs * A,
            const csi * pinv,
            csi values )
```

Definition at line 3 of file cs_symperm.c.

4.35.3.60 cs_tdfs()

```
csi cs_tdfs (
            csi j,
            csi k,
            csi * head,
            const csi * next,
            csi * post,
            csi * stack )
```

Definition at line 3 of file cs_tdfs.c.

4.35.3.61 cs_transpose()

```
cs * cs_transpose (
            const cs * A,
            csi values )
```

Definition at line 3 of file cs_transpose.c.

4.35.3.62 cs_updown()

```
csi cs_updown (
            cs * L,
            csi sigma,
            const cs * C,
            const csi * parent )
```

Definition at line 3 of file cs_updown.c.

4.35.3.63 cs_usolve()

```
csi cs_usolve (
            const cs * U,
            double * x )
```

Definition at line 3 of file cs_usolve.c.

4.35.3.64 cs_utsolve()

```
csi cs_utsolve (
            const cs * U,
            double * x )
```

Definition at line 3 of file cs_utsolve.c.

4.36 cs.h

Go to the documentation of this file.

```
#ifndef _CS_H
#define _CS_H
#include <stdlib.h>
#include <limits.h>
#include <math.h>
#include <stdio.h>
#include <stddef.h>
#include "glbopts.h"
#ifdef MATLAB_MEX_FILE
#include "mex.h"
#endif
#define CS_VER 3                    /* CSparse Version */
#define CS_SUBVER 2
#define CS_SUBSUB 0
#define CS_DATE "Sept 12, 2017"       /* CSparse release date */
#define CS_COPYRIGHT "Copyright (c) Timothy A. Davis, 2006-2016"

// #ifdef MATLAB_MEX_FILE
// #undef csi
// #define csi mwSignedIndex
// #endif
#ifndef csi
// #define csi ptrdiff_t
#define csi abip_int
#endif

/* --- primary CSparse routines and data structures ------------------------- */
typedef struct cs_sparse    /* matrix in compressed-column or triplet form */
{
    csi nzmax ;     /* maximum number of entries */
    csi m ;         /* number of rows */
    csi n ;         /* number of columns */
    csi *p ;        /* column pointers (size n+1) or col indices (size nzmax) */
    csi *i ;        /* row indices, size nzmax */
    double *x ;     /* numerical values, size nzmax */
    csi nz ;        /* # of entries in triplet matrix, -1 for compressed-col */
} cs ;

cs *cs_add (const cs *A, const cs *B, double alpha, double beta) ;
csi cs_cholsol (csi order, const cs *A, double *b) ;
cs *cs_compress (const cs *T) ;
csi cs_dupl (cs *A) ;
csi cs_entry (cs *T, csi i, csi j, double x) ;
csi cs_gaxpy (const cs *A, const double *x, double *y) ;
cs *cs_load (FILE *f) ;
csi cs_lusol (csi order, const cs *A, double *b, double tol) ;
cs *cs_multiply (const cs *A, const cs *B) ;
double cs_norm (const cs *A) ;
csi cs_print (const cs *A, csi brief) ;
csi cs_qrsol (csi order, const cs *A, double *b) ;
cs *cs_transpose (const cs *A, csi values) ;
/* utilities */
void *cs_calloc (csi n, size_t size) ;
void *cs_free (void *p) ;
void *cs_realloc (void *p, csi n, size_t size, csi *ok) ;
cs *cs_spalloc (csi m, csi n, csi nzmax, csi values, csi triplet) ;
cs *cs_spfree (cs *A) ;
csi cs_sprealloc (cs *A, csi nzmax) ;
void *cs_malloc (csi n, size_t size) ;

/* --- secondary CSparse routines and data structures ----------------------- */
typedef struct cs_symbolic  /* symbolic Cholesky, LU, or QR analysis */
{
    csi *pinv ;     /* inverse row perm. for QR, fill red. perm for Chol */
    csi *q ;        /* fill-reducing column permutation for LU and QR */
    csi *parent ;   /* elimination tree for Cholesky and QR */
    csi *cp ;       /* column pointers for Cholesky, row counts for QR */
    csi *leftmost ; /* leftmost[i] = min(find(A(i,:))), for QR */
    csi m2 ;        /* # of rows for QR, after adding fictitious rows */
    double lnz ;    /* # entries in L for LU or Cholesky; in V for QR */
    double unz ;    /* # entries in U for LU; in R for QR */
} css ;

typedef struct cs_numeric   /* numeric Cholesky, LU, or QR factorization */
{
    cs *L ;         /* L for LU and Cholesky, V for QR */
    cs *U ;         /* U for LU, R for QR, not used for Cholesky */
    csi *pinv ;     /* partial pivoting for LU */
    double *B ;     /* beta [0..n-1] for QR */
} csn ;

typedef struct cs_dmperm_results    /* cs_dmperm or cs_scc output */
```

```
{
    csi *p ;        /* size m, row permutation */
    csi *q ;        /* size n, column permutation */
    csi *r ;        /* size nb+1, block k is rows r[k] to r[k+1]-1 in A(p,q) */
    csi *s ;        /* size nb+1, block k is cols s[k] to s[k+1]-1 in A(p,q) */
    csi nb ;        /* # of blocks in fine dmperm decomposition */
    csi rr [5] ;    /* coarse row decomposition */
    csi cc [5] ;    /* coarse column decomposition */
} csd ;

csi *cs_amd (csi order, const cs *A) ;
csn *cs_chol (const cs *A, const css *S) ;
csd *cs_dmperm (const cs *A, csi seed) ;
csi cs_droptol (cs *A, double tol) ;
csi cs_dropzeros (cs *A) ;
csi cs_happly (const cs *V, csi i, double beta, double *x) ;
csi cs_ipvec (const csi *p, const double *b, double *x, csi n) ;
csi cs_lsolve (const cs *L, double *x) ;
csi cs_ltsolve (const cs *L, double *x) ;
csn *cs_lu (const cs *A, const css *S, double tol) ;
cs *cs_permute (const cs *A, const csi *pinv, const csi *q, csi values) ;
csi *cs_pinv (const csi *p, csi n) ;
csi cs_pvec (const csi *p, const double *b, double *x, csi n) ;
csn *cs_qr (const cs *A, const css *S) ;
css *cs_schol (csi order, const cs *A) ;
css *cs_sqr (csi order, const cs *A, csi qr) ;
cs *cs_symperm (const cs *A, const csi *pinv, csi values) ;
csi cs_updown (cs *L, csi sigma, const cs *C, const csi *parent) ;
csi cs_usolve (const cs *U, double *x) ;
csi cs_utsolve (const cs *U, double *x) ;
/* utilities */
css *cs_sfree (css *S) ;
csn *cs_nfree (csn *N) ;
csd *cs_dfree (csd *D) ;

/* --- tertiary CSparse routines -------------------------------------------- */
csi *cs_counts (const cs *A, const csi *parent, const csi *post, csi ata) ;
double cs_cumsum (csi *p, csi *c, csi n) ;
csi cs_dfs (csi j, cs *G, csi top, csi *xi, csi *pstack, const csi *pinv) ;
csi cs_ereach (const cs *A, csi k, const csi *parent, csi *s, csi *w) ;
csi *cs_etree (const cs *A, csi ata) ;
csi cs_fkeep (cs *A, csi (*fkeep) (csi, csi, double, void *), void *other) ;
double cs_house (double *x, double *beta, csi n) ;
csi cs_leaf (csi i, csi j, const csi *first, csi *maxfirst, csi *prevleaf,
    csi *ancestor, csi *jleaf) ;
csi *cs_maxtrans (const cs *A, csi seed) ;
csi *cs_post (const csi *parent, csi n) ;
csi *cs_randperm (csi n, csi seed) ;
csi cs_reach (cs *G, const cs *B, csi k, csi *xi, const csi *pinv) ;
csi cs_scatter (const cs *A, csi j, double beta, csi *w, double *x, csi mark,
    cs *C, csi nz) ;
csd *cs_scc (cs *A) ;
csi cs_spsolve (cs *G, const cs *B, csi k, csi *xi, double *x,
    const csi *pinv, csi lo) ;
csi cs_tdfs (csi j, csi k, csi *head, const csi *next, csi *post,
    csi *stack) ;
/* utilities */
csd *cs_dalloc (csi m, csi n) ;
csd *cs_ddone (csd *D, cs *C, void *w, csi ok) ;
cs *cs_done (cs *C, void *w, void *x, csi ok) ;
csi *cs_idone (csi *p, cs *C, void *w, csi ok) ;
csn *cs_ndone (csn *N, cs *C, void *w, void *x, csi ok) ;

#define CS_MAX(a,b) (((a) > (b)) ? (a) : (b))
#define CS_MIN(a,b) (((a) < (b)) ? (a) : (b))
#define CS_FLIP(i) (-(i)-2)
#define CS_UNFLIP(i) (((i) < 0) ? CS_FLIP(i) : (i))
#define CS_MARKED(w,j) (w [j] < 0)
#define CS_MARK(w,j) { w [j] = CS_FLIP (w [j]) ; }
#define CS_CSC(A) (A && (A->nz == -1))
#define CS_TRIPLET(A) (A && (A->nz >= 0))
#endif
```

4.37 csparse/Source/cs_add.c File Reference

```
#include "cs.h"
```

Functions

- cs * cs_add (const cs *A, const cs *B, double alpha, double beta)

4.37.1 Function Documentation

4.37.1.1 cs_add()

```
cs * cs_add (
            const cs * A,
            const cs * B,
            double alpha,
            double beta )
```

Definition at line 3 of file cs_add.c.

4.38 cs_add.c

Go to the documentation of this file.

```
00001 #include "cs.h"
00002 /* C = alpha*A + beta*B */
00003 cs *cs_add (const cs *A, const cs *B, double alpha, double beta)
00004 {
00005     csi p, j, nz = 0, anz, *Cp, *Ci, *Bp, m, n, bnz, *w, values ;
00006     double *x, *Bx, *Cx ;
00007     cs *C ;
00008     if (!CS_CSC (A) || !CS_CSC (B)) return (NULL) ;         /* check inputs */
00009     if (A->m != B->m || A->n != B->n) return (NULL) ;
00010     m = A->m ; anz = A->p [A->n] ;
00011     n = B->n ; Bp = B->p ; Bx = B->x ; bnz = Bp [n] ;
00012     w = cs_calloc (m, sizeof (csi)) ;                       /* get workspace */
00013     values = (A->x != NULL) && (Bx != NULL) ;
00014     x = values ? cs_malloc (m, sizeof (double)) : NULL ;    /* get workspace */
00015     C = cs_spalloc (m, n, anz + bnz, values, 0) ;           /* allocate result*/
00016     if (!C || !w || (values && !x)) return (cs_done (C, w, x, 0)) ;
00017     Cp = C->p ; Ci = C->i ; Cx = C->x ;
00018     for (j = 0 ; j < n ; j++)
00019     {
00020         Cp [j] = nz ;                   /* column j of C starts here */
00021         nz = cs_scatter (A, j, alpha, w, x, j+1, C, nz) ;   /* alpha*A(:,j)*/
00022         nz = cs_scatter (B, j, beta, w, x, j+1, C, nz) ;    /* beta*B(:,j) */
00023         if (values) for (p = Cp [j] ; p < nz ; p++) Cx [p] = x [Ci [p]] ;
00024     }
00025     Cp [n] = nz ;                       /* finalize the last column of C */
00026     cs_sprealloc (C, 0) ;               /* remove extra space from C */
00027     return (cs_done (C, w, x, 1)) ;     /* success; free workspace, return C */
00028 }
```

4.39 csparse/Source/cs_amd.c File Reference

```
#include "cs.h"
```

Functions

- csi * cs_amd (csi order, const cs *A)

4.39.1 Function Documentation

4.39.1.1 cs_amd()

```
csi * cs_amd (
            csi order,
            const cs * A )
```

Definition at line 18 of file cs_amd.c.

4.40 cs_amd.c

Go to the documentation of this file.

```
00001 #include "cs.h"
00002 /* clear w */
00003 static csi cs_wclear (csi mark, csi lemax, csi *w, csi n)
00004 {
00005     csi k ;
00006     if (mark < 2 || (mark + lemax < 0))
00007     {
00008         for (k = 0 ; k < n ; k++) if (w [k] != 0) w [k] = 1 ;
00009         mark = 2 ;
00010     }
00011     return (mark) ;     /* at this point, w [0..n-1] < mark holds */
00012 }
00013
00014 /* keep off-diagonal entries; drop diagonal entries */
00015 static csi cs_diag (csi i, csi j, double aij, void *other) { return (i != j) ; }
00016
00017 /* p = amd(A+A') if symmetric is true, or amd(A'A) otherwise */
00018 csi *cs_amd (csi order, const cs *A)  /* order 0:natural, 1:Chol, 2:LU, 3:QR */
00019 {
00020     cs *C, *A2, *AT ;
00021     csi *Cp, *Ci, *last, *W, *len, *nv, *next, *P, *head, *elen, *degree, *w,
00022         *hhead, *ATp, *ATi, d, dk, dext, lemax = 0, e, elenk, eln, i, j, k, k1,
00023         k2, k3, jlast, ln, dense, nzmax, mindeg = 0, nvi, nvj, nvk, mark, wnvi,
00024         ok, cnz, nel = 0, p, p1, p2, p3, p4, pj, pk, pk1, pk2, pn, q, n, m, t ;
00025     csi h ;
00026     /* --- Construct matrix C ----------------------------------------------- */
00027     if (!CS_CSC (A) || order <= 0 || order > 3) return (NULL) ; /* check */
00028     AT = cs_transpose (A, 0) ;              /* compute A' */
00029     if (!AT) return (NULL) ;
00030     m = A->m ; n = A->n ;
00031     dense = CS_MAX (16, 10 * sqrt ((double) n)) ;   /* find dense threshold */
00032     dense = CS_MIN (n-2, dense) ;
00033     if (order == 1 && n == m)
00034     {
00035         C = cs_add (A, AT, 0, 0) ;          /* C = A+A' */
00036     }
00037     else if (order == 2)
00038     {
00039         ATp = AT->p ;                       /* drop dense columns from AT */
00040         ATi = AT->i ;
00041         for (p2 = 0, j = 0 ; j < m ; j++)
00042         {
00043             p = ATp [j] ;                   /* column j of AT starts here */
00044             ATp [j] = p2 ;                  /* new column j starts here */
00045             if (ATp [j+1] - p > dense) continue ;   /* skip dense col j */
00046             for ( ; p < ATp [j+1] ; p++) ATi [p2++] = ATi [p] ;
00047         }
00048         ATp [m] = p2 ;                      /* finalize AT */
00049         A2 = cs_transpose (AT, 0) ;         /* A2 = AT' */
00050         C = A2 ? cs_multiply (AT, A2) : NULL ;  /* C=A'*A with no dense rows */
00051         cs_spfree (A2) ;
00052     }
00053     else
00054     {
00055         C = cs_multiply (AT, A) ;           /* C=A'*A */
00056     }
00057     cs_spfree (AT) ;
00058     if (!C) return (NULL) ;
```

```
    cs_fkeep (C, &cs_diag, NULL) ;          /* drop diagonal entries */
    Cp = C->p ;
    cnz = Cp [n] ;
    P = cs_malloc (n+1, sizeof (csi)) ;     /* allocate result */
    W = cs_malloc (8*(n+1), sizeof (csi)) ; /* get workspace */
    t = cnz + cnz/5 + 2*n ;                 /* add elbow room to C */
    if (!P || !W || !cs_sprealloc (C, t)) return (cs_idone (P, C, W, 0)) ;
    len  = W           ; nv     = W +   (n+1) ; next   = W + 2*(n+1) ;
    head = W + 3*(n+1) ; elen   = W + 4*(n+1) ; degree = W + 5*(n+1) ;
    w    = W + 6*(n+1) ; hhead  = W + 7*(n+1) ;
    last = P ;                              /* use P as workspace for last */
    /* --- Initialize quotient graph ---------------------------------------- */
    for (k = 0 ; k < n ; k++) len [k] = Cp [k+1] - Cp [k] ;
    len [n] = 0 ;
    nzmax = C->nzmax ;
    Ci = C->i ;
    for (i = 0 ; i <= n ; i++)
    {
        head [i] = -1 ;                     /* degree list i is empty */
        last [i] = -1 ;
        next [i] = -1 ;
        hhead [i] = -1 ;                    /* hash list i is empty */
        nv [i] = 1 ;                        /* node i is just one node */
        w [i] = 1 ;                         /* node i is alive */
        elen [i] = 0 ;                      /* Ek of node i is empty */
        degree [i] = len [i] ;              /* degree of node i */
    }
    mark = cs_wclear (0, 0, w, n) ;         /* clear w */
    elen [n] = -2 ;                         /* n is a dead element */
    Cp [n] = -1 ;                           /* n is a root of assembly tree */
    w [n] = 0 ;                             /* n is a dead element */
    /* --- Initialize degree lists ------------------------------------------ */
    for (i = 0 ; i < n ; i++)
    {
        d = degree [i] ;
        if (d == 0)                         /* node i is empty */
        {
            elen [i] = -2 ;                 /* element i is dead */
            nel++ ;
            Cp [i] = -1 ;                   /* i is a root of assembly tree */
            w [i] = 0 ;
        }
        else if (d > dense)                 /* node i is dense */
        {
            nv [i] = 0 ;                    /* absorb i into element n */
            elen [i] = -1 ;                 /* node i is dead */
            nel++ ;
            Cp [i] = CS_FLIP (n) ;
            nv [n]++ ;
        }
        else
        {
            if (head [d] != -1) last [head [d]] = i ;
            next [i] = head [d] ;           /* put node i in degree list d */
            head [d] = i ;
        }
    }
    while (nel < n)                         /* while (selecting pivots) do */
    {
        /* --- Select node of minimum approximate degree -------------------- */
        for (k = -1 ; mindeg < n && (k = head [mindeg]) == -1 ; mindeg++) ;
        if (next [k] != -1) last [next [k]] = -1 ;
        head [mindeg] = next [k] ;          /* remove k from degree list */
        elenk = elen [k] ;                  /* elenk = |Ek| */
        nvk = nv [k] ;                      /* # of nodes k represents */
        nel += nvk ;                        /* nv[k] nodes of A eliminated */
        /* --- Garbage collection ------------------------------------------- */
        if (elenk > 0 && cnz + mindeg >= nzmax)
        {
            for (j = 0 ; j < n ; j++)
            {
                if ((p = Cp [j]) >= 0)      /* j is a live node or element */
                {
                    Cp [j] = Ci [p] ;       /* save first entry of object */
                    Ci [p] = CS_FLIP (j) ;  /* first entry is now CS_FLIP(j) */
                }
            }
            for (q = 0, p = 0 ; p < cnz ; ) /* scan all of memory */
            {
                if ((j = CS_FLIP (Ci [p++])) >= 0)  /* found object j */
                {
                    Ci [q] = Cp [j] ;       /* restore first entry of object */
                    Cp [j] = q++ ;          /* new pointer to object j */
                    for (k3 = 0 ; k3 < len [j]-1 ; k3++) Ci [q++] = Ci [p++] ;
                }
            }
            cnz = q ;                       /* Ci [cnz...nzmax-1] now free */
```

```
        }
        /* --- Construct new element ---------------------------------------- */
        dk = 0 ;
        nv [k] = -nvk ;                     /* flag k as in Lk */
        p = Cp [k] ;
        pk1 = (elenk == 0) ? p : cnz ;      /* do in place if elen[k] == 0 */
        pk2 = pk1 ;
        for (k1 = 1 ; k1 <= elenk + 1 ; k1++)
        {
            if (k1 > elenk)
            {
                e = k ;                     /* search the nodes in k */
                pj = p ;                    /* list of nodes starts at Ci[pj]*/
                ln = len [k] - elenk ;      /* length of list of nodes in k */
            }
            else
            {
                e = Ci [p++] ;              /* search the nodes in e */
                pj = Cp [e] ;
                ln = len [e] ;              /* length of list of nodes in e */
            }
            for (k2 = 1 ; k2 <= ln ; k2++)
            {
                i = Ci [pj++] ;
                if ((nvi = nv [i]) <= 0) continue ; /* node i dead, or seen */
                dk += nvi ;                 /* degree[Lk] += size of node i */
                nv [i] = -nvi ;             /* negate nv[i] to denote i in Lk*/
                Ci [pk2++] = i ;            /* place i in Lk */
                if (next [i] != -1) last [next [i]] = last [i] ;
                if (last [i] != -1)         /* remove i from degree list */
                {
                    next [last [i]] = next [i] ;
                }
                else
                {
                    head [degree [i]] = next [i] ;
                }
            }
            if (e != k)
            {
                Cp [e] = CS_FLIP (k) ;      /* absorb e into k */
                w [e] = 0 ;                 /* e is now a dead element */
            }
        }
        if (elenk != 0) cnz = pk2 ;         /* Ci [cnz...nzmax] is free */
        degree [k] = dk ;                   /* external degree of k - |Lk\i| */
        Cp [k] = pk1 ;                      /* element k is in Ci[pk1..pk2-1] */
        len [k] = pk2 - pk1 ;
        elen [k] = -2 ;                     /* k is now an element */
        /* --- Find set differences ----------------------------------------- */
        mark = cs_wclear (mark, lemax, w, n) ;  /* clear w if necessary */
        for (pk = pk1 ; pk < pk2 ; pk++)    /* scan 1: find |Le\Lk| */
        {
            i = Ci [pk] ;
            if ((eln = elen [i]) <= 0) continue ;/* skip if elen[i] empty */
            nvi = -nv [i] ;                      /* nv [i] was negated */
            wnvi = mark - nvi ;
            for (p = Cp [i] ; p <= Cp [i] + eln - 1 ; p++)  /* scan Ei */
            {
                e = Ci [p] ;
                if (w [e] >= mark)
                {
                    w [e] -= nvi ;          /* decrement |Le\Lk| */
                }
                else if (w [e] != 0)        /* ensure e is a live element */
                {
                    w [e] = degree [e] + wnvi ; /* 1st time e seen in scan 1 */
                }
            }
        }
        /* --- Degree update ------------------------------------------------ */
        for (pk = pk1 ; pk < pk2 ; pk++)    /* scan2: degree update */
        {
            i = Ci [pk] ;                   /* consider node i in Lk */
            p1 = Cp [i] ;
            p2 = p1 + elen [i] - 1 ;
            pn = p1 ;
            for (h = 0, d = 0, p = p1 ; p <= p2 ; p++)    /* scan Ei */
            {
                e = Ci [p] ;
                if (w [e] != 0)             /* e is an unabsorbed element */
                {
                    dext = w [e] - mark ;   /* dext = |Le\Lk| */
                    if (dext > 0)
                    {
                        d += dext ;         /* sum up the set differences */
                        Ci [pn++] = e ;     /* keep e in Ei */
```

```
                h += e ;            /* compute the hash of node i */
            }
            else
            {
                Cp [e] = CS_FLIP (k) ;  /* aggressive absorb. e->k */
                w [e] = 0 ;             /* e is a dead element */
            }
        }
    }
    elen [i] = pn - p1 + 1 ;        /* elen[i] = |Ei| */
    p3 = pn ;
    p4 = p1 + len [i] ;
    for (p = p2 + 1 ; p < p4 ; p++) /* prune edges in Ai */
    {
        j = Ci [p] ;
        if ((nvj = nv [j]) <= 0) continue ; /* node j dead or in Lk */
        d += nvj ;                  /* degree(i) += |j| */
        Ci [pn++] = j ;             /* place j in node list of i */
        h += j ;                    /* compute hash for node i */
    }
    if (d == 0)                     /* check for mass elimination */
    {
        Cp [i] = CS_FLIP (k) ;      /* absorb i into k */
        nvi = -nv [i] ;
        dk -= nvi ;                 /* |Lk| -= |i| */
        nvk += nvi ;                /* |k| += nv[i] */
        nel += nvi ;
        nv [i] = 0 ;
        elen [i] = -1 ;             /* node i is dead */
    }
    else
    {
        degree [i] = CS_MIN (degree [i], d) ;   /* update degree(i) */
        Ci [pn] = Ci [p3] ;         /* move first node to end */
        Ci [p3] = Ci [p1] ;         /* move 1st el. to end of Ei */
        Ci [p1] = k ;               /* add k as 1st element in of Ei */
        len [i] = pn - p1 + 1 ;     /* new len of adj. list of node i */
        h = ((h<0) ? (-h):h) % n ;  /* finalize hash of i */
        next [i] = hhead [h] ;      /* place i in hash bucket */
        hhead [h] = i ;
        last [i] = h ;              /* save hash of i in last[i] */
    }
}                                   /* scan2 is done */
degree [k] = dk ;                   /* finalize |Lk| */
lemax = CS_MAX (lemax, dk) ;
mark = cs_wclear (mark+lemax, lemax, w, n) ;    /* clear w */
/* --- Supernode detection ------------------------------------------ */
for (pk = pk1 ; pk < pk2 ; pk++)
{
    i = Ci [pk] ;
    if (nv [i] >= 0) continue ;         /* skip if i is dead */
    h = last [i] ;                      /* scan hash bucket of node i */
    i = hhead [h] ;
    hhead [h] = -1 ;                    /* hash bucket will be empty */
    for ( ; i != -1 && next [i] != -1 ; i = next [i], mark++)
    {
        ln = len [i] ;
        eln = elen [i] ;
        for (p = Cp [i]+1 ; p <= Cp [i] + ln-1 ; p++) w [Ci [p]] = mark;
        jlast = i ;
        for (j = next [i] ; j != -1 ; ) /* compare i with all j */
        {
            ok = (len [j] == ln) && (elen [j] == eln) ;
            for (p = Cp [j] + 1 ; ok && p <= Cp [j] + ln - 1 ; p++)
            {
                if (w [Ci [p]] != mark) ok = 0 ;    /* compare i and j*/
            }
            if (ok)                     /* i and j are identical */
            {
                Cp [j] = CS_FLIP (i) ;  /* absorb j into i */
                nv [i] += nv [j] ;
                nv [j] = 0 ;
                elen [j] = -1 ;         /* node j is dead */
                j = next [j] ;          /* delete j from hash bucket */
                next [jlast] = j ;
            }
            else
            {
                jlast = j ;             /* j and i are different */
                j = next [j] ;
            }
        }
    }
}
/* --- Finalize new element------------------------------------------ */
for (p = pk1, pk = pk1 ; pk < pk2 ; pk++)   /* finalize Lk */
{
```

```
            i = Ci [pk] ;
            if ((nvi = -nv [i]) <= 0) continue ;/* skip if i is dead */
            nv [i] = nvi ;                      /* restore nv[i] */
            d = degree [i] + dk - nvi ;         /* compute external degree(i) */
            d = CS_MIN (d, n - nel - nvi) ;
            if (head [d] != -1) last [head [d]] = i ;
            next [i] = head [d] ;               /* put i back in degree list */
            last [i] = -1 ;
            head [d] = i ;
            mindeg = CS_MIN (mindeg, d) ;       /* find new minimum degree */
            degree [i] = d ;
            Ci [p++] = i ;                      /* place i in Lk */
        }
        nv [k] = nvk ;                      /* # nodes absorbed into k */
        if ((len [k] = p-pk1) == 0)         /* length of adj list of element k*/
        {
            Cp [k] = -1 ;                   /* k is a root of the tree */
            w [k] = 0 ;                     /* k is now a dead element */
        }
        if (elenk != 0) cnz = p ;           /* free unused space in Lk */
    }
    /* --- Postordering ----------------------------------------------------- */
    for (i = 0 ; i < n ; i++) Cp [i] = CS_FLIP (Cp [i]) ;/* fix assembly tree */
    for (j = 0 ; j <= n ; j++) head [j] = -1 ;
    for (j = n ; j >= 0 ; j--)              /* place unordered nodes in lists */
    {
        if (nv [j] > 0) continue ;          /* skip if j is an element */
        next [j] = head [Cp [j]] ;          /* place j in list of its parent */
        head [Cp [j]] = j ;
    }
    for (e = n ; e >= 0 ; e--)              /* place elements in lists */
    {
        if (nv [e] <= 0) continue ;         /* skip unless e is an element */
        if (Cp [e] != -1)
        {
            next [e] = head [Cp [e]] ;      /* place e in list of its parent */
            head [Cp [e]] = e ;
        }
    }
    for (k = 0, i = 0 ; i <= n ; i++)       /* postorder the assembly tree */
    {
        if (Cp [i] == -1) k = cs_tdfs (i, k, head, next, P, w) ;
    }
    return (cs_idone (P, C, W, 1)) ;
}
```

4.41 csparse/Source/cs_chol.c File Reference

```
#include "cs.h"
```

Functions

- csn ∗ cs_chol (const cs ∗A, const css ∗S)

4.41.1 Function Documentation

4.41.1.1 cs_chol()

```
csn * cs_chol (
        const cs * A,
        const css * S )
```

Definition at line 3 of file cs_chol.c.

4.42 cs_chol.c

Go to the documentation of this file.

```c
#include "cs.h"
/* L = chol (A, [pinv parent cp]), pinv is optional */
csn *cs_chol (const cs *A, const css *S)
{
    double d, lki, *Lx, *x, *Cx ;
    csi top, i, p, k, n, *Li, *Lp, *cp, *pinv, *s, *c, *parent, *Cp, *Ci ;
    cs *L, *C, *E ;
    csn *N ;
    if (!CS_CSC (A) || !S || !S->cp || !S->parent) return (NULL) ;
    n = A->n ;
    N = cs_calloc (1, sizeof (csn)) ;       /* allocate result */
    c = cs_malloc (2*n, sizeof (csi)) ;     /* get csi workspace */
    x = cs_malloc (n, sizeof (double)) ;    /* get double workspace */
    cp = S->cp ; pinv = S->pinv ; parent = S->parent ;
    C = pinv ? cs_symperm (A, pinv, 1) : ((cs *) A) ;
    E = pinv ? C : NULL ;           /* E is alias for A, or a copy E=A(p,p) */
    if (!N || !c || !x || !C) return (cs_ndone (N, E, c, x, 0)) ;
    s = c + n ;
    Cp = C->p ; Ci = C->i ; Cx = C->x ;
    N->L = L = cs_spalloc (n, n, cp [n], 1, 0) ;    /* allocate result */
    if (!L) return (cs_ndone (N, E, c, x, 0)) ;
    Lp = L->p ; Li = L->i ; Lx = L->x ;
    for (k = 0 ; k < n ; k++) Lp [k] = c [k] = cp [k] ;
    for (k = 0 ; k < n ; k++)       /* compute L(k,:) for L*L' = C */
    {
        /* --- Nonzero pattern of L(k,:) ------------------------------------ */
        top = cs_ereach (C, k, parent, s, c) ;      /* find pattern of L(k,:) */
        x [k] = 0 ;                                 /* x (0:k) is now zero */
        for (p = Cp [k] ; p < Cp [k+1] ; p++)       /* x = full(triu(C(:,k))) */
        {
            if (Ci [p] <= k) x [Ci [p]] = Cx [p] ;
        }
        d = x [k] ;                     /* d = C(k,k) */
        x [k] = 0 ;                     /* clear x for k+1st iteration */
        /* --- Triangular solve --------------------------------------------- */
        for ( ; top < n ; top++)    /* solve L(0:k-1,0:k-1) * x = C(:,k) */
        {
            i = s [top] ;               /* s [top..n-1] is pattern of L(k,:) */
            lki = x [i] / Lx [Lp [i]] ; /* L(k,i) = x (i) / L(i,i) */
            x [i] = 0 ;                 /* clear x for k+1st iteration */
            for (p = Lp [i] + 1 ; p < c [i] ; p++)
            {
                x [Li [p]] -= Lx [p] * lki ;
            }
            d -= lki * lki ;            /* d = d - L(k,i)*L(k,i) */
            p = c [i]++ ;
            Li [p] = k ;                /* store L(k,i) in column i */
            Lx [p] = lki ;
        }
        /* --- Compute L(k,k) ----------------------------------------------- */
        if (d <= 0) return (cs_ndone (N, E, c, x, 0)) ; /* not pos def */
        p = c [k]++ ;
        Li [p] = k ;                /* store L(k,k) = sqrt (d) in column k */
        Lx [p] = sqrt (d) ;
    }
    Lp [n] = cp [n] ;               /* finalize L */
    return (cs_ndone (N, E, c, x, 1)) ; /* success: free E,s,x; return N */
}
```

4.43 csparse/Source/cs_cholsol.c File Reference

```c
#include "cs.h"
```

Functions

- csi cs_cholsol (csi order, const cs *A, double *b)

4.43.1 Function Documentation

4.43.1.1 cs_cholsol()

```
csi cs_cholsol (
            csi order,
            const cs * A,
            double * b )
```

Definition at line 3 of file cs_cholsol.c.

4.44 cs_cholsol.c

Go to the documentation of this file.

```
00001 #include "cs.h"
00002 /* x=A\b where A is symmetric positive definite; b overwritten with solution */
00003 csi cs_cholsol (csi order, const cs *A, double *b)
00004 {
00005     double *x ;
00006     css *S ;
00007     csn *N ;
00008     csi n, ok ;
00009     if (!CS_CSC (A) || !b) return (0) ;      /* check inputs */
00010     n = A->n ;
00011     S = cs_schol (order, A) ;                /* ordering and symbolic analysis */
00012     N = cs_chol (A, S) ;                     /* numeric Cholesky factorization */
00013     x = cs_malloc (n, sizeof (double)) ;     /* get workspace */
00014     ok = (S && N && x) ;
00015     if (ok)
00016     {
00017         cs_ipvec (S->pinv, b, x, n) ;    /* x = P*b */
00018         cs_lsolve (N->L, x) ;            /* x = L\x */
00019         cs_ltsolve (N->L, x) ;           /* x = L'\x */
00020         cs_pvec (S->pinv, x, b, n) ;     /* b = P'*x */
00021     }
00022     cs_free (x) ;
00023     cs_sfree (S) ;
00024     cs_nfree (N) ;
00025     return (ok) ;
00026 }
```

4.45 csparse/Source/cs_compress.c File Reference

```
#include "cs.h"
```

Functions

- cs * cs_compress (const cs *T)

4.45.1 Function Documentation

4.45.1.1 cs_compress()

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

Definition at line 3 of file cs_compress.c.

4.46 cs_compress.c

Go to the documentation of this file.

```
#include "cs.h"
/* C = compressed-column form of a triplet matrix T */
cs *cs_compress (const cs *T)
{
    csi m, n, nz, p, k, *Cp, *Ci, *w, *Ti, *Tj ;
    double *Cx, *Tx ;
    cs *C ;
    if (!CS_TRIPLET (T)) return (NULL) ;                /* check inputs */
    m = T->m ; n = T->n ; Ti = T->i ; Tj = T->p ; Tx = T->x ; nz = T->nz ;
    C = cs_spalloc (m, n, nz, Tx != NULL, 0) ;          /* allocate result */
    w = cs_calloc (n, sizeof (csi)) ;                   /* get workspace */
    if (!C || !w) return (cs_done (C, w, NULL, 0)) ;    /* out of memory */
    Cp = C->p ; Ci = C->i ; Cx = C->x ;
    for (k = 0 ; k < nz ; k++) w [Tj [k]]++ ;           /* column counts */
    cs_cumsum (Cp, w, n) ;                              /* column pointers */
    for (k = 0 ; k < nz ; k++)
    {
        Ci [p = w [Tj [k]]++] = Ti [k] ;    /* A(i,j) is the pth entry in C */
        if (Cx) Cx [p] = Tx [k] ;
    }
    return (cs_done (C, w, NULL, 1)) ;      /* success; free w and return C */
}
```

4.47 csparse/Source/cs_counts.c File Reference

```
#include "cs.h"
```

Macros

- #define HEAD(k, j) (ata ? head [k] : j)
- #define NEXT(J) (ata ? next [J] : -1)

Functions

- csi * cs_counts (const cs *A, const csi *parent, const csi *post, csi ata)

4.47.1 Macro Definition Documentation

4.47.1.1 HEAD

```
#define HEAD(
                k,
                j ) (ata ? head [k] : j)
```

Definition at line 3 of file cs_counts.c.

4.47.1.2 NEXT

```
#define NEXT(
                J ) (ata ? next [J] : -1)
```

Definition at line 4 of file cs_counts.c.

4.47.2 Function Documentation

4.47.2.1 cs_counts()

```
csi * cs_counts (
            const cs * A,
            const csi * parent,
            const csi * post,
            csi ata )
```

Definition at line 17 of file cs_counts.c.

4.48 cs_counts.c

Go to the documentation of this file.

```
00001 #include "cs.h"
00002 /* column counts of LL'=A or LL'=A'A, given parent & post ordering */
00003 #define HEAD(k,j) (ata ? head [k] : j)
00004 #define NEXT(J)   (ata ? next [J] : -1)
00005 static void init_ata (cs *AT, const csi *post, csi *w, csi **head, csi **next)
00006 {
00007     csi i, k, p, m = AT->n, n = AT->m, *ATp = AT->p, *ATi = AT->i ;
00008     *head = w+4*n, *next = w+5*n+1 ;
00009     for (k = 0 ; k < n ; k++) w [post [k]] = k ;     /* invert post */
00010     for (i = 0 ; i < m ; i++)
00011     {
00012         for (k = n, p = ATp[i] ; p < ATp[i+1] ; p++) k = CS_MIN (k, w [ATi[p]]);
00013         (*next) [i] = (*head) [k] ;      /* place row i in linked list k */
00014         (*head) [k] = i ;
00015     }
00016 }
00017 csi *cs_counts (const cs *A, const csi *parent, const csi *post, csi ata)
00018 {
00019     csi i, j, k, n, m, J, s, p, q, jleaf, *ATp, *ATi, *maxfirst, *prevleaf,
00020         *ancestor, *head = NULL, *next = NULL, *colcount, *w, *first, *delta ;
00021     cs *AT ;
00022     if (!CS_CSC (A) || !parent || !post) return (NULL) ;    /* check inputs */
00023     m = A->m ; n = A->n ;
00024     s = 4*n + (ata ? (n+m+1) : 0) ;
```

```
    delta = colcount = cs_malloc (n, sizeof (csi)) ;    /* allocate result */
    w = cs_malloc (s, sizeof (csi)) ;                   /* get workspace */
    AT = cs_transpose (A, 0) ;                          /* AT = A' */
    if (!AT || !colcount || !w) return (cs_idone (colcount, AT, w, 0)) ;
    ancestor = w ; maxfirst = w+n ; prevleaf = w+2*n ; first = w+3*n ;
    for (k = 0 ; k < s ; k++) w [k] = -1 ;      /* clear workspace w [0..s-1] */
    for (k = 0 ; k < n ; k++)                   /* find first [j] */
    {
        j = post [k] ;
        delta [j] = (first [j] == -1) ? 1 : 0 ;  /* delta[j]=1 if j is a leaf */
        for ( ; j != -1 && first [j] == -1 ; j = parent [j]) first [j] = k ;
    }
    ATp = AT->p ; ATi = AT->i ;
    if (ata) init_ata (AT, post, w, &head, &next) ;
    for (i = 0 ; i < n ; i++) ancestor [i] = i ; /* each node in its own set */
    for (k = 0 ; k < n ; k++)
    {
        j = post [k] ;          /* j is the kth node in postordered etree */
        if (parent [j] != -1) delta [parent [j]]-- ;    /* j is not a root */
        for (J = HEAD (k,j) ; J != -1 ; J = NEXT (J))   /* J=j for LL'=A case */
        {
            for (p = ATp [J] ; p < ATp [J+1] ; p++)
            {
                i = ATi [p] ;
                q = cs_leaf (i, j, first, maxfirst, prevleaf, ancestor, &jleaf);
                if (jleaf >= 1) delta [j]++ ;   /* A(i,j) is in skeleton */
                if (jleaf == 2) delta [q]-- ;   /* account for overlap in q */
            }
        }
        if (parent [j] != -1) ancestor [j] = parent [j] ;
    }
    for (j = 0 ; j < n ; j++)           /* sum up delta's of each child */
    {
        if (parent [j] != -1) colcount [parent [j]] += colcount [j] ;
    }
    return (cs_idone (colcount, AT, w, 1)) ;    /* success: free workspace */
}
```

4.49 csparse/Source/cs_cumsum.c File Reference

```
#include "cs.h"
```

Functions

- double cs_cumsum (csi *p, csi *c, csi n)

4.49.1 Function Documentation

4.49.1.1 cs_cumsum()

```
double cs_cumsum (
        csi * p,
        csi * c,
        csi n )
```

Definition at line 3 of file cs_cumsum.c.

4.50 cs_cumsum.c

Go to the documentation of this file.

```
#include "cs.h"
/* p [0..n] = cumulative sum of c [0..n-1], and then copy p [0..n-1] into c */
double cs_cumsum (csi *p, csi *c, csi n)
{
    csi i, nz = 0 ;
    double nz2 = 0 ;
    if (!p || !c) return (-1) ;     /* check inputs */
    for (i = 0 ; i < n ; i++)
    {
        p [i] = nz ;
        nz += c [i] ;
        nz2 += c [i] ;              /* also in double to avoid csi overflow */
        c [i] = p [i] ;             /* also copy p[0..n-1] back into c[0..n-1]*/
    }
    p [n] = nz ;
    return (nz2) ;                  /* return sum (c [0..n-1]) */
}
```

4.51 csparse/Source/cs_dfs.c File Reference

```
#include "cs.h"
```

Functions

- csi cs_dfs (csi j, cs *G, csi top, csi *xi, csi *pstack, const csi *pinv)

4.51.1 Function Documentation

4.51.1.1 cs_dfs()

```
csi cs_dfs (
            csi j,
            cs * G,
            csi top,
            csi * xi,
            csi * pstack,
            const csi * pinv )
```

Definition at line 3 of file cs_dfs.c.

4.52 cs_dfs.c

Go to the documentation of this file.

```
#include "cs.h"
/* depth-first-search of the graph of a matrix, starting at node j */
csi cs_dfs (csi j, cs *G, csi top, csi *xi, csi *pstack, const csi *pinv)
{
    csi i, p, p2, done, jnew, head = 0, *Gp, *Gi ;
    if (!CS_CSC (G) || !xi || !pstack) return (-1) ;    /* check inputs */
    Gp = G->p ; Gi = G->i ;
    xi [0] = j ;                /* initialize the recursion stack */
    while (head >= 0)
    {
        j = xi [head] ;         /* get j from the top of the recursion stack */
        jnew = pinv ? (pinv [j]) : j ;
        if (!CS_MARKED (Gp, j))
        {
            CS_MARK (Gp, j) ;       /* mark node j as visited */
            pstack [head] = (jnew < 0) ? 0 : CS_UNFLIP (Gp [jnew]) ;
        }
        done = 1 ;                  /* node j done if no unvisited neighbors */
        p2 = (jnew < 0) ? 0 : CS_UNFLIP (Gp [jnew+1]) ;
        for (p = pstack [head] ; p < p2 ; p++)  /* examine all neighbors of j */
        {
            i = Gi [p] ;            /* consider neighbor node i */
            if (CS_MARKED (Gp, i)) continue ;   /* skip visited node i */
            pstack [head] = p ;     /* pause depth-first search of node j */
            xi [++head] = i ;       /* start dfs at node i */
            done = 0 ;              /* node j is not done */
            break ;                 /* break, to start dfs (i) */
        }
        if (done)               /* depth-first search at node j is done */
        {
            head-- ;            /* remove j from the recursion stack */
            xi [--top] = j ;    /* and place in the output stack */
        }
    }
    return (top) ;
}
```

4.53 csparse/Source/cs_dmperm.c File Reference

`#include "cs.h"`

Functions

- csd * cs_dmperm (const cs *A, csi seed)

4.53.1 Function Documentation

4.53.1.1 cs_dmperm()

```
csd * cs_dmperm (
            const cs * A,
            csi seed )
```

Definition at line 68 of file cs_dmperm.c.

4.54 cs_dmperm.c

Go to the documentation of this file.

```
#include "cs.h"
/* breadth-first search for coarse decomposition (C0,C1,R1 or R0,R3,C3) */
static csi cs_bfs (const cs *A, csi n, csi *wi, csi *wj, csi *queue,
    const csi *imatch, const csi *jmatch, csi mark)
{
    csi *Ap, *Ai, head = 0, tail = 0, j, i, p, j2 ;
    cs *C ;
    for (j = 0 ; j < n ; j++)           /* place all unmatched nodes in queue */
    {
        if (imatch [j] >= 0) continue ; /* skip j if matched */
        wj [j] = 0 ;                    /* j in set C0 (R0 if transpose) */
        queue [tail++] = j ;            /* place unmatched col j in queue */
    }
    if (tail == 0) return (1) ;         /* quick return if no unmatched nodes */
    C = (mark == 1) ? ((cs *) A) : cs_transpose (A, 0) ;
    if (!C) return (0) ;                /* bfs of C=A' to find R3,C3 from R0 */
    Ap = C->p ; Ai = C->i ;
    while (head < tail)                 /* while queue is not empty */
    {
        j = queue [head++] ;            /* get the head of the queue */
        for (p = Ap [j] ; p < Ap [j+1] ; p++)
        {
            i = Ai [p] ;
            if (wi [i] >= 0) continue ; /* skip if i is marked */
            wi [i] = mark ;             /* i in set R1 (C3 if transpose) */
            j2 = jmatch [i] ;           /* traverse alternating path to j2 */
            if (wj [j2] >= 0) continue ;/* skip j2 if it is marked */
            wj [j2] = mark ;            /* j2 in set C1 (R3 if transpose) */
            queue [tail++] = j2 ;       /* add j2 to queue */
        }
    }
    if (mark != 1) cs_spfree (C) ;      /* free A' if it was created */
    return (1) ;
}

/* collect matched rows and columns into p and q */
static void cs_matched (csi n, const csi *wj, const csi *imatch, csi *p, csi *q,
    csi *cc, csi *rr, csi set, csi mark)
{
    csi kc = cc [set], j ;
    csi kr = rr [set-1] ;
    for (j = 0 ; j < n ; j++)
    {
        if (wj [j] != mark) continue ;      /* skip if j is not in C set */
        p [kr++] = imatch [j] ;
        q [kc++] = j ;
    }
    cc [set+1] = kc ;
    rr [set] = kr ;
}

/* collect unmatched rows into the permutation vector p */
static void cs_unmatched (csi m, const csi *wi, csi *p, csi *rr, csi set)
{
    csi i, kr = rr [set] ;
    for (i = 0 ; i < m ; i++) if (wi [i] == 0) p [kr++] = i ;
    rr [set+1] = kr ;
}

/* return 1 if row i is in R2 */
static csi cs_rprune (csi i, csi j, double aij, void *other)
{
    csi *rr = (csi *) other ;
    return (i >= rr [1] && i < rr [2]) ;
}

/* Given A, compute coarse and then fine dmperm */
csd *cs_dmperm (const cs *A, csi seed)
{
    csi m, n, i, j, k, cnz, nc, *jmatch, *imatch, *wi, *wj, *pinv, *Cp, *Ci,
        *ps, *rs, nb1, nb2, *p, *q, *cc, *rr, *r, *s, ok ;
    cs *C ;
    csd *D, *scc ;
    /* --- Maximum matching ------------------------------------------------- */
    if (!CS_CSC (A)) return (NULL) ;            /* check inputs */
    m = A->m ; n = A->n ;
    D = cs_dalloc (m, n) ;                      /* allocate result */
    if (!D) return (NULL) ;
    p = D->p ; q = D->q ; r = D->r ; s = D->s ; cc = D->cc ; rr = D->rr ;
    jmatch = cs_maxtrans (A, seed) ;            /* max transversal */
    imatch = jmatch + m ;                       /* imatch = inverse of jmatch */
    if (!jmatch) return (cs_ddone (D, NULL, jmatch, 0)) ;
```

```
    /* --- Coarse decomposition --------------------------------------------- */
    wi = r ; wj = s ;                               /* use r and s as workspace */
    for (j = 0 ; j < n ; j++) wj [j] = -1 ;         /* unmark all cols for bfs */
    for (i = 0 ; i < m ; i++) wi [i] = -1 ;         /* unmark all rows for bfs */
    cs_bfs (A, n, wi, wj, q, imatch, jmatch, 1) ;        /* find C1, R1 from C0*/
    ok = cs_bfs (A, m, wj, wi, p, jmatch, imatch, 3) ;  /* find R3, C3 from R0*/
    if (!ok) return (cs_ddone (D, NULL, jmatch, 0)) ;
    cs_unmatched (n, wj, q, cc, 0) ;                     /* unmatched set C0 */
    cs_matched (n, wj, imatch, p, q, cc, rr, 1, 1) ;     /* set R1 and C1 */
    cs_matched (n, wj, imatch, p, q, cc, rr, 2, -1) ;    /* set R2 and C2 */
    cs_matched (n, wj, imatch, p, q, cc, rr, 3, 3) ;     /* set R3 and C3 */
    cs_unmatched (m, wi, p, rr, 3) ;                     /* unmatched set R0 */
    cs_free (jmatch) ;
    /* --- Fine decomposition ----------------------------------------------- */
    pinv = cs_pinv (p, m) ;         /* pinv=p' */
    if (!pinv) return (cs_ddone (D, NULL, NULL, 0)) ;
    C = cs_permute (A, pinv, q, 0) ;/* C=A(p,q) (it will hold A(R2,C2)) */
    cs_free (pinv) ;
    if (!C) return (cs_ddone (D, NULL, NULL, 0)) ;
    Cp = C->p ;
    nc = cc [3] - cc [2] ;          /* delete cols C0, C1, and C3 from C */
    if (cc [2] > 0) for (j = cc [2] ; j <= cc [3] ; j++) Cp [j-cc[2]] = Cp [j] ;
    C->n = nc ;
    if (rr [2] - rr [1] < m)        /* delete rows R0, R1, and R3 from C */
    {
        cs_fkeep (C, cs_rprune, rr) ;
        cnz = Cp [nc] ;
        Ci = C->i ;
        if (rr [1] > 0) for (k = 0 ; k < cnz ; k++) Ci [k] -= rr [1] ;
    }
    C->m = nc ;
    scc = cs_scc (C) ;              /* find strongly connected components of C*/
    if (!scc) return (cs_ddone (D, C, NULL, 0)) ;
    /* --- Combine coarse and fine decompositions --------------------------- */
    ps = scc->p ;                   /* C(ps,ps) is the permuted matrix */
    rs = scc->r ;                   /* kth block is rs[k]..rs[k+1]-1 */
    nb1 = scc->nb  ;                /* # of blocks of A(R2,C2) */
    for (k = 0 ; k < nc ; k++) wj [k] = q [ps [k] + cc [2]] ;
    for (k = 0 ; k < nc ; k++) q [k + cc [2]] = wj [k] ;
    for (k = 0 ; k < nc ; k++) wi [k] = p [ps [k] + rr [1]] ;
    for (k = 0 ; k < nc ; k++) p [k + rr [1]] = wi [k] ;
    nb2 = 0 ;                       /* create the fine block partitions */
    r [0] = s [0] = 0 ;
    if (cc [2] > 0) nb2++ ;         /* leading coarse block A (R1, [C0 C1]) */
    for (k = 0 ; k < nb1 ; k++)     /* coarse block A (R2,C2) */
    {
        r [nb2] = rs [k] + rr [1] ; /* A (R2,C2) splits into nb1 fine blocks */
        s [nb2] = rs [k] + cc [2] ;
        nb2++ ;
    }
    if (rr [2] < m)
    {
        r [nb2] = rr [2] ;          /* trailing coarse block A ([R3 R0], C3) */
        s [nb2] = cc [3] ;
        nb2++ ;
    }
    r [nb2] = m ;
    s [nb2] = n ;
    D->nb = nb2 ;
    cs_dfree (scc) ;
    return (cs_ddone (D, C, NULL, 1)) ;
}
```

4.55 csparse/Source/cs_droptol.c File Reference

```
#include "cs.h"
```

Functions

- csi cs_droptol (cs *A, double tol)

4.55.1 Function Documentation

4.55.1.1 cs_droptol()

```
csi cs_droptol (
            cs * A,
            double tol )
```

Definition at line 6 of file cs_droptol.c.

4.56 cs_droptol.c

Go to the documentation of this file.

```
#include "cs.h"
static csi cs_tol (csi i, csi j, double aij, void *tol)
{
    return (fabs (aij) > *((double *) tol)) ;
}
csi cs_droptol (cs *A, double tol)
{
    return (cs_fkeep (A, &cs_tol, &tol)) ;    /* keep all large entries */
}
```

4.57 csparse/Source/cs_dropzeros.c File Reference

```
#include "cs.h"
```

Functions

- csi cs_dropzeros (cs *A)

4.57.1 Function Documentation

4.57.1.1 cs_dropzeros()

```
csi cs_dropzeros (
            cs * A )
```

Definition at line 6 of file cs_dropzeros.c.

4.58 cs_dropzeros.c

Go to the documentation of this file.

```
#include "cs.h"
static csi cs_nonzero (csi i, csi j, double aij, void *other)
{
    return (aij != 0) ;
}
csi cs_dropzeros (cs *A)
{
    return (cs_fkeep (A, &cs_nonzero, NULL)) ;  /* keep all nonzero entries */
}
```

4.59 csparse/Source/cs_dupl.c File Reference

```
#include "cs.h"
```

Functions

- csi cs_dupl (cs ∗A)

4.59.1 Function Documentation

4.59.1.1 cs_dupl()

```
csi cs_dupl (
        cs ∗ A )
```

Definition at line 3 of file cs_dupl.c.

4.60 cs_dupl.c

Go to the documentation of this file.

```
00001 #include "cs.h"
00002 /* remove duplicate entries from A */
00003 csi cs_dupl (cs *A)
00004 {
00005     csi i, j, p, q, nz = 0, n, m, *Ap, *Ai, *w ;
00006     double *Ax ;
00007     if (!CS_CSC (A)) return (0) ;                /* check inputs */
00008     m = A->m ; n = A->n ; Ap = A->p ; Ai = A->i ; Ax = A->x ;
00009     w = cs_malloc (m, sizeof (csi)) ;            /* get workspace */
00010     if (!w) return (0) ;                         /* out of memory */
00011     for (i = 0 ; i < m ; i++) w [i] = -1 ;       /* row i not yet seen */
00012     for (j = 0 ; j < n ; j++)
00013     {
00014         q = nz ;                                 /* column j will start at q */
00015         for (p = Ap [j] ; p < Ap [j+1] ; p++)
00016         {
00017             i = Ai [p] ;                         /* A(i,j) is nonzero */
00018             if (w [i] >= q)
00019             {
00020                 Ax [w [i]] += Ax [p] ;           /* A(i,j) is a duplicate */
00021             }
00022             else
00023             {
00024                 w [i] = nz ;                     /* record where row i occurs */
00025                 Ai [nz] = i ;                    /* keep A(i,j) */
00026                 Ax [nz++] = Ax [p] ;
00027             }
00028         }
00029         Ap [j] = q ;                             /* record start of column j */
00030     }
00031     Ap [n] = nz ;                                /* finalize A */
00032     cs_free (w) ;                                /* free workspace */
00033     return (cs_sprealloc (A, 0)) ;               /* remove extra space from A */
00034 }
```

4.61 csparse/Source/cs_entry.c File Reference

```
#include "cs.h"
```

Functions

- csi cs_entry (cs ∗T, csi i, csi j, double x)

4.61.1 Function Documentation

4.61.1.1 cs_entry()

```
csi cs_entry (
            cs * T,
            csi i,
            csi j,
            double x )
```

Definition at line 3 of file cs_entry.c.

4.62 cs_entry.c

Go to the documentation of this file.

```
00001 #include "cs.h"
00002 /* add an entry to a triplet matrix; return 1 if ok, 0 otherwise */
00003 csi cs_entry (cs *T, csi i, csi j, double x)
00004 {
00005     if (!CS_TRIPLET (T) || i < 0 || j < 0) return (0) ;     /* check inputs */
00006     if (T->nz >= T->nzmax && !cs_sprealloc (T,2*(T->nzmax))) return (0) ;
00007     if (T->x) T->x [T->nz] = x ;
00008     T->i [T->nz] = i ;
00009     T->p [T->nz++] = j ;
00010     T->m = CS_MAX (T->m, i+1) ;
00011     T->n = CS_MAX (T->n, j+1) ;
00012     return (1) ;
00013 }
```

4.63 csparse/Source/cs_ereach.c File Reference

```
#include "cs.h"
```

Functions

- csi cs_ereach (const cs ∗A, csi k, const csi ∗parent, csi ∗s, csi ∗w)

4.63.1 Function Documentation

4.63.1.1 cs_ereach()

```
csi cs_ereach (
            const cs * A,
            csi k,
            const csi * parent,
            csi * s,
            csi * w )
```

Definition at line 3 of file cs_ereach.c.

4.64 cs_ereach.c

Go to the documentation of this file.

```
00001 #include "cs.h"
00002 /* find nonzero pattern of Cholesky L(k,1:k-1) using etree and triu(A(:,k)) */
00003 csi cs_ereach (const cs *A, csi k, const csi *parent, csi *s, csi *w)
00004 {
00005     csi i, p, n, len, top, *Ap, *Ai ;
00006     if (!CS_CSC (A) || !parent || !s || !w) return (-1) ;   /* check inputs */
00007     top = n = A->n ; Ap = A->p ; Ai = A->i ;
00008     CS_MARK (w, k) ;                    /* mark node k as visited */
00009     for (p = Ap [k] ; p < Ap [k+1] ; p++)
00010     {
00011         i = Ai [p] ;                    /* A(i,k) is nonzero */
00012         if (i > k) continue ;           /* only use upper triangular part of A */
00013         for (len = 0 ; !CS_MARKED (w,i) ; i = parent [i]) /* traverse up etree*/
00014         {
00015             s [len++] = i ;             /* L(k,i) is nonzero */
00016             CS_MARK (w, i) ;            /* mark i as visited */
00017         }
00018         while (len > 0) s [--top] = s [--len] ; /* push path onto stack */
00019     }
00020     for (p = top ; p < n ; p++) CS_MARK (w, s [p]) ;    /* unmark all nodes */
00021     CS_MARK (w, k) ;                    /* unmark node k */
00022     return (top) ;                      /* s [top..n-1] contains pattern of L(k,:)*/
00023 }
```

4.65 csparse/Source/cs_etree.c File Reference

```
#include "cs.h"
```

Functions

- csi ∗ cs_etree (const cs ∗A, csi ata)

4.65.1 Function Documentation

4.65.1.1 cs_etree()

```
csi * cs_etree (
            const cs * A,
            csi ata )
```

Definition at line 3 of file cs_etree.c.

4.66 cs_etree.c

Go to the documentation of this file.

```
00001 #include "cs.h"
00002 /* compute the etree of A (using triu(A), or A'A without forming A'A */
00003 csi *cs_etree (const cs *A, csi ata)
00004 {
00005     csi i, k, p, m, n, inext, *Ap, *Ai, *w, *parent, *ancestor, *prev ;
00006     if (!CS_CSC (A)) return (NULL) ;        /* check inputs */
00007     m = A->m ; n = A->n ; Ap = A->p ; Ai = A->i ;
00008     parent = cs_malloc (n, sizeof (csi)) ;              /* allocate result */
00009     w = cs_malloc (n + (ata ? m : 0), sizeof (csi)) ;   /* get workspace */
00010     if (!w || !parent) return (cs_idone (parent, NULL, w, 0)) ;
00011     ancestor = w ; prev = w + n ;
00012     if (ata) for (i = 0 ; i < m ; i++) prev [i] = -1 ;
00013     for (k = 0 ; k < n ; k++)
00014     {
00015         parent [k] = -1 ;                   /* node k has no parent yet */
00016         ancestor [k] = -1 ;                 /* nor does k have an ancestor */
00017         for (p = Ap [k] ; p < Ap [k+1] ; p++)
00018         {
00019             i = ata ? (prev [Ai [p]]) : (Ai [p]) ;
00020             for ( ; i != -1 && i < k ; i = inext)   /* traverse from i to k */
00021             {
00022                 inext = ancestor [i] ;              /* inext = ancestor of i */
00023                 ancestor [i] = k ;                  /* path compression */
00024                 if (inext == -1) parent [i] = k ;   /* no anc., parent is k */
00025             }
00026             if (ata) prev [Ai [p]] = k ;
00027         }
00028     }
00029     return (cs_idone (parent, NULL, w, 1)) ;
00030 }
```

4.67 csparse/Source/cs_fkeep.c File Reference

```
#include "cs.h"
```

Functions

- csi cs_fkeep (cs ∗A, csi(∗fkeep)(csi, csi, double, void ∗), void ∗other)

4.67.1 Function Documentation

4.67.1.1 cs_fkeep()

```
csi cs_fkeep (
        cs * A,
        csi(*)(csi, csi, double, void *) fkeep,
        void * other )
```

Definition at line 3 of file cs_fkeep.c.

4.68 cs_fkeep.c

Go to the documentation of this file.

```
#include "cs.h"
/* drop entries for which fkeep(A(i,j)) is false; return nz if OK, else -1 */
csi cs_fkeep (cs *A, csi (*fkeep) (csi, csi, double, void *), void *other)
{
    csi j, p, nz = 0, n, *Ap, *Ai ;
    double *Ax ;
    if (!CS_CSC (A) || !fkeep) return (-1) ;    /* check inputs */
    n = A->n ; Ap = A->p ; Ai = A->i ; Ax = A->x ;
    for (j = 0 ; j < n ; j++)
    {
        p = Ap [j] ;                        /* get current location of col j */
        Ap [j] = nz ;                       /* record new location of col j */
        for ( ; p < Ap [j+1] ; p++)
        {
            if (fkeep (Ai [p], j, Ax ? Ax [p] : 1, other))
            {
                if (Ax) Ax [nz] = Ax [p] ;  /* keep A(i,j) */
                Ai [nz++] = Ai [p] ;
            }
        }
    }
    Ap [n] = nz ;                           /* finalize A */
    cs_sprealloc (A, 0) ;                   /* remove extra space from A */
    return (nz) ;
}
```

4.69 csparse/Source/cs_gaxpy.c File Reference

```
#include "cs.h"
```

Functions

- csi cs_gaxpy (const cs *A, const double *x, double *y)

4.69.1 Function Documentation

4.69.1.1 cs_gaxpy()

```
csi cs_gaxpy (
        const cs * A,
        const double * x,
        double * y )
```

Definition at line 3 of file cs_gaxpy.c.

4.70 cs_gaxpy.c

Go to the documentation of this file.

```
#include "cs.h"
/* y = A*x+y */
csi cs_gaxpy (const cs *A, const double *x, double *y)
{
    csi p, j, n, *Ap, *Ai ;
    double *Ax ;
    if (!CS_CSC (A) || !x || !y) return (0) ;       /* check inputs */
    n = A->n ; Ap = A->p ; Ai = A->i ; Ax = A->x ;
    for (j = 0 ; j < n ; j++)
    {
        for (p = Ap [j] ; p < Ap [j+1] ; p++)
        {
            y [Ai [p]] += Ax [p] * x [j] ;
        }
    }
    return (1) ;
}
```

4.71 csparse/Source/cs_happly.c File Reference

`#include "cs.h"`

Functions

- csi cs_happly (const cs *V, csi i, double beta, double *x)

4.71.1 Function Documentation

4.71.1.1 cs_happly()

```
csi cs_happly (
        const cs * V,
        csi i,
        double beta,
        double * x )
```

Definition at line 3 of file cs_happly.c.

4.72 cs_happly.c

Go to the documentation of this file.

```
#include "cs.h"
/* apply the ith Householder vector to x */
csi cs_happly (const cs *V, csi i, double beta, double *x)
{
    csi p, *Vp, *Vi ;
    double *Vx, tau = 0 ;
    if (!CS_CSC (V) || !x) return (0) ;     /* check inputs */
    Vp = V->p ; Vi = V->i ; Vx = V->x ;
    for (p = Vp [i] ; p < Vp [i+1] ; p++)   /* tau = v'*x */
    {
        tau += Vx [p] * x [Vi [p]] ;
    }
    tau *= beta ;                           /* tau = beta*(v'*x) */
    for (p = Vp [i] ; p < Vp [i+1] ; p++)   /* x = x - v*tau */
    {
        x [Vi [p]] -= Vx [p] * tau ;
    }
    return (1) ;
}
```

4.73 csparse/Source/cs_house.c File Reference

```
#include "cs.h"
```

Functions

- double cs_house (double *x, double *beta, csi n)

4.73.1 Function Documentation

4.73.1.1 cs_house()

```
double cs_house (
            double * x,
            double * beta,
            csi n )
```

Definition at line 4 of file cs_house.c.

4.74 cs_house.c

Go to the documentation of this file.

```
00001 #include "cs.h"
00002 /* create a Householder reflection [v,beta,s]=house(x), overwrite x with v,
00003  * where (I-beta*v*v')*x = s*e1.  See Algo 5.1.1, Golub & Van Loan, 3rd ed. */
00004 double cs_house (double *x, double *beta, csi n)
00005 {
00006     double s, sigma = 0 ;
00007     csi i ;
00008     if (!x || !beta) return (-1) ;          /* check inputs */
00009     for (i = 1 ; i < n ; i++) sigma += x [i] * x [i] ;
00010     if (sigma == 0)
00011     {
00012         s = fabs (x [0]) ;                  /* s = |x(0)| */
00013         (*beta) = (x [0] <= 0) ? 2 : 0 ;
00014         x [0] = 1 ;
00015     }
00016     else
00017     {
00018         s = sqrt (x [0] * x [0] + sigma) ;  /* s = norm (x) */
00019         x [0] = (x [0] <= 0) ? (x [0] - s) : (-sigma / (x [0] + s)) ;
00020         (*beta) = -1. / (s * x [0]) ;
00021     }
00022     return (s) ;
00023 }
```

4.75 csparse/Source/cs_ipvec.c File Reference

```
#include "cs.h"
```

Functions

- csi cs_ipvec (const csi ∗p, const double ∗b, double ∗x, csi n)

4.75.1 Function Documentation

4.75.1.1 cs_ipvec()

```
csi cs_ipvec (
            const csi * p,
            const double * b,
            double * x,
            csi n )
```

Definition at line 3 of file cs_ipvec.c.

4.76 cs_ipvec.c

Go to the documentation of this file.

```
00001 #include "cs.h"
00002 /* x(p) = b, for dense vectors x and b; p=NULL denotes identity */
00003 csi cs_ipvec (const csi *p, const double *b, double *x, csi n)
00004 {
00005     csi k ;
00006     if (!x || !b) return (0) ;                               /* check inputs */
00007     for (k = 0 ; k < n ; k++) x [p ? p [k] : k] = b [k] ;
00008     return (1) ;
00009 }
```

4.77 csparse/Source/cs_leaf.c File Reference

```
#include "cs.h"
```

Functions

- csi cs_leaf (csi i, csi j, const csi ∗first, csi ∗maxfirst, csi ∗prevleaf, csi ∗ancestor, csi ∗jleaf)

4.77.1 Function Documentation

4.77.1.1 cs_leaf()

```
csi cs_leaf (
            csi i,
            csi j,
            const csi * first,
            csi * maxfirst,
            csi * prevleaf,
            csi * ancestor,
            csi * jleaf )
```

Definition at line 3 of file cs_leaf.c.

4.78 cs_leaf.c

Go to the documentation of this file.

```
00001 #include "cs.h"
00002 /* consider A(i,j), node j in ith row subtree and return lca(jprev,j) */
00003 csi cs_leaf (csi i, csi j, const csi *first, csi *maxfirst, csi *prevleaf,
00004     csi *ancestor, csi *jleaf)
00005 {
00006     csi q, s, sparent, jprev ;
00007     if (!first || !maxfirst || !prevleaf || !ancestor || !jleaf) return (-1) ;
00008     *jleaf = 0 ;
00009     if (i <= j || first [j] <= maxfirst [i]) return (-1) ;  /* j not a leaf */
00010     maxfirst [i] = first [j] ;      /* update max first[j] seen so far */
00011     jprev = prevleaf [i] ;          /* jprev = previous leaf of ith subtree */
00012     prevleaf [i] = j ;
00013     *jleaf = (jprev == -1) ? 1: 2 ; /* j is first or subsequent leaf */
00014     if (*jleaf == 1) return (i) ;   /* if 1st leaf, q = root of ith subtree */
00015     for (q = jprev ; q != ancestor [q] ; q = ancestor [q]) ;
00016     for (s = jprev ; s != q ; s = sparent)
00017     {
00018         sparent = ancestor [s] ;    /* path compression */
00019         ancestor [s] = q ;
00020     }
00021     return (q) ;                    /* q = least common ancester (jprev,j) */
00022 }
```

4.79 csparse/Source/cs_load.c File Reference

```
#include "cs.h"
```

Functions

- cs * cs_load (FILE *f)

4.79.1 Function Documentation

4.79.1.1 cs_load()

```
cs * cs_load (
            FILE * f )
```

Definition at line 3 of file cs_load.c.

4.80 cs_load.c

Go to the documentation of this file.

```
#include "cs.h"
/* load a triplet matrix from a file */
cs *cs_load (FILE *f)
{
    double i, j ;   /* use double for integers to avoid csi conflicts */
    double x ;
    cs *T ;
    if (!f) return (NULL) ;                             /* check inputs */
    T = cs_spalloc (0, 0, 1, 1, 1) ;                    /* allocate result */
    while (fscanf (f, "%lg %lg %lg\n", &i, &j, &x) == 3)
    {
        if (!cs_entry (T, (csi) i, (csi) j, x)) return (cs_spfree (T)) ;
    }
    return (T) ;
}
```

4.81 csparse/Source/cs_lsolve.c File Reference

```
#include "cs.h"
```

Functions

- csi cs_lsolve (const cs *L, double *x)

4.81.1 Function Documentation

4.81.1.1 cs_lsolve()

```
csi cs_lsolve (
        const cs * L,
        double * x )
```

Definition at line 3 of file cs_lsolve.c.

4.82 cs_lsolve.c

Go to the documentation of this file.

```
#include "cs.h"
/* solve Lx=b where x and b are dense.  x=b on input, solution on output. */
csi cs_lsolve (const cs *L, double *x)
{
    csi p, j, n, *Lp, *Li ;
    double *Lx ;
    if (!CS_CSC (L) || !x) return (0) ;                     /* check inputs */
    n = L->n ; Lp = L->p ; Li = L->i ; Lx = L->x ;
    for (j = 0 ; j < n ; j++)
    {
        x [j] /= Lx [Lp [j]] ;
        for (p = Lp [j]+1 ; p < Lp [j+1] ; p++)
        {
            x [Li [p]] -= Lx [p] * x [j] ;
        }
    }
    return (1) ;
}
```

4.83 csparse/Source/cs_ltsolve.c File Reference

```
#include "cs.h"
```

Functions

- csi cs_ltsolve (const cs *L, double *x)

4.83.1 Function Documentation

4.83.1.1 cs_ltsolve()

```
csi cs_ltsolve (
            const cs * L,
            double * x )
```

Definition at line 3 of file cs_ltsolve.c.

4.84 cs_ltsolve.c

Go to the documentation of this file.

```
00001 #include "cs.h"
00002 /* solve L'x=b where x and b are dense.  x=b on input, solution on output. */
00003 csi cs_ltsolve (const cs *L, double *x)
00004 {
00005     csi p, j, n, *Lp, *Li ;
00006     double *Lx ;
00007     if (!CS_CSC (L) || !x) return (0) ;                     /* check inputs */
00008     n = L->n ; Lp = L->p ; Li = L->i ; Lx = L->x ;
00009     for (j = n-1 ; j >= 0 ; j--)
00010     {
00011         for (p = Lp [j]+1 ; p < Lp [j+1] ; p++)
00012         {
00013             x [j] -= Lx [p] * x [Li [p]] ;
00014         }
00015         x [j] /= Lx [Lp [j]] ;
00016     }
00017     return (1) ;
00018 }
```

4.85 csparse/Source/cs_lu.c File Reference

```
#include "cs.h"
```

Functions

- csn * cs_lu (const cs *A, const css *S, double tol)

4.85.1 Function Documentation

4.85.1.1 cs_lu()

```
csn * cs_lu (
            const cs * A,
            const css * S,
            double tol )
```

Definition at line 3 of file cs_lu.c.

4.86 cs_lu.c

Go to the documentation of this file.

```
00001 #include "cs.h"
00002 /* [L,U,pinv]=lu(A, [q lnz unz]). lnz and unz can be guess */
00003 csn *cs_lu (const cs *A, const css *S, double tol)
00004 {
00005     cs *L, *U ;
00006     csn *N ;
00007     double pivot, *Lx, *Ux, *x,  a, t ;
00008     csi *Lp, *Li, *Up, *Ui, *pinv, *xi, *q, n, ipiv, k, top, p, i, col, lnz,unz;
00009     if (!CS_CSC (A) || !S) return (NULL) ;          /* check inputs */
00010     n = A->n ;
00011     q = S->q ; lnz = S->lnz ; unz = S->unz ;
00012     x = cs_malloc (n, sizeof (double)) ;            /* get double workspace */
00013     xi = cs_malloc (2*n, sizeof (csi)) ;            /* get csi workspace */
00014     N = cs_calloc (1, sizeof (csn)) ;               /* allocate result */
00015     if (!x || !xi || !N) return (cs_ndone (N, NULL, xi, x, 0)) ;
00016     N->L = L = cs_spalloc (n, n, lnz, 1, 0) ;       /* allocate result L */
00017     N->U = U = cs_spalloc (n, n, unz, 1, 0) ;       /* allocate result U */
00018     N->pinv = pinv = cs_malloc (n, sizeof (csi)) ;  /* allocate result pinv */
00019     if (!L || !U || !pinv) return (cs_ndone (N, NULL, xi, x, 0)) ;
00020     Lp = L->p ; Up = U->p ;
00021     for (i = 0 ; i < n ; i++) x [i] = 0 ;           /* clear workspace */
00022     for (i = 0 ; i < n ; i++) pinv [i] = -1 ;       /* no rows pivotal yet */
00023     for (k = 0 ; k <= n ; k++) Lp [k] = 0 ;         /* no cols of L yet */
00024     lnz = unz = 0 ;
00025     for (k = 0 ; k < n ; k++)       /* compute L(:,k) and U(:,k) */
00026     {
00027         /* --- Triangular solve --------------------------------------------- */
00028         Lp [k] = lnz ;              /* L(:,k) starts here */
00029         Up [k] = unz ;              /* U(:,k) starts here */
00030         if ((lnz + n > L->nzmax && !cs_sprealloc (L, 2*L->nzmax + n)) ||
00031             (unz + n > U->nzmax && !cs_sprealloc (U, 2*U->nzmax + n)))
00032         {
00033             return (cs_ndone (N, NULL, xi, x, 0)) ;
00034         }
00035         Li = L->i ; Lx = L->x ; Ui = U->i ; Ux = U->x ;
00036         col = q ? (q [k]) : k ;
00037         top = cs_spsolve (L, A, col, xi, x, pinv, 1) ;  /* x = L\A(:,col) */
00038         /* --- Find pivot --------------------------------------------------- */
00039         ipiv = -1 ;
00040         a = -1 ;
00041         for (p = top ; p < n ; p++)
00042         {
00043             i = xi [p] ;            /* x(i) is nonzero */
00044             if (pinv [i] < 0)       /* row i is not yet pivotal */
00045             {
00046                 if ((t = fabs (x [i])) > a)
00047                 {
00048                     a = t ;         /* largest pivot candidate so far */
00049                     ipiv = i ;
00050                 }
00051             }
00052             else                    /* x(i) is the entry U(pinv[i],k) */
00053             {
00054                 Ui [unz] = pinv [i] ;
00055                 Ux [unz++] = x [i] ;
00056             }
```

```
        }
        if (ipiv == -1 || a <= 0) return (cs_ndone (N, NULL, xi, x, 0)) ;
        /* tol=1 for  partial pivoting; tol<1 gives preference to diagonal */
        if (pinv [col] < 0 && fabs (x [col]) >= a*tol) ipiv = col ;
        /* --- Divide by pivot ---------------------------------------------- */
        pivot = x [ipiv] ;          /* the chosen pivot */
        Ui [unz] = k ;              /* last entry in U(:,k) is U(k,k) */
        Ux [unz++] = pivot ;
        pinv [ipiv] = k ;           /* ipiv is the kth pivot row */
        Li [lnz] = ipiv ;           /* first entry in L(:,k) is L(k,k) = 1 */
        Lx [lnz++] = 1 ;
        for (p = top ; p < n ; p++) /* L(k+1:n,k) = x / pivot */
        {
            i = xi [p] ;
            if (pinv [i] < 0)       /* x(i) is an entry in L(:,k) */
            {
                Li [lnz] = i ;      /* save unpermuted row in L */
                Lx [lnz++] = x [i] / pivot ;    /* scale pivot column */
            }
            x [i] = 0 ;             /* x [0..n-1] = 0 for next k */
        }
    }
    /* --- Finalize L and U ------------------------------------------------- */
    Lp [n] = lnz ;
    Up [n] = unz ;
    Li = L->i ;                     /* fix row indices of L for final pinv */
    for (p = 0 ; p < lnz ; p++) Li [p] = pinv [Li [p]] ;
    cs_sprealloc (L, 0) ;           /* remove extra space from L and U */
    cs_sprealloc (U, 0) ;
    return (cs_ndone (N, NULL, xi, x, 1)) ;     /* success */
}
```

4.87 csparse/Source/cs_lusol.c File Reference

```
#include "cs.h"
```

Functions

- csi cs_lusol (csi order, const cs ∗A, double ∗b, double tol)

4.87.1 Function Documentation

4.87.1.1 cs_lusol()

```
csi cs_lusol (
        csi order,
        const cs * A,
        double * b,
        double tol )
```

Definition at line 3 of file cs_lusol.c.

4.88 cs_lusol.c

Go to the documentation of this file.

```
#include "cs.h"
/* x=A\b where A is unsymmetric; b overwritten with solution */
csi cs_lusol (csi order, const cs *A, double *b, double tol)
{
    double *x ;
    css *S ;
    csn *N ;
    csi n, ok ;
    if (!CS_CSC (A) || !b) return (0) ;     /* check inputs */
    n = A->n ;
    S = cs_sqr (order, A, 0) ;              /* ordering and symbolic analysis */
    N = cs_lu (A, S, tol) ;                 /* numeric LU factorization */
    x = cs_malloc (n, sizeof (double)) ;    /* get workspace */
    ok = (S && N && x) ;
    if (ok)
    {
        cs_ipvec (N->pinv, b, x, n) ;       /* x = b(p) */
        cs_lsolve (N->L, x) ;               /* x = L\x */
        cs_usolve (N->U, x) ;               /* x = U\x */
        cs_ipvec (S->q, x, b, n) ;          /* b(q) = x */
    }
    cs_free (x) ;
    cs_sfree (S) ;
    cs_nfree (N) ;
    return (ok) ;
}
```

4.89 csparse/Source/cs_malloc.c File Reference

```
#include "cs.h"
```

Functions

- void * cs_malloc (csi n, size_t size)
- void * cs_calloc (csi n, size_t size)
- void * cs_free (void *p)
- void * cs_realloc (void *p, csi n, size_t size, csi *ok)

4.89.1 Function Documentation

4.89.1.1 cs_calloc()

```
void * cs_calloc (
            csi n,
            size_t size )
```

Definition at line 16 of file cs_malloc.c.

4.89.1.2 cs_free()

```
void * cs_free (
            void * p )
```

Definition at line 22 of file cs_malloc.c.

4.89.1.3 cs_malloc()

```
void * cs_malloc (
            csi n,
            size_t size )
```

Definition at line 10 of file cs_malloc.c.

4.89.1.4 cs_realloc()

```
void * cs_realloc (
            void * p,
            csi n,
            size_t size,
            csi * ok )
```

Definition at line 29 of file cs_malloc.c.

4.90 cs_malloc.c

Go to the documentation of this file.

```
00001 #include "cs.h"
00002 #ifdef MATLAB_MEX_FILE
00003 #define malloc mxMalloc
00004 #define free mxFree
00005 #define realloc mxRealloc
00006 #define calloc mxCalloc
00007 #endif
00008
00009 /* wrapper for malloc */
00010 void *cs_malloc (csi n, size_t size)
00011 {
00012     return (malloc (CS_MAX (n,1) * size)) ;
00013 }
00014
00015 /* wrapper for calloc */
00016 void *cs_calloc (csi n, size_t size)
00017 {
00018     return (calloc (CS_MAX (n,1), size)) ;
00019 }
00020
00021 /* wrapper for free */
00022 void *cs_free (void *p)
00023 {
00024     if (p) free (p) ;       /* free p if it is not already NULL */
00025     return (NULL) ;         /* return NULL to simplify the use of cs_free */
00026 }
00027
00028 /* wrapper for realloc */
00029 void *cs_realloc (void *p, csi n, size_t size, csi *ok)
00030 {
00031     void *pnew ;
00032     pnew = realloc (p, CS_MAX (n,1) * size) ; /* realloc the block */
00033     *ok = (pnew != NULL) ;                    /* realloc fails if pnew is NULL */
00034     return ((*ok) ? pnew : p) ;               /* return original p if failure */
00035 }
```

4.91 csparse/Source/cs_maxtrans.c File Reference

```
#include "cs.h"
```

Functions

- csi * cs_maxtrans (const cs *A, csi seed)

4.91.1 Function Documentation

4.91.1.1 cs_maxtrans()

```
csi * cs_maxtrans (
            const cs * A,
            csi seed )
```

Definition at line 44 of file cs_maxtrans.c.

4.92 cs_maxtrans.c

Go to the documentation of this file.

```
#include "cs.h"
/* find an augmenting path starting at column k and extend the match if found */
static void cs_augment (csi k, const cs *A, csi *jmatch, csi *cheap, csi *w,
        csi *js, csi *is, csi *ps)
{
    csi found = 0, p, i = -1, *Ap = A->p, *Ai = A->i, head = 0, j ;
    js [0] = k ;                        /* start with just node k in jstack */
    while (head >= 0)
    {
        /* --- Start (or continue) depth-first-search at node j ------------- */
        j = js [head] ;                 /* get j from top of jstack */
        if (w [j] != k)                 /* 1st time j visited for kth path */
        {
            w [j] = k ;                 /* mark j as visited for kth path */
            for (p = cheap [j] ; p < Ap [j+1] && !found ; p++)
            {
                i = Ai [p] ;            /* try a cheap assignment (i,j) */
                found = (jmatch [i] == -1) ;
            }
            cheap [j] = p ;             /* start here next time j is traversed*/
            if (found)
            {
                is [head] = i ;         /* column j matched with row i */
                break ;                 /* end of augmenting path */
            }
            ps [head] = Ap [j] ;        /* no cheap match: start dfs for j */
        }
        /* --- Depth-first-search of neighbors of j ------------------------- */
        for (p = ps [head] ; p < Ap [j+1] ; p++)
        {
            i = Ai [p] ;                /* consider row i */
            if (w [jmatch [i]] == k) continue ; /* skip jmatch [i] if marked */
            ps [head] = p + 1 ;         /* pause dfs of node j */
            is [head] = i ;             /* i will be matched with j if found */
            js [++head] = jmatch [i] ;  /* start dfs at column jmatch [i] */
            break ;
        }
        if (p == Ap [j+1]) head-- ;     /* node j is done; pop from stack */
    }                                   /* augment the match if path found: */
```

```
    if (found) for (p = head ; p >= 0 ; p--) jmatch [is [p]] = js [p] ;
}

/* find a maximum transveral */
csi *cs_maxtrans (const cs *A, csi seed)  /*[jmatch [0..m-1]; imatch [0..n-1]]*/
{
    csi i, j, k, n, m, p, n2 = 0, m2 = 0, *Ap, *jimatch, *w, *cheap, *js, *is,
        *ps, *Ai, *Cp, *jmatch, *imatch, *q ;
    cs *C ;
    if (!CS_CSC (A)) return (NULL) ;                /* check inputs */
    n = A->n ; m = A->m ; Ap = A->p ; Ai = A->i ;
    w = jimatch = cs_calloc (m+n, sizeof (csi)) ;   /* allocate result */
    if (!jimatch) return (NULL) ;
    for (k = 0, j = 0 ; j < n ; j++)     /* count nonempty rows and columns */
    {
        n2 += (Ap [j] < Ap [j+1]) ;
        for (p = Ap [j] ; p < Ap [j+1] ; p++)
        {
            w [Ai [p]] = 1 ;
            k += (j == Ai [p]) ;         /* count entries already on diagonal */
        }
    }
    if (k == CS_MIN (m,n))               /* quick return if diagonal zero-free */
    {
        jmatch = jimatch ; imatch = jimatch + m ;
        for (i = 0 ; i < k ; i++) jmatch [i] = i ;
        for (      ; i < m ; i++) jmatch [i] = -1 ;
        for (j = 0 ; j < k ; j++) imatch [j] = j ;
        for (      ; j < n ; j++) imatch [j] = -1 ;
        return (cs_idone (jimatch, NULL, NULL, 1)) ;
    }
    for (i = 0 ; i < m ; i++) m2 += w [i] ;
    C = (m2 < n2) ? cs_transpose (A,0) : ((cs *) A) ; /* transpose if needed */
    if (!C) return (cs_idone (jimatch, (m2 < n2) ? C : NULL, NULL, 0)) ;
    n = C->n ; m = C->m ; Cp = C->p ;
    jmatch = (m2 < n2) ? jimatch + n : jimatch ;
    imatch = (m2 < n2) ? jimatch : jimatch + m ;
    w = cs_malloc (5*n, sizeof (csi)) ;             /* get workspace */
    if (!w) return (cs_idone (jimatch, (m2 < n2) ? C : NULL, w, 0)) ;
    cheap = w + n ; js = w + 2*n ; is = w + 3*n ; ps = w + 4*n ;
    for (j = 0 ; j < n ; j++) cheap [j] = Cp [j] ;  /* for cheap assignment */
    for (j = 0 ; j < n ; j++) w [j] = -1 ;          /* all columns unflagged */
    for (i = 0 ; i < m ; i++) jmatch [i] = -1 ;     /* nothing matched yet */
    q = cs_randperm (n, seed) ;                     /* q = random permutation */
    for (k = 0 ; k < n ; k++)    /* augment, starting at column q[k] */
    {
        cs_augment (q ? q [k]: k, C, jmatch, cheap, w, js, is, ps) ;
    }
    cs_free (q) ;
    for (j = 0 ; j < n ; j++) imatch [j] = -1 ;     /* find row match */
    for (i = 0 ; i < m ; i++) if (jmatch [i] >= 0) imatch [jmatch [i]] = i ;
    return (cs_idone (jimatch, (m2 < n2) ? C : NULL, w, 1)) ;
}
```

4.93 csparse/Source/cs_multiply.c File Reference

```
#include "cs.h"
```

Functions

- cs * cs_multiply (const cs *A, const cs *B)

4.93.1 Function Documentation

4.93.1.1 cs_multiply()

```
cs * cs_multiply (
            const cs * A,
            const cs * B )
```

Definition at line 3 of file cs_multiply.c.

4.94 cs_multiply.c

Go to the documentation of this file.

```
#include "cs.h"
/* C = A*B */
cs *cs_multiply (const cs *A, const cs *B)
{
    csi p, j, nz = 0, anz, *Cp, *Ci, *Bp, m, n, bnz, *w, values, *Bi ;
    double *x, *Bx, *Cx ;
    cs *C ;
    if (!CS_CSC (A) || !CS_CSC (B)) return (NULL) ;      /* check inputs */
    if (A->n != B->m) return (NULL) ;
    m = A->m ; anz = A->p [A->n] ;
    n = B->n ; Bp = B->p ; Bi = B->i ; Bx = B->x ; bnz = Bp [n] ;
    w = cs_calloc (m, sizeof (csi)) ;                    /* get workspace */
    values = (A->x != NULL) && (Bx != NULL) ;
    x = values ? cs_malloc (m, sizeof (double)) : NULL ; /* get workspace */
    C = cs_spalloc (m, n, anz + bnz, values, 0) ;        /* allocate result */
    if (!C || !w || (values && !x)) return (cs_done (C, w, x, 0)) ;
    Cp = C->p ;
    for (j = 0 ; j < n ; j++)
    {
        if (nz + m > C->nzmax && !cs_sprealloc (C, 2*(C->nzmax)+m))
        {
            return (cs_done (C, w, x, 0)) ;             /* out of memory */
        }
        Ci = C->i ; Cx = C->x ;         /* C->i and C->x may be reallocated */
        Cp [j] = nz ;                   /* column j of C starts here */
        for (p = Bp [j] ; p < Bp [j+1] ; p++)
        {
            nz = cs_scatter (A, Bi [p], Bx ? Bx [p] : 1, w, x, j+1, C, nz) ;
        }
        if (values) for (p = Cp [j] ; p < nz ; p++) Cx [p] = x [Ci [p]] ;
    }
    Cp [n] = nz ;                       /* finalize the last column of C */
    cs_sprealloc (C, 0) ;               /* remove extra space from C */
    return (cs_done (C, w, x, 1)) ;     /* success; free workspace, return C */
}
```

4.95 csparse/Source/cs_norm.c File Reference

```
#include "cs.h"
```

Functions

- double cs_norm (const cs *A)

4.95.1 Function Documentation

4.95.1.1 cs_norm()

```
double cs_norm (
            const cs * A )
```

Definition at line 3 of file cs_norm.c.

4.96 cs_norm.c

Go to the documentation of this file.

```
#include "cs.h"
/* 1-norm of a sparse matrix = max (sum (abs (A))), largest column sum */
double cs_norm (const cs *A)
{
    csi p, j, n, *Ap ;
    double *Ax,  norm = 0, s ;
    if (!CS_CSC (A) || !A->x) return (-1) ;             /* check inputs */
    n = A->n ; Ap = A->p ; Ax = A->x ;
    for (j = 0 ; j < n ; j++)
    {
        for (s = 0, p = Ap [j] ; p < Ap [j+1] ; p++) s += fabs (Ax [p]) ;
        norm = CS_MAX (norm, s) ;
    }
    return (norm) ;
}
```

4.97 csparse/Source/cs_permute.c File Reference

```
#include "cs.h"
```

Functions

- cs * cs_permute (const cs *A, const csi *pinv, const csi *q, csi values)

4.97.1 Function Documentation

4.97.1.1 cs_permute()

```
cs * cs_permute (
            const cs * A,
            const csi * pinv,
            const csi * q,
            csi values )
```

Definition at line 3 of file cs_permute.c.

4.98 cs_permute.c

Go to the documentation of this file.

```
#include "cs.h"
/* C = A(p,q) where p and q are permutations of 0..m-1 and 0..n-1. */
cs *cs_permute (const cs *A, const csi *pinv, const csi *q, csi values)
{
    csi t, j, k, nz = 0, m, n, *Ap, *Ai, *Cp, *Ci ;
    double *Cx, *Ax ;
    cs *C ;
    if (!CS_CSC (A)) return (NULL) ;    /* check inputs */
    m = A->m ; n = A->n ; Ap = A->p ; Ai = A->i ; Ax = A->x ;
    C = cs_spalloc (m, n, Ap [n], values && Ax != NULL, 0) ;  /* alloc result */
    if (!C) return (cs_done (C, NULL, NULL, 0)) ;   /* out of memory */
    Cp = C->p ; Ci = C->i ; Cx = C->x ;
    for (k = 0 ; k < n ; k++)
    {
        Cp [k] = nz ;                   /* column k of C is column q[k] of A */
        j = q ? (q [k]) : k ;
        for (t = Ap [j] ; t < Ap [j+1] ; t++)
        {
            if (Cx) Cx [nz] = Ax [t] ;  /* row i of A is row pinv[i] of C */
            Ci [nz++] = pinv ? (pinv [Ai [t]]) : Ai [t] ;
        }
    }
    Cp [n] = nz ;                       /* finalize the last column of C */
    return (cs_done (C, NULL, NULL, 1)) ;
}
```

4.99 csparse/Source/cs_pinv.c File Reference

```
#include "cs.h"
```

Functions

- csi ∗ cs_pinv (csi const ∗p, csi n)

4.99.1 Function Documentation

4.99.1.1 cs_pinv()

```
csi * cs_pinv (
        csi const * p,
        csi n )
```

Definition at line 3 of file cs_pinv.c.

4.100 cs_pinv.c

Go to the documentation of this file.

```
#include "cs.h"
/* pinv = p', or p = pinv' */
csi *cs_pinv (csi const *p, csi n)
{
    csi k, *pinv ;
    if (!p) return (NULL) ;                     /* p = NULL denotes identity */
    pinv = cs_malloc (n, sizeof (csi)) ;        /* allocate result */
    if (!pinv) return (NULL) ;                  /* out of memory */
    for (k = 0 ; k < n ; k++) pinv [p [k]] = k ;/* invert the permutation */
    return (pinv) ;                             /* return result */
}
```

4.101 csparse/Source/cs_post.c File Reference

```
#include "cs.h"
```

Functions

- csi * cs_post (const csi *parent, csi n)

4.101.1 Function Documentation

4.101.1.1 cs_post()

```
csi * cs_post (
            const csi * parent,
            csi n )
```

Definition at line 3 of file cs_post.c.

4.102 cs_post.c

Go to the documentation of this file.

```
#include "cs.h"
/* post order a forest */
csi *cs_post (const csi *parent, csi n)
{
    csi j, k = 0, *post, *w, *head, *next, *stack ;
    if (!parent) return (NULL) ;                            /* check inputs */
    post = cs_malloc (n, sizeof (csi)) ;                    /* allocate result */
    w = cs_malloc (3*n, sizeof (csi)) ;                     /* get workspace */
    if (!w || !post) return (cs_idone (post, NULL, w, 0)) ;
    head = w ; next = w + n ; stack = w + 2*n ;
    for (j = 0 ; j < n ; j++) head [j] = -1 ;             /* empty linked lists */
    for (j = n-1 ; j >= 0 ; j--)            /* traverse nodes in reverse order*/
    {
        if (parent [j] == -1) continue ;    /* j is a root */
        next [j] = head [parent [j]] ;      /* add j to list of its parent */
        head [parent [j]] = j ;
    }
    for (j = 0 ; j < n ; j++)
    {
        if (parent [j] != -1) continue ;    /* skip j if it is not a root */
        k = cs_tdfs (j, k, head, next, post, stack) ;
    }
    return (cs_idone (post, NULL, w, 1)) ;  /* success; free w, return post */
}
```

4.103 csparse/Source/cs_print.c File Reference

```
#include "cs.h"
```

Functions

- csi cs_print (const cs ∗A, csi brief)

4.103.1 Function Documentation

4.103.1.1 cs_print()

```
csi cs_print (
            const cs * A,
            csi brief )
```

Definition at line 3 of file cs_print.c.

4.104 cs_print.c

Go to the documentation of this file.

```
#include "cs.h"
/* print a sparse matrix; use %g for integers to avoid differences with csi */
csi cs_print (const cs *A, csi brief)
{
    csi p, j, m, n, nzmax, nz, *Ap, *Ai ;
    double *Ax ;
    if (!A) { printf ("(null)\n") ; return (0) ; }
    m = A->m ; n = A->n ; Ap = A->p ; Ai = A->i ; Ax = A->x ;
    nzmax = A->nzmax ; nz = A->nz ;
    printf ("CSparse Version %d.%d.%d, %s.  %s\n", CS_VER, CS_SUBVER,
        CS_SUBSUB, CS_DATE, CS_COPYRIGHT) ;
    if (nz < 0)
    {
        printf ("%g-by-%g, nzmax: %g nnz: %g, 1-norm: %g\n", (double) m,
            (double) n, (double) nzmax, (double) (Ap [n]), cs_norm (A)) ;
        for (j = 0 ; j < n ; j++)
        {
            printf ("    col %g : locations %g to %g\n", (double) j,
                (double) (Ap [j]), (double) (Ap [j+1]-1)) ;
            for (p = Ap [j] ; p < Ap [j+1] ; p++)
            {
                printf ("      %g : %g\n", (double) (Ai [p]), Ax ? Ax [p] : 1) ;
                if (brief && p > 20) { printf ("  ...\n") ; return (1) ; }
            }
        }
    }
    else
    {
        printf ("triplet: %g-by-%g, nzmax: %g nnz: %g\n", (double) m,
            (double) n, (double) nzmax, (double) nz) ;
        for (p = 0 ; p < nz ; p++)
        {
            printf ("    %g %g : %g\n", (double) (Ai [p]), (double) (Ap [p]),
                Ax ? Ax [p] : 1) ;
            if (brief && p > 20) { printf ("  ...\n") ; return (1) ; }
        }
    }
    return (1) ;
}
```

4.105 csparse/Source/cs_pvec.c File Reference

```
#include "cs.h"
```

Functions

- csi cs_pvec (const csi *p, const double *b, double *x, csi n)

4.105.1 Function Documentation

4.105.1.1 cs_pvec()

```
csi cs_pvec (
            const csi * p,
            const double * b,
            double * x,
            csi n )
```

Definition at line 3 of file cs_pvec.c.

4.106 cs_pvec.c

Go to the documentation of this file.

```
00001 #include "cs.h"
00002 /* x = b(p), for dense vectors x and b; p=NULL denotes identity */
00003 csi cs_pvec (const csi *p, const double *b, double *x, csi n)
00004 {
00005     csi k ;
00006     if (!x || !b) return (0) ;                              /* check inputs */
00007     for (k = 0 ; k < n ; k++) x [k] = b [p ? p [k] : k] ;
00008     return (1) ;
00009 }
```

4.107 csparse/Source/cs_qr.c File Reference

```
#include "cs.h"
```

Functions

- csn * cs_qr (const cs *A, const css *S)

4.107.1 Function Documentation

4.107.1.1 cs_qr()

```
csn * cs_qr (
            const cs * A,
            const css * S )
```

Definition at line 3 of file cs_qr.c.

4.108 cs_qr.c

Go to the documentation of this file.

```
00001 #include "cs.h"
00002 /* sparse QR factorization [V,beta,pinv,R] = qr (A) */
00003 csn *cs_qr (const cs *A, const css *S)
00004 {
00005     double *Rx, *Vx, *Ax, *x,  *Beta ;
00006     csi i, k, p, m, n, vnz, p1, top, m2, len, col, rnz, *s, *leftmost, *Ap, *Ai,
00007         *parent, *Rp, *Ri, *Vp, *Vi, *w, *pinv, *q ;
00008     cs *R, *V ;
00009     csn *N ;
00010     if (!CS_CSC (A) || !S) return (NULL) ;
00011     m = A->m ; n = A->n ; Ap = A->p ; Ai = A->i ; Ax = A->x ;
00012     q = S->q ; parent = S->parent ; pinv = S->pinv ; m2 = S->m2 ;
00013     vnz = S->lnz ; rnz = S->unz ; leftmost = S->leftmost ;
00014     w = cs_malloc (m2+n, sizeof (csi)) ;            /* get csi workspace */
00015     x = cs_malloc (m2, sizeof (double)) ;           /* get double workspace */
00016     N = cs_calloc (1, sizeof (csn)) ;               /* allocate result */
00017     if (!w || !x || !N) return (cs_ndone (N, NULL, w, x, 0)) ;
00018     s = w + m2 ;                                    /* s is size n */
00019     for (k = 0 ; k < m2 ; k++) x [k] = 0 ;          /* clear workspace x */
00020     N->L = V = cs_spalloc (m2, n, vnz, 1, 0) ;      /* allocate result V */
00021     N->U = R = cs_spalloc (m2, n, rnz, 1, 0) ;      /* allocate result R */
00022     N->B = Beta = cs_malloc (n, sizeof (double)) ;  /* allocate result Beta */
00023     if (!R || !V || !Beta) return (cs_ndone (N, NULL, w, x, 0)) ;
00024     Rp = R->p ; Ri = R->i ; Rx = R->x ;
00025     Vp = V->p ; Vi = V->i ; Vx = V->x ;
00026     for (i = 0 ; i < m2 ; i++) w [i] = -1 ; /* clear w, to mark nodes */
00027     rnz = 0 ; vnz = 0 ;
00028     for (k = 0 ; k < n ; k++)               /* compute V and R */
00029     {
00030         Rp [k] = rnz ;                      /* R(:,k) starts here */
00031         Vp [k] = p1 = vnz ;                 /* V(:,k) starts here */
00032         w [k] = k ;                         /* add V(k,k) to pattern of V */
00033         Vi [vnz++] = k ;
00034         top = n ;
00035         col = q ? q [k] : k ;
00036         for (p = Ap [col] ; p < Ap [col+1] ; p++)   /* find R(:,k) pattern */
00037         {
00038             i = leftmost [Ai [p]] ;         /* i = min(find(A(i,q))) */
00039             for (len = 0 ; w [i] != k ; i = parent [i]) /* traverse up to k */
00040             {
00041                 s [len++] = i ;
00042                 w [i] = k ;
00043             }
00044             while (len > 0) s [--top] = s [--len] ; /* push path on stack */
00045             i = pinv [Ai [p]] ;             /* i = permuted row of A(:,col) */
00046             x [i] = Ax [p] ;                /* x (i) = A(:,col) */
00047             if (i > k && w [i] < k)         /* pattern of V(:,k) = x (k+1:m) */
00048             {
00049                 Vi [vnz++] = i ;            /* add i to pattern of V(:,k) */
00050                 w [i] = k ;
00051             }
00052         }
00053         for (p = top ; p < n ; p++) /* for each i in pattern of R(:,k) */
00054         {
00055             i = s [p] ;                     /* R(i,k) is nonzero */
00056             cs_happly (V, i, Beta [i], x) ; /* apply (V(i),Beta(i)) to x */
00057             Ri [rnz] = i ;                  /* R(i,k) = x(i) */
00058             Rx [rnz++] = x [i] ;
00059             x [i] = 0 ;
00060             if (parent [i] == k) vnz = cs_scatter (V, i, 0, w, NULL, k, V, vnz);
00061         }
00062         for (p = p1 ; p < vnz ; p++)        /* gather V(:,k) = x */
00063         {
00064             Vx [p] = x [Vi [p]] ;
00065             x [Vi [p]] = 0 ;
00066         }
```

```
        Ri [rnz] = k ;                     /* R(k,k) = norm (x) */
        Rx [rnz++] = cs_house (Vx+p1, Beta+k, vnz-p1) ; /* [v,beta]=house(x) */
    }
    Rp [n] = rnz ;                          /* finalize R */
    Vp [n] = vnz ;                          /* finalize V */
    return (cs_ndone (N, NULL, w, x, 1)) ;  /* success */
}
```

4.109 csparse/Source/cs_qrsol.c File Reference

```
#include "cs.h"
```

Functions

- csi cs_qrsol (csi order, const cs ∗A, double ∗b)

4.109.1 Function Documentation

4.109.1.1 cs_qrsol()

```
csi cs_qrsol (
        csi order,
        const cs * A,
        double * b )
```

Definition at line 3 of file cs_qrsol.c.

4.110 cs_qrsol.c

Go to the documentation of this file.

```
#include "cs.h"
/* x=A\b where A can be rectangular; b overwritten with solution */
csi cs_qrsol (csi order, const cs *A, double *b)
{
    double *x ;
    css *S ;
    csn *N ;
    cs *AT = NULL ;
    csi k, m, n, ok ;
    if (!CS_CSC (A) || !b) return (0) ; /* check inputs */
    n = A->n ;
    m = A->m ;
    if (m >= n)
    {
        S = cs_sqr (order, A, 1) ;          /* ordering and symbolic analysis */
        N = cs_qr (A, S) ;                  /* numeric QR factorization */
        x = cs_calloc (S ? S->m2 : 1, sizeof (double)) ;    /* get workspace */
        ok = (S && N && x) ;
        if (ok)
        {
            cs_ipvec (S->pinv, b, x, m) ;   /* x(0:m-1) = b(p(0:m-1) */
            for (k = 0 ; k < n ; k++)       /* apply Householder refl. to x */
            {
                cs_happly (N->L, k, N->B [k], x) ;
            }
            cs_usolve (N->U, x) ;           /* x = R\x */
```

```
            cs_ipvec (S->q, x, b, n) ;      /* b(q(0:n-1)) = x(0:n-1) */
        }
    }
    else
    {
        AT = cs_transpose (A, 1) ;          /* Ax=b is underdetermined */
        S = cs_sqr (order, AT, 1) ;         /* ordering and symbolic analysis */
        N = cs_qr (AT, S) ;                 /* numeric QR factorization of A' */
        x = cs_calloc (S ? S->m2 : 1, sizeof (double)) ;    /* get workspace */
        ok = (AT && S && N && x) ;
        if (ok)
        {
            cs_pvec (S->q, b, x, m) ;       /* x(q(0:m-1)) = b(0:m-1) */
            cs_utsolve (N->U, x) ;          /* x = R'\x */
            for (k = m-1 ; k >= 0 ; k--)    /* apply Householder refl. to x */
            {
                cs_happly (N->L, k, N->B [k], x) ;
            }
            cs_pvec (S->pinv, x, b, n) ;    /* b(0:n-1) = x(p(0:n-1)) */
        }
    }
    cs_free (x) ;
    cs_sfree (S) ;
    cs_nfree (N) ;
    cs_spfree (AT) ;
    return (ok) ;
}
```

4.111 csparse/Source/cs_randperm.c File Reference

```
#include "cs.h"
```

Functions

- csi * cs_randperm (csi n, csi seed)

4.111.1 Function Documentation

4.111.1.1 cs_randperm()

```
csi * cs_randperm (
            csi n,
            csi seed )
```

Definition at line 5 of file cs_randperm.c.

4.112 cs_randperm.c

Go to the documentation of this file.

```
#include "cs.h"
/* return a random permutation vector, the identity perm, or p = n-1:-1:0.
 * seed = -1 means p = n-1:-1:0.  seed = 0 means p = identity.  otherwise
 * p = random permutation.  */
csi *cs_randperm (csi n, csi seed)
{
    csi *p, k, j, t ;
    if (seed == 0) return (NULL) ;      /* return p = NULL (identity) */
    p = cs_malloc (n, sizeof (csi)) ;   /* allocate result */
    if (!p) return (NULL) ;             /* out of memory */
    for (k = 0 ; k < n ; k++) p [k] = n-k-1 ;
    if (seed == -1) return (p) ;        /* return reverse permutation */
    srand (seed) ;                      /* get new random number seed */
    for (k = 0 ; k < n ; k++)
    {
        j = k + (rand ( ) % (n-k)) ;    /* j = rand integer in range k to n-1 */
        t = p [j] ;                     /* swap p[k] and p[j] */
        p [j] = p [k] ;
        p [k] = t ;
    }
    return (p) ;
}
```

4.113 csparse/Source/cs_reach.c File Reference

```
#include "cs.h"
```

Functions

- csi cs_reach (cs *G, const cs *B, csi k, csi *xi, const csi *pinv)

4.113.1 Function Documentation

4.113.1.1 cs_reach()

```
csi cs_reach (
        cs * G,
        const cs * B,
        csi k,
        csi * xi,
        const csi * pinv )
```

Definition at line 4 of file cs_reach.c.

4.114 cs_reach.c

Go to the documentation of this file.

```
#include "cs.h"
/* xi [top...n-1] = nodes reachable from graph of G*P' via nodes in B(:,k).
 * xi [n...2n-1] used as workspace */
csi cs_reach (cs *G, const cs *B, csi k, csi *xi, const csi *pinv)
{
    csi p, n, top, *Bp, *Bi, *Gp ;
    if (!CS_CSC (G) || !CS_CSC (B) || !xi) return (-1) ;    /* check inputs */
    n = G->n ; Bp = B->p ; Bi = B->i ; Gp = G->p ;
    top = n ;
    for (p = Bp [k] ; p < Bp [k+1] ; p++)
    {
        if (!CS_MARKED (Gp, Bi [p]))    /* start a dfs at unmarked node i */
        {
            top = cs_dfs (Bi [p], G, top, xi, xi+n, pinv) ;
        }
    }
    for (p = top ; p < n ; p++) CS_MARK (Gp, xi [p]) ;  /* restore G */
    return (top) ;
}
```

4.115 csparse/Source/cs_scatter.c File Reference

```
#include "cs.h"
```

Functions

- csi cs_scatter (const cs *A, csi j, double beta, csi *w, double *x, csi mark, cs *C, csi nz)

4.115.1 Function Documentation

4.115.1.1 cs_scatter()

```
csi cs_scatter (
        const cs * A,
        csi j,
        double beta,
        csi * w,
        double * x,
        csi mark,
        cs * C,
        csi nz )
```

Definition at line 3 of file cs_scatter.c.

4.116 cs_scatter.c

Go to the documentation of this file.

```
#include "cs.h"
/* x = x + beta * A(:,j), where x is a dense vector and A(:,j) is sparse */
csi cs_scatter (const cs *A, csi j, double beta, csi *w, double *x, csi mark,
    cs *C, csi nz)
{
    csi i, p, *Ap, *Ai, *Ci ;
    double *Ax ;
    if (!CS_CSC (A) || !w || !CS_CSC (C)) return (-1) ;      /* check inputs */
    Ap = A->p ; Ai = A->i ; Ax = A->x ; Ci = C->i ;
    for (p = Ap [j] ; p < Ap [j+1] ; p++)
    {
        i = Ai [p] ;                            /* A(i,j) is nonzero */
        if (w [i] < mark)
        {
            w [i] = mark ;                      /* i is new entry in column j */
            Ci [nz++] = i ;                     /* add i to pattern of C(:,j) */
            if (x) x [i] = beta * Ax [p] ;      /* x(i) = beta*A(i,j) */
        }
        else if (x) x [i] += beta * Ax [p] ;    /* i exists in C(:,j) already */
    }
    return (nz) ;
}
```

4.117 csparse/Source/cs_scc.c File Reference

```
#include "cs.h"
```

Functions

- csd * cs_scc (cs *A)

4.117.1 Function Documentation

4.117.1.1 cs_scc()

```
csd * cs_scc (
        cs * A )
```

Definition at line 3 of file cs_scc.c.

4.118 cs_scc.c

Go to the documentation of this file.

```
#include "cs.h"
/* find the strongly connected components of a square matrix */
csd *cs_scc (cs *A)     /* matrix A temporarily modified, then restored */
{
    csi n, i, k, b, nb = 0, top, *xi, *pstack, *p, *r, *Ap, *ATp, *rcopy, *Blk ;
    cs *AT ;
    csd *D ;
    if (!CS_CSC (A)) return (NULL) ;                /* check inputs */
    n = A->n ; Ap = A->p ;
    D = cs_dalloc (n, 0) ;                          /* allocate result */
    AT = cs_transpose (A, 0) ;                      /* AT = A' */
    xi = cs_malloc (2*n+1, sizeof (csi)) ;          /* get workspace */
    if (!D || !AT || !xi) return (cs_ddone (D, AT, xi, 0)) ;
    Blk = xi ; rcopy = pstack = xi + n ;
    p = D->p ; r = D->r ; ATp = AT->p ;
    top = n ;
    for (i = 0 ; i < n ; i++)   /* first dfs(A) to find finish times (xi) */
    {
        if (!CS_MARKED (Ap, i)) top = cs_dfs (i, A, top, xi, pstack, NULL) ;
    }
    for (i = 0 ; i < n ; i++) CS_MARK (Ap, i) ; /* restore A; unmark all nodes*/
    top = n ;
    nb = n ;
    for (k = 0 ; k < n ; k++)   /* dfs(A') to find strongly connnected comp */
    {
        i = xi [k] ;            /* get i in reverse order of finish times */
        if (CS_MARKED (ATp, i)) continue ;  /* skip node i if already ordered */
        r [nb--] = top ;        /* node i is the start of a component in p */
        top = cs_dfs (i, AT, top, p, pstack, NULL) ;
    }
    r [nb] = 0 ;                /* first block starts at zero; shift r up */
    for (k = nb ; k <= n ; k++) r [k-nb] = r [k] ;
    D->nb = nb = n-nb ;         /* nb = # of strongly connected components */
    for (b = 0 ; b < nb ; b++)  /* sort each block in natural order */
    {
        for (k = r [b] ; k < r [b+1] ; k++) Blk [p [k]] = b ;
    }
    for (b = 0 ; b <= nb ; b++) rcopy [b] = r [b] ;
    for (i = 0 ; i < n ; i++) p [rcopy [Blk [i]]++] = i ;
    return (cs_ddone (D, AT, xi, 1)) ;
}
```

4.119 csparse/Source/cs_schol.c File Reference

```
#include "cs.h"
```

Functions

- css * cs_schol (csi order, const cs *A)

4.119.1 Function Documentation

4.119.1.1 cs_schol()

```
css * cs_schol (
        csi order,
        const cs * A )
```

Definition at line 3 of file cs_schol.c.

4.120 cs_schol.c

Go to the documentation of this file.

```
#include "cs.h"
/* ordering and symbolic analysis for a Cholesky factorization */
css *cs_schol (csi order, const cs *A)
{
    csi n, *c, *post, *P ;
    cs *C ;
    css *S ;
    if (!CS_CSC (A)) return (NULL) ;        /* check inputs */
    n = A->n ;
    S = cs_calloc (1, sizeof (css)) ;       /* allocate result S */
    if (!S) return (NULL) ;                 /* out of memory */
    P = cs_amd (order, A) ;                 /* P = amd(A+A'), or natural */
    S->pinv = cs_pinv (P, n) ;              /* find inverse permutation */
    cs_free (P) ;
    if (order && !S->pinv) return (cs_sfree (S)) ;
    C = cs_symperm (A, S->pinv, 0) ;        /* C = spones(triu(A(P,P))) */
    S->parent = cs_etree (C, 0) ;           /* find etree of C */
    post = cs_post (S->parent, n) ;         /* postorder the etree */
    c = cs_counts (C, S->parent, post, 0) ; /* find column counts of chol(C) */
    cs_free (post) ;
    cs_spfree (C) ;
    S->cp = cs_malloc (n+1, sizeof (csi)) ; /* allocate result S->cp */
    S->unz = S->lnz = cs_cumsum (S->cp, c, n) ; /* find column pointers for L */
    cs_free (c) ;
    return ((S->lnz >= 0) ? S : cs_sfree (S)) ;
}
```

4.121 csparse/Source/cs_spsolve.c File Reference

```
#include "cs.h"
```

Functions

- csi cs_spsolve (cs ∗G, const cs ∗B, csi k, csi ∗xi, double ∗x, const csi ∗pinv, csi lo)

4.121.1 Function Documentation

4.121.1.1 cs_spsolve()

```
csi cs_spsolve (
        cs * G,
        const cs * B,
        csi k,
        csi * xi,
        double * x,
        const csi * pinv,
        csi lo )
```

Definition at line 3 of file cs_spsolve.c.

4.122 cs_spsolve.c

Go to the documentation of this file.

```
#include "cs.h"
/* solve Gx=b(:,k), where G is either upper (lo=0) or lower (lo=1) triangular */
csi cs_spsolve (cs *G, const cs *B, csi k, csi *xi, double *x, const csi *pinv,
    csi lo)
{
    csi j, J, p, q, px, top, n, *Gp, *Gi, *Bp, *Bi ;
    double *Gx, *Bx ;
    if (!CS_CSC (G) || !CS_CSC (B) || !xi || !x) return (-1) ;
    Gp = G->p ; Gi = G->i ; Gx = G->x ; n = G->n ;
    Bp = B->p ; Bi = B->i ; Bx = B->x ;
    top = cs_reach (G, B, k, xi, pinv) ;        /* xi[top..n-1]=Reach(B(:,k)) */
    for (p = top ; p < n ; p++) x [xi [p]] = 0 ;    /* clear x */
    for (p = Bp [k] ; p < Bp [k+1] ; p++) x [Bi [p]] = Bx [p] ; /* scatter B */
    for (px = top ; px < n ; px++)
    {
        j = xi [px] ;                               /* x(j) is nonzero */
        J = pinv ? (pinv [j]) : j ;                 /* j maps to col J of G */
        if (J < 0) continue ;                       /* column J is empty */
        x [j] /= Gx [lo ? (Gp [J]) : (Gp [J+1]-1)] ;/* x(j) /= G(j,j) */
        p = lo ? (Gp [J]+1) : (Gp [J]) ;            /* lo: L(j,j) 1st entry */
        q = lo ? (Gp [J+1]) : (Gp [J+1]-1) ;        /* up: U(j,j) last entry */
        for ( ; p < q ; p++)
        {
            x [Gi [p]] -= Gx [p] * x [j] ;          /* x(i) -= G(i,j) * x(j) */
        }
    }
    return (top) ;                                  /* return top of stack */
}
```

4.123 csparse/Source/cs_sqr.c File Reference

```
#include "cs.h"
```

Functions

- css * cs_sqr (csi order, const cs *A, csi qr)

4.123.1 Function Documentation

4.123.1.1 cs_sqr()

```
css * cs_sqr (
        csi order,
        const cs * A,
        csi qr )
```

Definition at line 60 of file cs_sqr.c.

4.124 cs_sqr.c

Go to the documentation of this file.

```
#include "cs.h"
/* compute nnz(V) = S->lnz, S->pinv, S->leftmost, S->m2 from A and S->parent */
static csi cs_vcount (const cs *A, css *S)
{
    csi i, k, p, pa, n = A->n, m = A->m, *Ap = A->p, *Ai = A->i, *next, *head,
        *tail, *nque, *pinv, *leftmost, *w, *parent = S->parent ;
    S->pinv = pinv = cs_malloc (m+n, sizeof (csi)) ;        /* allocate pinv, */
    S->leftmost = leftmost = cs_malloc (m, sizeof (csi)) ;  /* and leftmost */
    w = cs_malloc (m+3*n, sizeof (csi)) ;   /* get workspace */
    if (!pinv || !w || !leftmost)
    {
        cs_free (w) ;                       /* pinv and leftmost freed later */
        return (0) ;                        /* out of memory */
    }
    next = w ; head = w + m ; tail = w + m + n ; nque = w + m + 2*n ;
    for (k = 0 ; k < n ; k++) head [k] = -1 ;   /* queue k is empty */
    for (k = 0 ; k < n ; k++) tail [k] = -1 ;
    for (k = 0 ; k < n ; k++) nque [k] = 0 ;
    for (i = 0 ; i < m ; i++) leftmost [i] = -1 ;
    for (k = n-1 ; k >= 0 ; k--)
    {
        for (p = Ap [k] ; p < Ap [k+1] ; p++)
        {
            leftmost [Ai [p]] = k ;         /* leftmost[i] = min(find(A(i,:)))*/
        }
    }
    for (i = m-1 ; i >= 0 ; i--)            /* scan rows in reverse order */
    {
        pinv [i] = -1 ;                     /* row i is not yet ordered */
        k = leftmost [i] ;
        if (k == -1) continue ;             /* row i is empty */
        if (nque [k]++ == 0) tail [k] = i ; /* first row in queue k */
        next [i] = head [k] ;               /* put i at head of queue k */
        head [k] = i ;
    }
    S->lnz = 0 ;
    S->m2 = m ;
    for (k = 0 ; k < n ; k++)               /* find row permutation and nnz(V)*/
    {
        i = head [k] ;                      /* remove row i from queue k */
        S->lnz++ ;                          /* count V(k,k) as nonzero */
        if (i < 0) i = S->m2++ ;            /* add a fictitious row */
        pinv [i] = k ;                      /* associate row i with V(:,k) */
        if (--nque [k] <= 0) continue ;     /* skip if V(k+1:m,k) is empty */
        S->lnz += nque [k] ;                /* nque [k] is nnz (V(k+1:m,k)) */
        if ((pa = parent [k]) != -1)        /* move all rows to parent of k */
        {
            if (nque [pa] == 0) tail [pa] = tail [k] ;
            next [tail [k]] = head [pa] ;
            head [pa] = next [i] ;
            nque [pa] += nque [k] ;
        }
    }
    for (i = 0 ; i < m ; i++) if (pinv [i] < 0) pinv [i] = k++ ;
    cs_free (w) ;
    return (1) ;
}

/* symbolic ordering and analysis for QR or LU */
css *cs_sqr (csi order, const cs *A, csi qr)
{
    csi n, k, ok = 1, *post ;
    css *S ;
    if (!CS_CSC (A)) return (NULL) ;        /* check inputs */
    n = A->n ;
    S = cs_calloc (1, sizeof (css)) ;       /* allocate result S */
    if (!S) return (NULL) ;                 /* out of memory */
    S->q = cs_amd (order, A) ;              /* fill-reducing ordering */
    if (order && !S->q) return (cs_sfree (S)) ;
    if (qr)                                 /* QR symbolic analysis */
    {
        cs *C = order ? cs_permute (A, NULL, S->q, 0) : ((cs *) A) ;
        S->parent = cs_etree (C, 1) ;       /* etree of C'*C, where C=A(:,q) */
        post = cs_post (S->parent, n) ;
        S->cp = cs_counts (C, S->parent, post, 1) ;  /* col counts chol(C'*C) */
        cs_free (post) ;
        ok = C && S->parent && S->cp && cs_vcount (C, S) ;
        if (ok) for (S->unz = 0, k = 0 ; k < n ; k++) S->unz += S->cp [k] ;
        if (order) cs_spfree (C) ;
    }
    else
    {
```

```
        S->unz = 4*(A->p [n]) + n ;         /* for LU factorization only, */
        S->lnz = S->unz ;                   /* guess nnz(L) and nnz(U) */
    }
    return (ok ? S : cs_sfree (S)) ;        /* return result S */
}
```

4.125 csparse/Source/cs_symperm.c File Reference

```
#include "cs.h"
```

Functions

- cs * cs_symperm (const cs *A, const csi *pinv, csi values)

4.125.1 Function Documentation

4.125.1.1 cs_symperm()

```
cs * cs_symperm (
            const cs * A,
            const csi * pinv,
            csi values )
```

Definition at line 3 of file cs_symperm.c.

4.126 cs_symperm.c

Go to the documentation of this file.

```
00001 #include "cs.h"
00002 /* C = A(p,p) where A and C are symmetric the upper part stored; pinv not p */
00003 cs *cs_symperm (const cs *A, const csi *pinv, csi values)
00004 {
00005     csi i, j, p, q, i2, j2, n, *Ap, *Ai, *Cp, *Ci, *w ;
00006     double *Cx, *Ax ;
00007     cs *C ;
00008     if (!CS_CSC (A)) return (NULL) ;                    /* check inputs */
00009     n = A->n ; Ap = A->p ; Ai = A->i ; Ax = A->x ;
00010     C = cs_spalloc (n, n, Ap [n], values && (Ax != NULL), 0) ; /* alloc result*/
00011     w = cs_calloc (n, sizeof (csi)) ;                   /* get workspace */
00012     if (!C || !w) return (cs_done (C, w, NULL, 0)) ;    /* out of memory */
00013     Cp = C->p ; Ci = C->i ; Cx = C->x ;
00014     for (j = 0 ; j < n ; j++)           /* count entries in each column of C */
00015     {
00016         j2 = pinv ? pinv [j] : j ;      /* column j of A is column j2 of C */
00017         for (p = Ap [j] ; p < Ap [j+1] ; p++)
00018         {
00019             i = Ai [p] ;
00020             if (i > j) continue ;       /* skip lower triangular part of A */
00021             i2 = pinv ? pinv [i] : i ;  /* row i of A is row i2 of C */
00022             w [CS_MAX (i2, j2)]++ ;     /* column count of C */
00023         }
00024     }
00025     cs_cumsum (Cp, w, n) ;              /* compute column pointers of C */
00026     for (j = 0 ; j < n ; j++)
00027     {
00028         j2 = pinv ? pinv [j] : j ;      /* column j of A is column j2 of C */
```

```
        for (p = Ap [j] ; p < Ap [j+1] ; p++)
        {
            i = Ai [p] ;
            if (i > j) continue ;       /* skip lower triangular part of A*/
            i2 = pinv ? pinv [i] : i ;  /* row i of A is row i2 of C */
            Ci [q = w [CS_MAX (i2, j2)]++] = CS_MIN (i2, j2) ;
            if (Cx) Cx [q] = Ax [p] ;
        }
    }
    return (cs_done (C, w, NULL, 1)) ;  /* success; free workspace, return C */
}
```

4.127 csparse/Source/cs_tdfs.c File Reference

`#include "cs.h"`

Functions

- csi cs_tdfs (csi j, csi k, csi *head, const csi *next, csi *post, csi *stack)

4.127.1 Function Documentation

4.127.1.1 cs_tdfs()

```
csi cs_tdfs (
        csi j,
        csi k,
        csi * head,
        const csi * next,
        csi * post,
        csi * stack )
```

Definition at line 3 of file cs_tdfs.c.

4.128 cs_tdfs.c

Go to the documentation of this file.

```
00001 #include "cs.h"
00002 /* depth-first search and postorder of a tree rooted at node j */
00003 csi cs_tdfs (csi j, csi k, csi *head, const csi *next, csi *post, csi *stack)
00004 {
00005     csi i, p, top = 0 ;
00006     if (!head || !next || !post || !stack) return (-1) ;    /* check inputs */
00007     stack [0] = j ;                 /* place j on the stack */
00008     while (top >= 0)                /* while (stack is not empty) */
00009     {
00010         p = stack [top] ;           /* p = top of stack */
00011         i = head [p] ;              /* i = youngest child of p */
00012         if (i == -1)
00013         {
00014             top-- ;                 /* p has no unordered children left */
00015             post [k++] = p ;        /* node p is the kth postordered node */
00016         }
00017         else
00018         {
00019             head [p] = next [i] ;   /* remove i from children of p */
00020             stack [++top] = i ;     /* start dfs on child node i */
00021         }
00022     }
00023     return (k) ;
00024 }
```

4.129 csparse/Source/cs_transpose.c File Reference

```
#include "cs.h"
```

Functions

- cs ∗ cs_transpose (const cs ∗A, csi values)

4.129.1 Function Documentation

4.129.1.1 cs_transpose()

```
cs * cs_transpose (
            const cs * A,
            csi values )
```

Definition at line 3 of file cs_transpose.c.

4.130 cs_transpose.c

Go to the documentation of this file.

```
00001 #include "cs.h"
00002 /* C = A' */
00003 cs *cs_transpose (const cs *A, csi values)
00004 {
00005     csi p, q, j, *Cp, *Ci, n, m, *Ap, *Ai, *w ;
00006     double *Cx, *Ax ;
00007     cs *C ;
00008     if (!CS_CSC (A)) return (NULL) ;    /* check inputs */
00009     m = A->m ; n = A->n ; Ap = A->p ; Ai = A->i ; Ax = A->x ;
00010     C = cs_spalloc (n, m, Ap [n], values && Ax, 0) ;       /* allocate result */
00011     w = cs_calloc (m, sizeof (csi)) ;                      /* get workspace */
00012     if (!C || !w) return (cs_done (C, w, NULL, 0)) ;       /* out of memory */
00013     Cp = C->p ; Ci = C->i ; Cx = C->x ;
00014     for (p = 0 ; p < Ap [n] ; p++) w [Ai [p]]++ ;          /* row counts */
00015     cs_cumsum (Cp, w, m) ;                                 /* row pointers */
00016     for (j = 0 ; j < n ; j++)
00017     {
00018         for (p = Ap [j] ; p < Ap [j+1] ; p++)
00019         {
00020             Ci [q = w [Ai [p]]++] = j ; /* place A(i,j) as entry C(j,i) */
00021             if (Cx) Cx [q] = Ax [p] ;
00022         }
00023     }
00024     return (cs_done (C, w, NULL, 1)) ;  /* success; free w and return C */
00025 }
```

4.131 csparse/Source/cs_updown.c File Reference

```
#include "cs.h"
```

Functions

- csi cs_updown (cs ∗L, csi sigma, const cs ∗C, const csi ∗parent)

4.131.1 Function Documentation

4.131.1.1 cs_updown()

```
csi cs_updown (
        cs * L,
        csi sigma,
        const cs * C,
        const csi * parent )
```

Definition at line 3 of file cs_updown.c.

4.132 cs_updown.c

Go to the documentation of this file.

```
#include "cs.h"
/* sparse Cholesky update/downdate, L*L' + sigma*w*w' (sigma = +1 or -1) */
csi cs_updown (cs *L, csi sigma, const cs *C, const csi *parent)
{
    csi n, p, f, j, *Lp, *Li, *Cp, *Ci ;
    double *Lx, *Cx, alpha, beta = 1, delta, gamma, w1, w2, *w, beta2 = 1 ;
    if (!CS_CSC (L) || !CS_CSC (C) || !parent) return (0) ;  /* check inputs */
    Lp = L->p ; Li = L->i ; Lx = L->x ; n = L->n ;
    Cp = C->p ; Ci = C->i ; Cx = C->x ;
    if ((p = Cp [0]) >= Cp [1]) return (1) ;        /* return if C empty */
    w = cs_malloc (n, sizeof (double)) ;            /* get workspace */
    if (!w) return (0) ;                            /* out of memory */
    f = Ci [p] ;
    for ( ; p < Cp [1] ; p++) f = CS_MIN (f, Ci [p]) ;  /* f = min (find (C)) */
    for (j = f ; j != -1 ; j = parent [j]) w [j] = 0 ;  /* clear workspace w */
    for (p = Cp [0] ; p < Cp [1] ; p++) w [Ci [p]] = Cx [p] ; /* w = C */
    for (j = f ; j != -1 ; j = parent [j])          /* walk path f up to root */
    {
        p = Lp [j] ;
        alpha = w [j] / Lx [p] ;                    /* alpha = w(j) / L(j,j) */
        beta2 = beta*beta + sigma*alpha*alpha ;
        if (beta2 <= 0) break ;                     /* not positive definite */
        beta2 = sqrt (beta2) ;
        delta = (sigma > 0) ? (beta / beta2) : (beta2 / beta) ;
        gamma = sigma * alpha / (beta2 * beta) ;
        Lx [p] = delta * Lx [p] + ((sigma > 0) ? (gamma * w [j]) : 0) ;
        beta = beta2 ;
        for (p++ ; p < Lp [j+1] ; p++)
        {
            w1 = w [Li [p]] ;
            w [Li [p]] = w2 = w1 - alpha * Lx [p] ;
            Lx [p] = delta * Lx [p] + gamma * ((sigma > 0) ? w1 : w2) ;
        }
    }
    cs_free (w) ;
    return (beta2 > 0) ;
}
```

4.133 csparse/Source/cs_usolve.c File Reference

```
#include "cs.h"
```

Functions

- csi cs_usolve (const cs *U, double *x)

4.133.1 Function Documentation

4.133.1.1 cs_usolve()

```
csi cs_usolve (
            const cs * U,
            double * x )
```

Definition at line 3 of file cs_usolve.c.

4.134 cs_usolve.c

Go to the documentation of this file.

```
00001 #include "cs.h"
00002 /* solve Ux=b where x and b are dense.  x=b on input, solution on output. */
00003 csi cs_usolve (const cs *U, double *x)
00004 {
00005     csi p, j, n, *Up, *Ui ;
00006     double *Ux ;
00007     if (!CS_CSC (U) || !x) return (0) ;                     /* check inputs */
00008     n = U->n ; Up = U->p ; Ui = U->i ; Ux = U->x ;
00009     for (j = n-1 ; j >= 0 ; j--)
00010     {
00011         x [j] /= Ux [Up [j+1]-1] ;
00012         for (p = Up [j] ; p < Up [j+1]-1 ; p++)
00013         {
00014             x [Ui [p]] -= Ux [p] * x [j] ;
00015         }
00016     }
00017     return (1) ;
00018 }
```

4.135 csparse/Source/cs_util.c File Reference

```
#include "cs.h"
```

Functions

- cs * cs_spalloc (csi m, csi n, csi nzmax, csi values, csi triplet)
- csi cs_sprealloc (cs *A, csi nzmax)
- cs * cs_spfree (cs *A)
- csn * cs_nfree (csn *N)
- css * cs_sfree (css *S)
- csd * cs_dalloc (csi m, csi n)
- csd * cs_dfree (csd *D)
- cs * cs_done (cs *C, void *w, void *x, csi ok)
- csi * cs_idone (csi *p, cs *C, void *w, csi ok)
- csn * cs_ndone (csn *N, cs *C, void *w, void *x, csi ok)
- csd * cs_ddone (csd *D, cs *C, void *w, csi ok)

4.135.1 Function Documentation

4.135.1.1 cs_dalloc()

```
csd * cs_dalloc (
            csi m,
            csi n )
```

Definition at line 66 of file cs_util.c.

4.135.1.2 cs_ddone()

```
csd * cs_ddone (
            csd * D,
            cs * C,
            void * w,
            csi ok )
```

Definition at line 115 of file cs_util.c.

4.135.1.3 cs_dfree()

```
csd * cs_dfree (
            csd * D )
```

Definition at line 79 of file cs_util.c.

4.135.1.4 cs_done()

```
cs * cs_done (
            cs * C,
            void * w,
            void * x,
            csi ok )
```

Definition at line 90 of file cs_util.c.

4.135.1.5 cs_idone()

```
csi * cs_idone (
            csi * p,
            cs * C,
            void * w,
            csi ok )
```

Definition at line 98 of file cs_util.c.

4.135.1.6 cs_ndone()

```
csn * cs_ndone (
            csn * N,
            cs * C,
            void * w,
            void * x,
            csi ok )
```

Definition at line 106 of file cs_util.c.

4.135.1.7 cs_nfree()

```
csn * cs_nfree (
            csn * N )
```

Definition at line 43 of file cs_util.c.

4.135.1.8 cs_sfree()

```
css * cs_sfree (
            css * S )
```

Definition at line 54 of file cs_util.c.

4.135.1.9 cs_spalloc()

```
cs * cs_spalloc (
            csi m,
            csi n,
            csi nzmax,
            csi values,
            csi triplet )
```

Definition at line 3 of file cs_util.c.

4.135.1.10 cs_spfree()

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

Definition at line 33 of file cs_util.c.

4.135.1.11 cs_sprealloc()

```
csi cs_sprealloc (
            cs * A,
            csi nzmax )
```

Definition at line 18 of file cs_util.c.

4.136 cs_util.c

Go to the documentation of this file.

```
00001 #include "cs.h"
00002 /* allocate a sparse matrix (triplet form or compressed-column form) */
00003 cs *cs_spalloc (csi m, csi n, csi nzmax, csi values, csi triplet)
00004 {
00005     cs *A = cs_calloc (1, sizeof (cs)) ;    /* allocate the cs struct */
00006     if (!A) return (NULL) ;                 /* out of memory */
00007     A->m = m ;                              /* define dimensions and nzmax */
00008     A->n = n ;
00009     A->nzmax = nzmax = CS_MAX (nzmax, 1) ;
00010     A->nz = triplet ? 0 : -1 ;              /* allocate triplet or comp.col */
00011     A->p = cs_malloc (triplet ? nzmax : n+1, sizeof (csi)) ;
00012     A->i = cs_malloc (nzmax, sizeof (csi)) ;
00013     A->x = values ? cs_malloc (nzmax, sizeof (double)) : NULL ;
00014     return ((!A->p || !A->i || (values && !A->x)) ? cs_spfree (A) : A) ;
00015 }
00016
00017 /* change the max # of entries sparse matrix */
00018 csi cs_sprealloc (cs *A, csi nzmax)
00019 {
00020     csi ok, oki, okj = 1, okx = 1 ;
00021     if (!A) return (0) ;
00022     if (nzmax <= 0) nzmax = (CS_CSC (A)) ? (A->p [A->n]) : A->nz ;
00023     nzmax = CS_MAX (nzmax, 1) ;
00024     A->i = cs_realloc (A->i, nzmax, sizeof (csi), &oki) ;
00025     if (CS_TRIPLET (A)) A->p = cs_realloc (A->p, nzmax, sizeof (csi), &okj) ;
00026     if (A->x) A->x = cs_realloc (A->x, nzmax, sizeof (double), &okx) ;
00027     ok = (oki && okj && okx) ;
00028     if (ok) A->nzmax = nzmax ;
00029     return (ok) ;
00030 }
00031
00032 /* free a sparse matrix */
00033 cs *cs_spfree (cs *A)
00034 {
00035     if (!A) return (NULL) ;      /* do nothing if A already NULL */
00036     cs_free (A->p) ;
00037     cs_free (A->i) ;
00038     cs_free (A->x) ;
00039     return ((cs *) cs_free (A)) ;    /* free the cs struct and return NULL */
00040 }
00041
00042 /* free a numeric factorization */
00043 csn *cs_nfree (csn *N)
00044 {
00045     if (!N) return (NULL) ;      /* do nothing if N already NULL */
00046     cs_spfree (N->L) ;
00047     cs_spfree (N->U) ;
00048     cs_free (N->pinv) ;
00049     cs_free (N->B) ;
00050     return ((csn *) cs_free (N)) ;  /* free the csn struct and return NULL */
```

```
}

/* free a symbolic factorization */
css *cs_sfree (css *S)
{
    if (!S) return (NULL) ;     /* do nothing if S already NULL */
    cs_free (S->pinv) ;
    cs_free (S->q) ;
    cs_free (S->parent) ;
    cs_free (S->cp) ;
    cs_free (S->leftmost) ;
    return ((css *) cs_free (S)) ;  /* free the css struct and return NULL */
}

/* allocate a cs_dmperm or cs_scc result */
csd *cs_dalloc (csi m, csi n)
{
    csd *D ;
    D = cs_calloc (1, sizeof (csd)) ;
    if (!D) return (NULL) ;
    D->p = cs_malloc (m, sizeof (csi)) ;
    D->r = cs_malloc (m+6, sizeof (csi)) ;
    D->q = cs_malloc (n, sizeof (csi)) ;
    D->s = cs_malloc (n+6, sizeof (csi)) ;
    return ((!D->p || !D->r || !D->q || !D->s) ? cs_dfree (D) : D) ;
}

/* free a cs_dmperm or cs_scc result */
csd *cs_dfree (csd *D)
{
    if (!D) return (NULL) ;     /* do nothing if D already NULL */
    cs_free (D->p) ;
    cs_free (D->q) ;
    cs_free (D->r) ;
    cs_free (D->s) ;
    return ((csd *) cs_free (D)) ;  /* free the csd struct and return NULL */
}

/* free workspace and return a sparse matrix result */
cs *cs_done (cs *C, void *w, void *x, csi ok)
{
    cs_free (w) ;                       /* free workspace */
    cs_free (x) ;
    return (ok ? C : cs_spfree (C)) ;   /* return result if OK, else free it */
}

/* free workspace and return csi array result */
csi *cs_idone (csi *p, cs *C, void *w, csi ok)
{
    cs_spfree (C) ;                     /* free temporary matrix */
    cs_free (w) ;                       /* free workspace */
    return (ok ? p : (csi *) cs_free (p)) ; /* return result, or free it */
}

/* free workspace and return a numeric factorization (Cholesky, LU, or QR) */
csn *cs_ndone (csn *N, cs *C, void *w, void *x, csi ok)
{
    cs_spfree (C) ;                     /* free temporary matrix */
    cs_free (w) ;                       /* free workspace */
    cs_free (x) ;
    return (ok ? N : cs_nfree (N)) ;    /* return result if OK, else free it */
}

/* free workspace and return a csd result */
csd *cs_ddone (csd *D, cs *C, void *w, csi ok)
{
    cs_spfree (C) ;                     /* free temporary matrix */
    cs_free (w) ;                       /* free workspace */
    return (ok ? D : cs_dfree (D)) ;    /* return result if OK, else free it */
}
```

4.137 csparse/Source/cs_utsolve.c File Reference

```
#include "cs.h"
```

Functions

- csi cs_utsolve (const cs *U, double *x)

4.137.1 Function Documentation

4.137.1.1 cs_utsolve()

```
csi cs_utsolve (
            const cs * U,
            double * x )
```

Definition at line 3 of file cs_utsolve.c.

4.138 cs_utsolve.c

Go to the documentation of this file.

```
00001 #include "cs.h"
00002 /* solve U'x=b where x and b are dense.  x=b on input, solution on output. */
00003 csi cs_utsolve (const cs *U, double *x)
00004 {
00005     csi p, j, n, *Up, *Ui ;
00006     double *Ux ;
00007     if (!CS_CSC (U) || !x) return (0) ;                     /* check inputs */
00008     n = U->n ; Up = U->p ; Ui = U->i ; Ux = U->x ;
00009     for (j = 0 ; j < n ; j++)
00010     {
00011         for (p = Up [j] ; p < Up [j+1]-1 ; p++)
00012         {
00013             x [j] -= Ux [p] * x [Ui [p]] ;
00014         }
00015         x [j] /= Ux [Up [j+1]-1] ;
00016     }
00017     return (1) ;
00018 }
```

4.139 include/abip.h File Reference

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

Classes

- struct solve_specific_problem
- struct ABIP_CONE
- struct ABIP_PROBLEM_DATA
- struct ABIP_SETTINGS
- struct ABIP_SOL_VARS
- struct ABIP_INFO
- struct ABIP_WORK
- struct ABIP_RESIDUALS

Typedefs

- typedef struct ABIP_A_DATA_MATRIX ABIPMatrix
- typedef struct ABIP_LIN_SYS_WORK ABIPLinSysWork
- typedef struct ABIP_PROBLEM_DATA ABIPData
- typedef struct ABIP_SETTINGS ABIPSettings
- typedef struct ABIP_SOL_VARS ABIPSolution
- typedef struct ABIP_INFO ABIPInfo
- typedef struct ABIP_WORK ABIPWork
- typedef struct ABIP_ADAPTIVE_WORK ABIPAdaptWork
- typedef struct ABIP_RESIDUALS ABIPResiduals
- typedef struct ABIP_CONE ABIPCone
- typedef struct mkl_lin_sys MKLlinsys
- typedef struct solve_specific_problem spe_problem

Enumerations

- enum problem_type { LASSO , SVM , QCP , SVMQP }

Functions

- ABIPWork ∗ABIP() init (const ABIPData ∗d, ABIPInfo ∗info, spe_problem ∗s, ABIPCone ∗c)

  Initialize and allocate memory for the solver.
- abip_int ABIP() solve (ABIPWork ∗w, const ABIPData ∗d, ABIPSolution ∗sol, ABIPInfo ∗info, ABIPCone ∗c, spe_problem ∗s)

  Main solve iteration of the solver.
- void ABIP() finish (ABIPWork ∗w, spe_problem ∗spe)

  Free the memory allocated for the solver.
- abip_int ABIP() main (const ABIPData ∗d, ABIPSolution ∗sol, ABIPInfo ∗info)
- const char ∗ABIP() version (void)
- abip_int abip (const ABIPData ∗d, ABIPSolution ∗sol, ABIPInfo ∗info, ABIPCone ∗K)

  the main function to call the abip conic solver

4.139.1 Typedef Documentation

4.139.1.1 ABIPAdaptWork

```
typedef struct ABIP_ADAPTIVE_WORK ABIPAdaptWork
```

Definition at line 23 of file abip.h.

4.139.1.2 ABIPCone

```
typedef struct ABIP_CONE ABIPCone
```

Definition at line 25 of file abip.h.

4.139.1.3 ABIPData

```
typedef struct ABIP_PROBLEM_DATA ABIPData
```

Definition at line 18 of file abip.h.

4.139.1.4 ABIPInfo

```
typedef struct ABIP_INFO ABIPInfo
```

Definition at line 21 of file abip.h.

4.139.1.5 ABIPLinSysWork

```
typedef struct ABIP_LIN_SYS_WORK ABIPLinSysWork
```

Definition at line 16 of file abip.h.

4.139.1.6 ABIPMatrix

```
typedef struct ABIP_A_DATA_MATRIX ABIPMatrix
```

Definition at line 15 of file abip.h.

4.139.1.7 ABIPResiduals

```
typedef struct ABIP_RESIDUALS ABIPResiduals
```

Definition at line 24 of file abip.h.

4.139.1.8 ABIPSettings

```
typedef struct ABIP_SETTINGS ABIPSettings
```

Definition at line 19 of file abip.h.

4.139.1.9 ABIPSolution

```
typedef struct ABIP_SOL_VARS ABIPSolution
```

Definition at line 20 of file abip.h.

4.139.1.10 ABIPWork

```
typedef struct ABIP_WORK ABIPWork
```

Definition at line 22 of file abip.h.

4.139.1.11 MKLlinsys

```
typedef struct mkl_lin_sys MKLlinsys
```

Definition at line 26 of file abip.h.

4.139.1.12 spe_problem

```
typedef struct solve_specific_problem spe_problem
```

Definition at line 27 of file abip.h.

4.139.2 Enumeration Type Documentation

4.139.2.1 problem_type

```
enum problem_type
```

Enumerator

LASSO	
SVM	
QCP	
SVMQP	

Definition at line 13 of file abip.h.

4.139.3 Function Documentation

4.139.3.1 abip()

```
abip_int abip (
            const ABIPData * d,
            ABIPSolution * sol,
            ABIPInfo * info,
            ABIPCone * K )
```

the main function to call the abip conic solver

Definition at line 1335 of file abip.c.

4.139.3.2 finish()

```
void ABIP() finish (
            ABIPWork * w,
            spe_problem * spe )
```

Free the memory allocated for the solver.

Definition at line 1254 of file abip.c.

4.139.3.3 init()

```
ABIPWork *ABIP() init (
            const ABIPData * d,
            ABIPInfo * info,
            spe_problem * s,
            ABIPCone * c )
```

Initialize and allocate memory for the solver.

Definition at line 1271 of file abip.c.

4.139.3.4 main()

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

4.139.3.5 solve()

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

Main solve iteration of the solver.

Definition at line 1076 of file abip.c.

4.139.3.6 version()

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

Definition at line 3 of file abip_version.c.

4.140 abip.h

Go to the documentation of this file.

```
#ifndef ABIP_H_GUARD
#define ABIP_H_GUARD
#ifdef __cplusplus
extern "C" {
#endif
#include "glbopts.h"
#include <string.h>
#include "amatrix.h"
/*-----special problem to solve-----------*/
enum problem_type{LASSO, SVM, QCP, SVMQP};
typedef struct ABIP_A_DATA_MATRIX ABIPMatrix;
typedef struct ABIP_LIN_SYS_WORK ABIPLinSysWork;
typedef struct ABIP_PROBLEM_DATA ABIPData;
typedef struct ABIP_SETTINGS ABIPSettings;
typedef struct ABIP_SOL_VARS ABIPSolution;
typedef struct ABIP_INFO ABIPInfo;
typedef struct ABIP_WORK ABIPWork;
typedef struct ABIP_ADAPTIVE_WORK ABIPAdaptWork;
typedef struct ABIP_RESIDUALS ABIPResiduals;
typedef struct ABIP_CONE ABIPCone;
```

```c
typedef struct mkl_lin_sys MKLlinsys;
typedef struct solve_specific_problem spe_problem;

struct solve_specific_problem{

      enum problem_type pro_type;
      abip_int m;  //rows of input data A
      abip_int n;  //cols of input data A
      abip_int p;  //rows of ABIP constraint matrix A
      abip_int q;  //cols of ABIP constraint matrix A
      ABIPLinSysWork *L;
      ABIPSettings *stgs;
      ABIPData *data; //original data
      abip_float sparsity;

      abip_float *rho_dr; // non-identity DR scaling

      /* scaled data */
      ABIPMatrix *A;  //scaled original constraint matrix
      ABIPMatrix *Q;  //scaled original quadratic matrix
      abip_float *b;  //scaled reformulated vector for abip
      abip_float *c;  //scaled reformulated vector for abip
      /*-------------*/

      void (*scaling_data)(spe_problem *self, ABIPCone *k);
      void (*un_scaling_sol)(spe_problem *self, ABIPSolution *sol);
      void (*calc_residuals)(spe_problem *self, ABIPWork *w, ABIPResiduals *r, abip_int ipm_iter, abip_int admm_iter);
      abip_int (*init_spe_linsys_work)(spe_problem *self);
      abip_int (*solve_spe_linsys)(spe_problem *self, abip_float *b, abip_float *pcg_warm_start, abip_int iter, abip_float error_ratio);
      void (*free_spe_linsys_work)(spe_problem *self);
      void (*spe_A_times)(spe_problem *self, const abip_float *x, abip_float *y);//y += Ax, where A is the reformulated constraint matrix of ABIP
      void (*spe_AT_times)(spe_problem *self, const abip_float *x, abip_float *y);//y += A'x, where A is the reformulated constraint matrix of ABIP
      abip_float (*inner_conv_check)(spe_problem *self, ABIPWork *w);

};


/*  cols of data matrix A must be specified in this exact order
    if change the definition of this order, remember to change the order in
    'solve_sub_problem' too
*/
struct ABIP_CONE {
    abip_int* q;     /* array of second-order cone constraints */
    abip_int qsize; /* length of SOC array */
    abip_int* rq;     /* array of rotated second-order cone constraints */
    abip_int rqsize; /* length of RSOC array */
    abip_int f;      /* length of free cone */
    abip_int z;      /* length of zero cone */
    abip_int l;      /* length of LP cone */

};


struct ABIP_PROBLEM_DATA
{
      abip_int m;
      abip_int n;
      ABIPMatrix *A;
      ABIPMatrix *Q;

      abip_float *b;
      abip_float *c;

      abip_float lambda;
      ABIPSettings *stgs;
};

struct ABIP_SETTINGS
{
      abip_int normalize;
      abip_int scale_E;
      abip_int scale_bc;
      abip_float scale;
      abip_float rho_x;
      abip_float rho_y;
      abip_float rho_tau;

      abip_int max_ipm_iters;
      abip_int max_admm_iters;
      abip_float eps;
      abip_float eps_p;
      abip_float eps_d;
      abip_float eps_g;
```

```
      abip_float eps_inf;
      abip_float eps_unb;

      abip_float err_dif; //tol between max(dres,pres,dgap) of two consecutive inters

      abip_float alpha;
      abip_float cg_rate;  /* for indirect, tolerance goes down like (1/iter)^cg_rate: 2 */

      abip_int use_indirect;
      abip_int inner_check_period;
      abip_int outer_check_period;

      abip_int verbose;    /* boolean, write out progress: 1 */
      abip_int linsys_solver;   // 0:mkl_dss, 1:qdldl, 2:sparse cholesky, 3:pcg,  4:pardiso, 5:dense cholesky
      abip_int prob_type;       // 0:general_qp, 1:lasso, 2:svm, 3:qcp
      abip_float time_limit;  // in s
      abip_float psi;

      abip_int origin_scaling;
      abip_int ruiz_scaling;
      abip_int pc_scaling;

};

struct ABIP_SOL_VARS
{
      abip_float *x;
      abip_float *y;
      abip_float *s;
};

struct ABIP_INFO
{
      char status[32];
      abip_int status_val;
      abip_int ipm_iter;
      abip_int admm_iter;

      abip_float pobj;
      abip_float dobj;
      abip_float res_pri;
      abip_float res_dual;
      abip_float rel_gap;
      abip_float res_infeas;
      abip_float res_unbdd;

      abip_float setup_time;
      abip_float solve_time;
      abip_float avg_linsys_time;
      abip_float avg_cg_iters;
};


struct ABIP_WORK
{
      abip_float sigma;
      abip_float gamma;
      abip_float mu;
      abip_float beta;
      abip_float *u;
      abip_float *v;
      abip_float *v_origin;
      abip_float *u_t;
      abip_float *rel_ut;
      abip_float nm_inf_b;
      abip_float nm_inf_c;
      abip_int m;
      abip_int n;
      ABIPMatrix *A;
      abip_float *r;
      abip_float a;

};

struct ABIP_RESIDUALS
{
      abip_int last_ipm_iter;
      abip_int last_admm_iter;
      abip_float last_mu;

      abip_float res_pri;
      abip_float res_dual;
      abip_float rel_gap;
      abip_float res_infeas;
      abip_float res_unbdd;
```

```

    abip_float ct_x_by_tau;
    abip_float bt_y_by_tau;

    abip_float pobj;
    abip_float dobj;

    abip_float tau;
    abip_float kap;

    abip_float res_dif;
    abip_float error_ratio;

    abip_float Ax_b_norm;
    abip_float Qx_ATy_c_s_norm;
};


ABIPWork *ABIP(init)
(
    const ABIPData *d,
    ABIPInfo *info,
    spe_problem *s,
    ABIPCone *c
) ;
abip_int ABIP(solve)
(
   ABIPWork *w,
   const ABIPData *d,
   ABIPSolution *sol,
   ABIPInfo *info,
   ABIPCone *c,
   spe_problem *s
);
void ABIP(finish)
(
   ABIPWork *w,
   spe_problem *spe
);

abip_int ABIP(main)(const ABIPData *d, ABIPSolution *sol, ABIPInfo *info);
const char *ABIP(version)(void);
abip_int abip
(
   const ABIPData* d,
   ABIPSolution* sol,
   ABIPInfo* info,
   ABIPCone *K
);


#ifdef __cplusplus
}
#endif
#endif
```

4.141 include/amatrix.h File Reference

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

Classes

- struct ABIP_A_DATA_MATRIX

4.142 amatrix.h

Go to the documentation of this file.

```
#ifndef AMATRIX_H_GUARD
#define AMATRIX_H_GUARD

```

```
#ifdef __cplusplus
extern "C" {
#endif

#include "glbopts.h"

/* A is supplied in column compressed format */
struct ABIP_A_DATA_MATRIX
{
        abip_float *x;     /* A values, size: NNZ A */
        abip_int *i;       /* A row index, size: NNZ A */
        abip_int *p;       /* A column pointer, size: n+1 */
        abip_int m;        /* m rows, n cols */
        abip_int n;
};

#ifdef __cplusplus
}
#endif
#endif
```

4.143 include/cones.h File Reference

```
#include "glbopts.h"
#include "abip.h"
#include "linalg.h"
#include "mkl.h"
#include "mkl_lapacke.h"
```

Functions

- char *ABIP() get_cone_header (const ABIPCone *k)

  Get the number of variables and blocks of each cone.
- abip_int ABIP() validate_cones (spe_problem *spe, const ABIPCone *k)

  Check if the cone dimensions are valid.
- abip_int ABIP() get_cone_dims (const ABIPCone *k)

  Calculate the total number of dimensions of all the cones.
- void ABIP() soc_barrier_subproblem (abip_float *x, abip_float *tmp, abip_float lambda, abip_int n)

  Barrier subproblem for the second order cone.
- void ABIP() rsoc_barrier_subproblem (abip_float *x, abip_float *tmp, abip_float lambda, abip_int n)

  Barrier subproblem for the rotated second order cone.
- void ABIP() free_barrier_subproblem (abip_float *x, abip_float *tmp, abip_float lambda, abip_int n)

  Barrier subproblem for the free cone.
- void ABIP() zero_barrier_subproblem (abip_float *x, abip_float *tmp, abip_float lambda, abip_int n)

  Barrier subproblem for the zero cone.
- void ABIP() positive_orthant_barrier_subproblem (abip_float *x, abip_float *tmp, abip_float lambda, abip_int n)

  Barrier subproblem for the positive orthant cone.

4.143.1 Function Documentation

4.143.1.1 free_barrier_subproblem()

```
void ABIP() free_barrier_subproblem (
            abip_float * x,
            abip_float * tmp,
            abip_float lambda,
            abip_int n )
```

Barrier subproblem for the free cone.

Definition at line 255 of file cones.c.

4.143.1.2 get_cone_dims()

```
abip_int ABIP() get_cone_dims (
            const ABIPCone * k )
```

Calculate the total number of dimensions of all the cones.

Definition at line 8 of file cones.c.

4.143.1.3 get_cone_header()

```
char *ABIP() get_cone_header (
            const ABIPCone * k )
```

Get the number of variables and blocks of each cone.

Definition at line 87 of file cones.c.

4.143.1.4 positive_orthant_barrier_subproblem()

```
void ABIP() positive_orthant_barrier_subproblem (
            abip_float * x,
            abip_float * tmp,
            abip_float lambda,
            abip_int n )
```

Barrier subproblem for the positive orthant cone.

Definition at line 279 of file cones.c.

4.143.1.5 rsoc_barrier_subproblem()

```
void ABIP() rsoc_barrier_subproblem (
            abip_float * x,
            abip_float * tmp,
            abip_float lambda,
            abip_int n )
```

Barrier subproblem for the rotated second order cone.

Definition at line 169 of file cones.c.

4.143.1.6 soc_barrier_subproblem()

```
void ABIP() soc_barrier_subproblem (
            abip_float * x,
            abip_float * tmp,
            abip_float lambda,
            abip_int n )
```

Barrier subproblem for the second order cone.

Definition at line 130 of file cones.c.

4.143.1.7 validate_cones()

```
abip_int ABIP() validate_cones (
            spe_problem * spe,
            const ABIPCone * k )
```

Check if the cone dimensions are valid.

Definition at line 37 of file cones.c.

4.143.1.8 zero_barrier_subproblem()

```
void ABIP() zero_barrier_subproblem (
            abip_float * x,
            abip_float * tmp,
            abip_float lambda,
            abip_int n )
```

Barrier subproblem for the zero cone.

Definition at line 267 of file cones.c.

4.144 cones.h

Go to the documentation of this file.

```
#ifndef CONES_H_GUARD
#define CONES_H_GUARD

#ifdef __cplusplus
extern "C" {
#endif

#include "glbopts.h"
#include "abip.h"
#include "linalg.h"
#include "mkl.h"
#include "mkl_lapacke.h"



char* ABIP(get_cone_header)(const ABIPCone* k);
abip_int ABIP(validate_cones)(spe_problem *spe, const ABIPCone* k);

abip_int ABIP(get_cone_dims)(const ABIPCone* k);


void ABIP(soc_barrier_subproblem)(
    abip_float* x,
    abip_float* tmp,
    abip_float lambda,
    abip_int n
);

void ABIP(rsoc_barrier_subproblem)(
    abip_float* x,
    abip_float* tmp,
    abip_float lambda,
    abip_int n
);

void ABIP(free_barrier_subproblem)(
    abip_float* x,
    abip_float* tmp,
    abip_float lambda,
    abip_int n
);

void ABIP(zero_barrier_subproblem)(
    abip_float* x,
    abip_float* tmp,
    abip_float lambda,
    abip_int n
);

void ABIP(positive_orthant_barrier_subproblem)(
    abip_float* x,
    abip_float* tmp,
    abip_float lambda,
    abip_int n
);


#ifdef __cplusplus
}
#endif
#endif

```

4.145 include/ctrlc.h File Reference

Macros

- #define abip_start_interrupt_listener()
- #define abip_end_interrupt_listener()
- #define abip_is_interrupted() 0

Typedefs

- typedef int abip_make_iso_compilers_happy

4.145.1 Macro Definition Documentation

4.145.1.1 abip_end_interrupt_listener

```
#define abip_end_interrupt_listener( )
```

Definition at line 36 of file ctrlc.h.

4.145.1.2 abip_is_interrupted

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

Definition at line 37 of file ctrlc.h.

4.145.1.3 abip_start_interrupt_listener

```
#define abip_start_interrupt_listener( )
```

Definition at line 35 of file ctrlc.h.

4.145.2 Typedef Documentation

4.145.2.1 abip_make_iso_compilers_happy

```
typedef int abip_make_iso_compilers_happy
```

Definition at line 33 of file ctrlc.h.

4.146 ctrlc.h

Go to the documentation of this file.

```
/* Interface for ABIP signal handling. */

#ifndef CTRLC_H_GUARD
#define CTRLC_H_GUARD

#ifdef __cplusplus
extern "C" {
#endif

#if CTRLC > 0

#if defined MATLAB_MEX_FILE

extern int utIsInterruptPending();
extern int utSetInterruptEnabled(int);

#elif(defined _WIN32 || defined _WIN64 || defined _WINDLL)

#include <windows.h>

#else

#include <signal.h>

#endif

void abip_start_interrupt_listener(void);
void abip_end_interrupt_listener(void);
int abip_is_interrupted(void);

#else

typedef int abip_make_iso_compilers_happy;

#define abip_start_interrupt_listener()
#define abip_end_interrupt_listener()
#define abip_is_interrupted() 0

#endif

#ifdef __cplusplus
}
#endif
#endif
```

4.147 include/glbopts.h File Reference

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

Macros

- #define ABIP(x) abip_##x
- #define ABIP_VERSION ("2.0.0") /∗ string literals automatically null-terminated ∗/
- #define ABIP_INFEASIBLE_INACCURATE (-7)
- #define ABIP_UNBOUNDED_INACCURATE (-6)
- #define ABIP_SIGINT (-5)
- #define ABIP_FAILED (-4)
- #define ABIP_INDETERMINATE (-3)
- #define ABIP_INFEASIBLE (-2)
- #define ABIP_UNBOUNDED (-1)
- #define ABIP_UNFINISHED (0)

- #define ABIP_SOLVED (1)
- #define ABIP_SOLVED_INACCURATE (2)
- #define ABIP_UNSOLVED (3)
- #define SIGMA (0.8)
- #define GAMMA (1.6)
- #define MAX_IPM_ITERS (500)
- #define MAX_ADMM_ITERS (10000000)
- #define EPS (1E-3)
- #define ALPHA (1.8)
- #define CG_RATE (2.0)
- #define CG_BEST_TOL (1e-9)
- #define CG_MIN_TOL (1e-5)
- #define NORMALIZE (1)
- #define SCALE (1.0)
- #define SPARSITY_RATIO (0.01)
- #define RHO_Y (1E-3)
- #define ADAPTIVE (1)
- #define EPS_COR (0.2)
- #define EPS_PEN (0.1)
- #define ADAPTIVE_LOOKBACK (20)
- #define VERBOSE (1)
- #define WARM_START (0)
- #define CONE_TOL (1e-8)
- #define abip_printf printf
- #define _abip_free free
- #define _abip_malloc malloc
- #define _abip_calloc calloc
- #define _abip_realloc realloc
- #define abip_free(x)
- #define abip_malloc(x) _abip_malloc(x)
- #define abip_calloc(x, y) _abip_calloc(x, y)
- #define abip_realloc(x, y) _abip_realloc(x, y)
- #define NAN ((abip_float)0x7ff8000000000000)
- #define INFINITY NAN
- #define ABIP_NULL 0
- #define MAX(a, b) (((a) > (b)) ? (a) : (b))
- #define MIN(a, b) (((a) < (b)) ? (a) : (b))
- #define ABS(x) (((x) < 0) ? -(x) : (x))
- #define POWF pow
- #define SQRTF sqrt
- #define DEBUG_FUNC
- #define RETURN return
- #define EPS_TOL (1E-18)
- #define SAFEDIV_POS(X, Y) ((Y) < EPS_TOL ? ((X) / EPS_TOL) : (X) / (Y))
- #define CONVERGED_INTERVAL (1)
- #define INDETERMINATE_TOL (1e-9)

Typedefs

- typedef int abip_int
- typedef double abip_float

4.147.1 Macro Definition Documentation

4.147.1.1 _abip_calloc

```
#define _abip_calloc calloc
```

Definition at line 80 of file glbopts.h.

4.147.1.2 _abip_free

```
#define _abip_free free
```

Definition at line 78 of file glbopts.h.

4.147.1.3 _abip_malloc

```
#define _abip_malloc malloc
```

Definition at line 79 of file glbopts.h.

4.147.1.4 _abip_realloc

```
#define _abip_realloc realloc
```

Definition at line 81 of file glbopts.h.

4.147.1.5 ABIP

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

Definition at line 13 of file glbopts.h.

4.147.1.6 abip_calloc

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

Definition at line 88 of file glbopts.h.

4.147.1.7 ABIP_FAILED

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

Definition at line 23 of file glbopts.h.

4.147.1.8 abip_free

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

Value:

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

Definition at line 84 of file glbopts.h.

4.147.1.9 ABIP_INDETERMINATE

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

Definition at line 24 of file glbopts.h.

4.147.1.10 ABIP_INFEASIBLE

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

Definition at line 25 of file glbopts.h.

4.147.1.11 ABIP_INFEASIBLE_INACCURATE

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

Definition at line 20 of file glbopts.h.

4.147.1.12 abip_malloc

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

Definition at line 87 of file glbopts.h.

4.147.1.13 ABIP_NULL

```
#define ABIP_NULL 0
```

Definition at line 135 of file glbopts.h.

4.147.1.14 abip_printf

```
#define abip_printf printf
```

Definition at line 77 of file glbopts.h.

4.147.1.15 abip_realloc

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

Definition at line 89 of file glbopts.h.

4.147.1.16 ABIP_SIGINT

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

Definition at line 22 of file glbopts.h.

4.147.1.17 ABIP_SOLVED

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

Definition at line 28 of file glbopts.h.

4.147.1.18 ABIP_SOLVED_INACCURATE

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

Definition at line 29 of file glbopts.h.

4.147.1.19 ABIP_UNBOUNDED

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

Definition at line 26 of file glbopts.h.

4.147.1.20 ABIP_UNBOUNDED_INACCURATE

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

Definition at line 21 of file glbopts.h.

4.147.1.21 ABIP_UNFINISHED

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

Definition at line 27 of file glbopts.h.

4.147.1.22 ABIP_UNSOLVED

```
#define ABIP_UNSOLVED (3)
```

Definition at line 30 of file glbopts.h.

4.147.1.23 ABIP_VERSION

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

Definition at line 17 of file glbopts.h.

4.147.1.24 ABS

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

Definition at line 146 of file glbopts.h.

4.147.1.25 ADAPTIVE

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

Definition at line 45 of file glbopts.h.

4.147.1.26 ADAPTIVE_LOOKBACK

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

Definition at line 48 of file glbopts.h.

4.147.1.27 ALPHA

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

Definition at line 37 of file glbopts.h.

4.147.1.28 CG_BEST_TOL

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

Definition at line 39 of file glbopts.h.

4.147.1.29 CG_MIN_TOL

```
#define CG_MIN_TOL (1e-5)
```

Definition at line 40 of file glbopts.h.

4.147.1.30 CG_RATE

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

Definition at line 38 of file glbopts.h.

4.147.1.31 CONE_TOL

```
#define CONE_TOL (1e-8)
```

Definition at line 52 of file glbopts.h.

4.147.1.32 CONVERGED_INTERVAL

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

Definition at line 181 of file glbopts.h.

4.147.1.33 DEBUG_FUNC

```
#define DEBUG_FUNC
```

Definition at line 174 of file glbopts.h.

4.147.1.34 EPS

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

Definition at line 36 of file glbopts.h.

4.147.1.35 EPS_COR

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

Definition at line 46 of file glbopts.h.

4.147.1.36 EPS_PEN

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

Definition at line 47 of file glbopts.h.

4.147.1.37 EPS_TOL

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

Definition at line 178 of file glbopts.h.

4.147.1.38 GAMMA

```
#define GAMMA (1.6)
```

Definition at line 32 of file glbopts.h.

4.147.1.39 INDETERMINATE_TOL

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

Definition at line 182 of file glbopts.h.

4.147.1.40 INFINITY

```
#define INFINITY NAN
```

Definition at line 123 of file glbopts.h.

4.147.1.41 MAX

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

Definition at line 138 of file glbopts.h.

4.147.1.42 MAX_ADMM_ITERS

```
#define MAX_ADMM_ITERS (10000000)
```

Definition at line 35 of file glbopts.h.

4.147.1.43 MAX_IPM_ITERS

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

Definition at line 34 of file glbopts.h.

4.147.1.44 MIN

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

Definition at line 142 of file glbopts.h.

4.147.1.45 NAN

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

Definition at line 120 of file glbopts.h.

4.147.1.46 NORMALIZE

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

Definition at line 41 of file glbopts.h.

4.147.1.47 POWF

```
#define POWF pow
```

Definition at line 153 of file glbopts.h.

4.147.1.48 RETURN

```
#define RETURN return
```

Definition at line 175 of file glbopts.h.

4.147.1.49 RHO_Y

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

Definition at line 44 of file glbopts.h.

4.147.1.50 SAFEDIV_POS

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

Definition at line 179 of file glbopts.h.

4.147.1.51 SCALE

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

Definition at line 42 of file glbopts.h.

4.147.1.52 SIGMA

```
#define SIGMA (0.8)
```

Definition at line 31 of file glbopts.h.

4.147.1.53 SPARSITY_RATIO

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

Definition at line 43 of file glbopts.h.

4.147.1.54 SQRTF

```
#define SQRTF sqrt
```

Definition at line 161 of file glbopts.h.

4.147.1.55 VERBOSE

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

Definition at line 49 of file glbopts.h.

4.147.1.56 WARM_START

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

Definition at line 50 of file glbopts.h.

4.147.2 Typedef Documentation

4.147.2.1 abip_float

```
typedef double abip_float
```

Definition at line 118 of file glbopts.h.

4.147.2.2 abip_int

```
typedef int abip_int
```

Definition at line 113 of file glbopts.h.

4.148 glbopts.h

Go to the documentation of this file.

```
#ifndef GLB_H_GUARD
#define GLB_H_GUARD

#ifdef __cplusplus
extern "C" {
#endif

#include <math.h>

//#define DLONG

#ifndef ABIP
#define ABIP(x) abip_##x
#endif

/* ABIP VERSION NUMBER ----------------------------------------------    */
#define ABIP_VERSION                                                          \
  ("2.0.0") /* string literals automatically null-terminated */

#define ABIP_INFEASIBLE_INACCURATE (-7)
#define ABIP_UNBOUNDED_INACCURATE (-6)
#define ABIP_SIGINT (-5)
#define ABIP_FAILED (-4)
#define ABIP_INDETERMINATE (-3)
#define ABIP_INFEASIBLE (-2)
#define ABIP_UNBOUNDED (-1)
#define ABIP_UNFINISHED (0)
#define ABIP_SOLVED (1)
#define ABIP_SOLVED_INACCURATE (2)
#define ABIP_UNSOLVED (3)
#define SIGMA (0.8)
#define GAMMA (1.6)

#define MAX_IPM_ITERS (500)
#define MAX_ADMM_ITERS (10000000)
#define EPS (1E-3)
#define ALPHA (1.8)
#define CG_RATE (2.0)
#define CG_BEST_TOL (1e-9)
#define CG_MIN_TOL (1e-5)
#define NORMALIZE (1)
#define SCALE (1.0)
#define SPARSITY_RATIO (0.01)
#define RHO_Y (1E-3)
#define ADAPTIVE (1)
#define EPS_COR (0.2)
#define EPS_PEN (0.1)
#define ADAPTIVE_LOOKBACK (20)
#define VERBOSE (1)
#define WARM_START (0)

#define CONE_TOL (1e-8)

#ifdef MATLAB_MEX_FILE
#include "mex.h"
#define abip_printf mexPrintf
#define _abip_free mxFree
#define _abip_malloc mxMalloc
#define _abip_calloc mxCalloc
#define _abip_realloc mxRealloc
#elif defined PYTHON
#include <Python.h>
#include <stdlib.h>
#define abip_printf(...)                                                          \
{                                                                                     \
      PyGILState_STATE gilstate = PyGILState_Ensure();        \
      PySys_WriteStdout(__VA_ARGS__);                                   \
      PyGILState_Release(gilstate);                                            \
}
#define _abip_free free
#define _abip_malloc malloc
#define _abip_calloc calloc
#define _abip_realloc realloc
#else
#include <stdio.h>
#include <stdlib.h>
#define abip_printf printf
#define _abip_free free
#define _abip_malloc malloc
#define _abip_calloc calloc
#define _abip_realloc realloc
#endif
```

```
#define abip_free(x)    \
      _abip_free(x);          \
      x = ABIP_NULL
#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)

// //#ifdef DLONG
// //#ifdef _WIN64
// //typedef __int64 abip_int;
// //#else
// //typedef long abip_int;
// //#endif
// //#else
// //typedef int abip_int;
// //#endif
// typedef int abip_int;


#ifdef DLONG
/*#ifdef _WIN64
#include <stdint.h>
typedef int64_t abip_int;
#else
typedef long abip_int;
#endif
*/
typedef long long abip_int;
#else
typedef int abip_int;
#endif


#ifndef SFLOAT
typedef double abip_float;
#ifndef NAN
#define NAN ((abip_float)0x7ff8000000000000)
#endif
#ifndef INFINITY
#define INFINITY NAN
#endif
#else
typedef float abip_float;
#ifndef NAN
#define NAN ((float)0x7fc00000)
#endif
#ifndef INFINITY
#define INFINITY NAN
#endif
#endif

#define ABIP_NULL 0

#ifndef MAX
#define MAX(a, b) (((a) > (b)) ? (a) : (b))
#endif

#ifndef MIN
#define MIN(a, b) (((a) < (b)) ? (a) : (b))
#endif

#ifndef ABS
#define ABS(x) (((x) < 0) ? -(x) : (x))
#endif

#ifndef POWF
#ifdef SFLOAT
#define POWF powf
#else
#define POWF pow
#endif
#endif

#ifndef SQRTF
#ifdef SFLOAT
#define SQRTF sqrtf
#else
#define SQRTF sqrt
#endif
#endif

#if EXTRA_VERBOSE > 1
#if (defined _WIN32 || defined _WIN64 || defined _WINDLL)
#define __func__ __FUNCTION__
#endif
#define DEBUG_FUNC abip_printf("IN function: %s, time: %4f ms, file: %s, line: %i\n", __func__,
```

```
 ABIP(tocq)(&global_timer), __FILE__, __LINE__);
#define RETURN
        abip_printf("EXIT function: %s, time: %4f ms, file: %s, line: %i\n", __func__,
 ABIP(tocq)(&global_timer), __FILE__, __LINE__);    \
        return
#else
#define DEBUG_FUNC
#define RETURN return
#endif

#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)

#ifdef __cplusplus
}
#endif
#endif
```

4.149 include/lasso_config.h File Reference

```
#include "glbopts.h"
#include "abip.h"
#include "amatrix.h"
#include "linsys.h"
```

Classes

- struct Lasso

Typedefs

- typedef struct Lasso lasso

Functions

- abip_int init_lasso (lasso **gen_lasso, ABIPData *d, ABIPSettings *stgs)

  Initialize the lasso problem structure.
- void scaling_lasso_data (lasso *self, ABIPCone *k)

  Customized scaling procedure for the lasso problem.
- void un_scaling_lasso_sol (lasso *self, ABIPSolution *sol)

  Get the unscaled solution of the original lasso problem.
- void calc_lasso_residuals (lasso *self, ABIPWork *w, ABIPResiduals *r, abip_int ipm_iter, abip_int admm←_iter)

  Calculate the residuals of the lasso qcp problem.
- abip_int init_lasso_linsys_work (lasso *self)

  Initialize the linear system solver work space for the lasso problem.
- abip_int solve_lasso_linsys (lasso *self, abip_float *b, abip_float *pcg_warm_start, abip_int iter, abip_float error_ratio)

  Customized linear system solver for the lasso problem.
- void free_lasso_linsys_work (lasso *self)

  Free the linear system solver work space for the lasso problem.
- void lasso_A_times (lasso *self, const abip_float *x, abip_float *y)

  Customized matrix-vector multiplication for the lasso problem with A untransposed.
- void lasso_AT_times (lasso *self, const abip_float *x, abip_float *y)

  Customized matrix-vector multiplication for the lasso problem with A transposed.
- abip_float lasso_inner_conv_check (lasso *self, ABIPWork *w)

  Check whether the inner loop of the lasso problem has converged.

4.149.1 Typedef Documentation

4.149.1.1 lasso

```
typedef struct Lasso lasso
```

Definition at line 12 of file lasso_config.h.

4.149.2 Function Documentation

4.149.2.1 calc_lasso_residuals()

```
void calc_lasso_residuals (
            lasso * self,
            ABIPWork * w,
            ABIPResiduals * r,
            abip_int ipm_iter,
            abip_int admm_iter )
```

Calculate the residuals of the lasso qcp problem.

Definition at line 367 of file lasso_config.c.

4.149.2.2 free_lasso_linsys_work()

```
void free_lasso_linsys_work (
            lasso * self )
```

Free the linear system solver work space for the lasso problem.

Definition at line 722 of file lasso_config.c.

4.149.2.3 init_lasso()

```
abip_int init_lasso (
            lasso ** gen_lasso,
            ABIPData * d,
            ABIPSettings * stgs )
```

Initialize the lasso problem structure.

Definition at line 8 of file lasso_config.c.

4.149.2.4 init_lasso_linsys_work()

```
abip_int init_lasso_linsys_work (
            lasso * self )
```

Initialize the linear system solver work space for the lasso problem.

Definition at line 624 of file lasso_config.c.

4.149.2.5 lasso_A_times()

```
void lasso_A_times (
            lasso * self,
            const abip_float * x,
            abip_float * y )
```

Customized matrix-vector multiplication for the lasso problem with A untransposed.

Definition at line 99 of file lasso_config.c.

4.149.2.6 lasso_AT_times()

```
void lasso_AT_times (
            lasso * self,
            const abip_float * x,
            abip_float * y )
```

Customized matrix-vector multiplication for the lasso problem with A transposed.

Definition at line 116 of file lasso_config.c.

4.149.2.7 lasso_inner_conv_check()

```
abip_float lasso_inner_conv_check (
            lasso * self,
            ABIPWork * w )
```

Check whether the inner loop of the lasso problem has converged.

Definition at line 323 of file lasso_config.c.

4.149.2.8 scaling_lasso_data()

```
void scaling_lasso_data (
            lasso * self,
            ABIPCone * k )
```

Customized scaling procedure for the lasso problem.

Definition at line 131 of file lasso_config.c.

4.149.2.9 solve_lasso_linsys()

```
abip_int solve_lasso_linsys (
            lasso * self,
            abip_float * b,
            abip_float * pcg_warm_start,
            abip_int iter,
            abip_float error_ratio )
```

Customized linear system solver for the lasso problem.

Definition at line 648 of file lasso_config.c.

4.149.2.10 un_scaling_lasso_sol()

```
void un_scaling_lasso_sol (
            lasso * self,
            ABIPSolution * sol )
```

Get the unscaled solution of the original lasso problem.

Definition at line 303 of file lasso_config.c.

4.150 lasso_config.h

Go to the documentation of this file.

```
#ifndef LASSO_CONFIG_H_GUARD
#define LASSO_CONFIG_H_GUARD

#ifdef __cplusplus
extern "C" {
#endif

#include "glbopts.h"
#include "abip.h"
#include "amatrix.h"
#include "linsys.h"
typedef struct Lasso lasso;

struct Lasso{

    /*----------common parts-------------*/
    enum problem_type pro_type;
    abip_int m;  //rows of input data A
```

```
    abip_int n;  //cols of input data A
    abip_int p;  //rows of ABIP constraint matrix A
    abip_int q;  //cols of ABIP constraint matrix A
    ABIPLinSysWork *L;
    ABIPSettings *stgs;
    ABIPData *data; //original data
    abip_float sparsity;

    abip_float *rho_dr; // non-identity DR scaling

    /* scaled data */
    ABIPMatrix *A;
    ABIPMatrix *Q;
    abip_float *b;
    abip_float *c;
    /*-------------*/

    void (*scaling_data)(lasso *self, ABIPCone *k);
    void (*un_scaling_sol)(lasso *self, ABIPSolution *sol);
    void (*calc_residuals)(lasso *self, ABIPWork *w, ABIPResiduals *r, abip_int ipm_iter, abip_int adm_iter);
    abip_int (*init_spe_linsys_work)(lasso *self);
    abip_int (*solve_spe_linsys)(lasso *self, abip_float *b, abip_float *pcg_warm_start, abip_int iter, abip_float error_ratio);
    void (*free_spe_linsys_work)(lasso *self);
    void (*spe_A_times)(lasso *self, const abip_float *x, abip_float *y);//y += Ax, where A is the reformulated constraint matrix of ABIP
    void (*spe_AT_times)(lasso *self, const abip_float *x, abip_float *y);//y += A'x, where A is the reformulated constraint matrix of ABIP
    abip_float (*inner_conv_check)(lasso *self, ABIPWork *w);
    /*--------------------------------*/

    abip_float lambda;
    /*-----scaling-------*/
    abip_float *D_hat;
    abip_float *D;
    abip_float *E;
    abip_float sc_b;
    abip_float sc_c;
    abip_float sc;
    abip_float sc_cone1;
    abip_float sc_cone2;
    /*-------------*/

};

abip_int init_lasso(lasso **gen_lasso, ABIPData *d, ABIPSettings *stgs);

void scaling_lasso_data(lasso *self, ABIPCone *k);
void un_scaling_lasso_sol(lasso *self, ABIPSolution *sol);
void calc_lasso_residuals(lasso *self, ABIPWork *w, ABIPResiduals *r, abip_int ipm_iter, abip_int admm_iter);
abip_int init_lasso_linsys_work(lasso *self);
abip_int solve_lasso_linsys(lasso *self, abip_float *b, abip_float *pcg_warm_start, abip_int iter, abip_float error_ratio);
void free_lasso_linsys_work(lasso *self);
void lasso_A_times(lasso *self, const abip_float *x, abip_float *y);//y += Ax, where A is the reformulated constraint matrix of ABIP
void lasso_AT_times(lasso *self, const abip_float *x, abip_float *y);//y += A'x, where A is the reformulated constraint matrix of ABIP
abip_float lasso_inner_conv_check(lasso *self, ABIPWork *w);

#ifdef __cplusplus
}
#endif
#endif
```

4.151 include/linalg.h File Reference

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

Macros

- #define RowMajor 0
- #define ColMajor 1

Functions

- void ABIP() c_dot (abip_float ∗x, const abip_float ∗y, const abip_int len)

  Elementwise multiplication of two vectors.
- abip_float ABIP() vec_mean (abip_float ∗x, abip_int len)

  Calculate the mean of a vector.
- void ABIP() set_as_scaled_array (abip_float ∗x, const abip_float ∗a, const abip_float b, abip_int len)

  Elementwise multiplication of a vector by a scalar.
- void ABIP() set_as_sqrt (abip_float ∗x, const abip_float ∗v, abip_int len)

  Elementwise square root of a vector.
- void ABIP() set_as_sq (abip_float ∗x, const abip_float ∗v, abip_int len)

  Elementwise square of a vector.
- void ABIP() scale_array (abip_float ∗a, const abip_float b, abip_int len)

  Elementwise multiplication of a vector by a scalar with replacement.
- abip_float ABIP() dot (const abip_float ∗x, const abip_float ∗y, abip_int len)

  Dot product of two vectors.
- abip_float ABIP() norm_sq (const abip_float ∗v, abip_int len)

  L2 norm of a vector.
- abip_float ABIP() norm_1 (const abip_float ∗x, const abip_int len)

  L1 norm of a vector.
- abip_float ABIP() cone_norm_1 (const abip_float ∗x, const abip_int len)

  The absolute value of the largest component of x.
- abip_float ABIP() norm (const abip_float ∗v, abip_int len)

  Square of L2 norm of a vector.
- abip_float ABIP() norm_inf (const abip_float ∗a, abip_int len)

  Calculate the maximum absolute value of a vector.
- void ABIP() add_array (abip_float ∗a, const abip_float b, abip_int len)

  Elementwise addition of two vectors with coefficients.
- void ABIP() add_scaled_array (abip_float ∗a, const abip_float ∗b, abip_int n, const abip_float sc)

  Elementwise addition of two vectors with coefficients 1.
- abip_float ABIP() norm_diff (const abip_float ∗a, const abip_float ∗b, abip_int len)

  L2 norm of the difference of two vectors.
- abip_float ABIP() norm_inf_diff (const abip_float ∗a, const abip_float ∗b, abip_int len)

  Maximum of the difference of two vectors.
- abip_float ∗ABIP() csc_to_dense (const cs ∗in_csc, const abip_int out_format)

  Convert a CSC matrix to a dense matrix.

4.151.1 Macro Definition Documentation

4.151.1.1 ColMajor

```
#define ColMajor 1
```

Definition at line 12 of file linalg.h.

4.151.1.2 RowMajor

```
#define RowMajor 0
```

Definition at line 11 of file linalg.h.

4.151.2 Function Documentation

4.151.2.1 add_array()

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

Elementwise addition of two vectors with coefficients.

Definition at line 181 of file linalg.c.

4.151.2.2 add_scaled_array()

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

Elementwise addition of two vectors with coefficients 1.

Definition at line 192 of file linalg.c.

4.151.2.3 c_dot()

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

Elementwise multiplication of two vectors.

Definition at line 9 of file linalg.c.

4.151.2.4 cone_norm_1()

```
abip_float ABIP() cone_norm_1 (
            const abip_float * x,
            const abip_int len )
```

The absolute value of the largest component of x.

Definition at line 144 of file linalg.c.

4.151.2.5 csc_to_dense()

```
abip_float *ABIP() csc_to_dense (
            const cs * in_csc,
            const abip_int out_format )
```

Convert a CSC matrix to a dense matrix.

Definition at line 247 of file linalg.c.

4.151.2.6 dot()

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

Dot product of two vectors.

Definition at line 85 of file linalg.c.

4.151.2.7 norm()

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

Square of L2 norm of a vector.

Definition at line 119 of file linalg.c.

4.151.2.8 norm_1()

```
abip_float ABIP() norm_1 (
            const abip_float * x,
            const abip_int len )
```

L1 norm of a vector.

Definition at line 130 of file linalg.c.

4.151.2.9 norm_diff()

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

L2 norm of the difference of two vectors.

Definition at line 207 of file linalg.c.

4.151.2.10 norm_inf()

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

Calculate the maximum absolute value of a vector.

Definition at line 161 of file linalg.c.

4.151.2.11 norm_inf_diff()

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

Maximum of the difference of two vectors.

Definition at line 223 of file linalg.c.

4.151.2.12 norm_sq()

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

L2 norm of a vector.

Definition at line 102 of file linalg.c.

4.151.2.13 scale_array()

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

Elementwise multiplication of a vector by a scalar with replacement.

Definition at line 71 of file linalg.c.

4.151.2.14 set_as_scaled_array()

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

Elementwise multiplication of a vector by a scalar.

Definition at line 37 of file linalg.c.

4.151.2.15 set_as_sq()

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

Elementwise square of a vector.

Definition at line 60 of file linalg.c.

4.151.2.16 set_as_sqrt()

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

Elementwise square root of a vector.

Definition at line 49 of file linalg.c.

4.151.2.17 vec_mean()

```
abip_float ABIP() vec_mean (
            abip_float * x,
            abip_int len )
```

Calculate the mean of a vector.

Definition at line 19 of file linalg.c.

4.152 linalg.h

Go to the documentation of this file.

```
#ifndef LINALG_H_GUARD
#define LINALG_H_GUARD

#ifdef __cplusplus
extern "C" {
#endif

#include "abip.h"
#include <math.h>
#include "cs.h"
#define RowMajor 0
#define ColMajor 1

void ABIP(c_dot)
(
      abip_float *x,
      const abip_float *y,
      const abip_int len
);

abip_float ABIP(vec_mean)
(
      abip_float *x,
      abip_int len
);

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

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

void ABIP(set_as_sq)
```

```
(
    abip_float *x,
    const abip_float *v,
    abip_int len
);

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


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

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

abip_float ABIP(norm_1)
(
    const abip_float *x,
    const abip_int len
);

abip_float ABIP(cone_norm_1)
(
    const abip_float *x,
     const abip_int len
);

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

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

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

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

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

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

abip_float * ABIP(csc_to_dense)(const cs * in_csc, const abip_int out_format);

#ifdef __cplusplus
}
#endif
#endif
```

4.153 include/linsys.h File Reference

```
#include <math.h>
#include "abip.h"
#include "amatrix.h"
#include "linalg.h"
#include "glbopts.h"
#include "cs.h"
#include "mkl.h"
#include "mkl_dss.h"
#include "mkl_pardiso.h"
#include "mkl_types.h"
#include "mkl_lapacke.h"
#include "util.h"
#include "qdldl.h"
```

Classes

- struct ABIP_LIN_SYS_WORK

Functions

- void ABIP() accum_by_Atrans (const ABIPMatrix ∗A, const abip_float ∗x, abip_float ∗y)

    Add the transposed matrix-vector product to a vector.
- void ABIP() accum_by_A (const ABIPMatrix ∗A, const abip_float ∗x, abip_float ∗y)

    Add the matrix-vector product to a vector.
- abip_int ABIP() validate_lin_sys (const ABIPMatrix ∗A)

    Check the validity of the linear system.
- char ∗ABIP() get_lin_sys_method (spe_problem ∗spe)

    Get the method used to solve the linear system.
- char ∗ABIP() get_lin_sys_summary (spe_problem ∗self, ABIPInfo ∗info)

    Get the summary infomation of the linear system.
- void ABIP() free_A_matrix (ABIPMatrix ∗A)

    Free the memory of a matrix.
- abip_int ABIP() copy_A_matrix (ABIPMatrix ∗∗dstp, const ABIPMatrix ∗src)

    Copy a matrix.
- abip_int LDL_factor (cs ∗A, cs ∗∗L, abip_float ∗Dinv)
- abip_int ABIP() init_linsys_work (spe_problem ∗spe)

    Initialize linear system solver work space.
- abip_int ABIP() solve_linsys (spe_problem ∗spe, abip_float ∗b, abip_int n, abip_float ∗pcg_warm_start, abip_float pcg_tol)

    solve linear system according to the specific linsys solver
- abip_int ABIP() free_linsys (spe_problem ∗spe)

    free memory for linear system solver

4.153.1 Function Documentation

4.153.1.1 accum_by_A()

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

Add the matrix-vector product to a vector.

Definition at line 229 of file linsys.c.

4.153.1.2 accum_by_Atrans()

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

Add the transposed matrix-vector product to a vector.

Definition at line 191 of file linsys.c.

4.153.1.3 copy_A_matrix()

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

Copy a matrix.

Definition at line 12 of file linsys.c.

4.153.1.4 free_A_matrix()

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

Free the memory of a matrix.

Definition at line 152 of file linsys.c.

4.153.1.5 free_linsys()

```
abip_int ABIP() free_linsys (
            spe_problem * spe )
```

free memory for linear system solver

Definition at line 1181 of file linsys.c.

4.153.1.6 get_lin_sys_method()

```
char *ABIP() get_lin_sys_method (
            spe_problem * spe )
```

Get the method used to solve the linear system.

Definition at line 39 of file linsys.c.

4.153.1.7 get_lin_sys_summary()

```
char *ABIP() get_lin_sys_summary (
            spe_problem * self,
            ABIPInfo * info )
```

Get the summary infomation of the linear system.

Definition at line 71 of file linsys.c.

4.153.1.8 init_linsys_work()

```
abip_int ABIP() init_linsys_work (
            spe_problem * spe )
```

Initialize linear system solver work space.

Definition at line 1027 of file linsys.c.

4.153.1.9 LDL_factor()

```
abip_int LDL_factor (
            cs * A,
            cs ** L,
            abip_float * Dinv )
```

Definition at line 542 of file linsys.c.

4.153.1.10 solve_linsys()

```
abip_int ABIP() solve_linsys (
        spe_problem * spe,
        abip_float * b,
        abip_int n,
        abip_float * pcg_warm_start,
        abip_float pcg_tol )
```

solve linear system according to the specific linsys solver

Definition at line 1141 of file linsys.c.

4.153.1.11 validate_lin_sys()

```
abip_int ABIP() validate_lin_sys (
        const ABIPMatrix * A )
```

Check the validity of the linear system.

Definition at line 102 of file linsys.c.

4.154 linsys.h

Go to the documentation of this file.

```
#ifndef LINSYS_H_GUARD
#define LINSYS_H_GUARD

#ifdef __cplusplus
extern "C" {
#endif

#include <math.h>
#include "abip.h"
#include "amatrix.h"
#include "linalg.h"
#include "glbopts.h"
#include "cs.h"
#include "mkl.h"
#include "mkl_dss.h"
#include "mkl_pardiso.h"
#include "mkl_types.h"
#include "mkl_lapacke.h"
#include "util.h"
#include "qdldl.h"


struct ABIP_LIN_SYS_WORK
{
    abip_float total_solve_time;

    //matrix to factorize
    cs *K;

    //mkl-dss
    _MKL_DSS_HANDLE_t handle;


    //mkl-pardiso
    void *pt[64];
    MKL_INT iparm[64];
    MKL_INT maxfct, mnum, error, msglvl;
    abip_float ddum;          /* Double dummy */
    MKL_INT idum;
    MKL_INT mtype;
```

```
    //pcg
    abip_float* M;     //preconditioner for pcg
    abip_int total_cg_iters;


    //sparse cholesky
    css *S ;
    csn *N ;


    //qdldl
    cs *L;              //matrix L of LDL' factorization
    abip_float *Dinv;     //the diagonal vector of the diagonal matrix D L of LDL' factorization
    abip_int nnz_LDL;
    abip_int *P;  //permutation for KKT matrix
    abip_float *bp;


    //lapack dense cholesky
    // A = UTU
    abip_float *U;
};

/*   y += A'*x
        A in column compressed format
        parallelizes over columns (rows of A')
*/
void ABIP(accum_by_Atrans)
(
    const ABIPMatrix* A,
    const abip_float* x,
    abip_float* y
);

/*  y += A*x
    A in column compressed format
    this parallelizes over columns and uses
    pragma atomic to prevent concurrent writes to y
*/
void ABIP(accum_by_A)
(
    const ABIPMatrix* A,
    const abip_float* x,
    abip_float* y
);



abip_int ABIP(validate_lin_sys)
(
    const ABIPMatrix *A
);

char *ABIP(get_lin_sys_method)
(
    spe_problem *spe
);

char *ABIP(get_lin_sys_summary)
(
    spe_problem *self,
    ABIPInfo *info
);

void ABIP(free_A_matrix)
(
    ABIPMatrix *A
);

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

abip_int LDL_factor(cs *A, cs **L, abip_float *Dinv);

abip_int ABIP(init_linsys_work)(spe_problem *spe);


abip_int ABIP(solve_linsys)(spe_problem *spe, abip_float *b, abip_int n, abip_float *pcg_warm_start,
 abip_float pcg_tol);


```

```
abip_int ABIP(free_linsys)(spe_problem *spe);
#ifdef __cplusplus
}
#endif

#endif
```

4.155 include/qcp_config.h File Reference

```
#include "glbopts.h"
#include "abip.h"
#include "amatrix.h"
#include "linsys.h"
```

Classes

- struct qcp

Typedefs

- typedef struct qcp qcp

Functions

- abip_int init_qcp (qcp **QCP, ABIPData *d, ABIPSettings *stgs)

  Initialize the qcp problem structure.
- void scaling_qcp_data (qcp *self, ABIPCone *k)

  Scale the data for the qcp problem.
- void un_scaling_qcp_sol (qcp *self, ABIPSolution *sol)

  Get the unscaled solution of the general qcp problem.
- void calc_qcp_residuals (qcp *self, ABIPWork *w, ABIPResiduals *r, abip_int ipm_iter, abip_int admm_iter)

  Calculate the residuals of the general qcp problem.
- abip_int init_qcp_linsys_work (qcp *self)

  Initialize the linear system solver work space for the general qcp problem.
- abip_int solve_qcp_linsys (qcp *self, abip_float *b, abip_float *pcg_warm_start, abip_int iter, abip_float error_ratio)

  Linear system solver for the general qcp problem.
- void free_qcp_linsys_work (qcp *self)

  Free the linear system solver work space for the general qcp problem.
- void qcp_A_times (qcp *self, const abip_float *x, abip_float *y)

  Matrix-vector multiplication for the general qcp problem with A untransposed.
- void qcp_AT_times (qcp *self, const abip_float *x, abip_float *y)

  Matrix-vector multiplication for the general qcp problem with A transposed.
- abip_float qcp_inner_conv_check (qcp *self, ABIPWork *w)

  Check whether the inner loop of the genral qcp problem has converged.

4.155.1 Typedef Documentation

4.155.1.1 qcp

```
typedef struct qcp qcp
```

Definition at line 12 of file qcp_config.h.

4.155.2 Function Documentation

4.155.2.1 calc_qcp_residuals()

```
void calc_qcp_residuals (
            qcp * self,
            ABIPWork * w,
            ABIPResiduals * r,
            abip_int ipm_iter,
            abip_int admm_iter )
```

Calculate the residuals of the general qcp problem.

Definition at line 562 of file qcp_config.c.

4.155.2.2 free_qcp_linsys_work()

```
void free_qcp_linsys_work (
            qcp * self )
```

Free the linear system solver work space for the general qcp problem.

Definition at line 886 of file qcp_config.c.

4.155.2.3 init_qcp()

```
abip_int init_qcp (
            qcp ** QCP,
            ABIPData * d,
            ABIPSettings * stgs )
```

Initialize the qcp problem structure.

Definition at line 8 of file qcp_config.c.

4.155.2.4 init_qcp_linsys_work()

```
abip_int init_qcp_linsys_work (
            qcp * self )
```

Initialize the linear system solver work space for the general qcp problem.

Definition at line 799 of file qcp_config.c.

4.155.2.5 qcp_A_times()

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

Matrix-vector multiplication for the general qcp problem with A untransposed.

Definition at line 72 of file qcp_config.c.

4.155.2.6 qcp_AT_times()

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

Matrix-vector multiplication for the general qcp problem with A transposed.

Definition at line 82 of file qcp_config.c.

4.155.2.7 qcp_inner_conv_check()

```
abip_float qcp_inner_conv_check (
            qcp * self,
            ABIPWork * w )
```

Check whether the inner loop of the genral qcp problem has converged.

Definition at line 518 of file qcp_config.c.

4.155.2.8 scaling_qcp_data()

```
void scaling_qcp_data (
            qcp * self,
            ABIPCone * k )
```

Scale the data for the qcp problem.

Definition at line 91 of file qcp_config.c.

4.155.2.9 solve_qcp_linsys()

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

Linear system solver for the general qcp problem.

Definition at line 826 of file qcp_config.c.

4.155.2.10 un_scaling_qcp_sol()

```
void un_scaling_qcp_sol (
            qcp * self,
            ABIPSolution * sol )
```

Get the unscaled solution of the general qcp problem.

Definition at line 496 of file qcp_config.c.

4.156 qcp_config.h

Go to the documentation of this file.

```
#ifndef QCP_CONFIG_H_GUARD
#define QCP_CONFIG_H_GUARD

#ifdef __cplusplus
extern "C" {
#endif

#include "glbopts.h"
#include "abip.h"
#include "amatrix.h"
#include "linsys.h"
typedef struct qcp qcp;

struct qcp{

    /*----------common parts-------------*/
    enum problem_type pro_type;
    abip_int m;  //rows of input data A
```

```
    abip_int n;  //cols of input data A
    abip_int p;  //rows of ABIP constraint matrix A
    abip_int q;  //cols of ABIP constraint matrix A
    ABIPLinSysWork *L;
    ABIPSettings *stgs;
    ABIPData *data; //original data
    abip_float sparsity;

    abip_float *rho_dr; // non-identity DR scaling

    /* scaled data */
    ABIPMatrix *A;
    ABIPMatrix *Q;
    abip_float *b;
    abip_float *c;
    /*-------------*/

    void (*scaling_data)(qcp *self, ABIPCone *k);
    void (*un_scaling_sol)(qcp *self, ABIPSolution *sol);
    void (*calc_residuals)(qcp *self, ABIPWork *w, ABIPResiduals *r, abip_int ipm_iter, abip_int admm_iter);
    abip_int (*init_spe_linsys_work)(qcp *self);
    abip_int (*solve_spe_linsys)(qcp *self, abip_float *b, abip_float *pcg_warm_start, abip_int iter, abip_float error_ratio);
    void (*free_spe_linsys_work)(qcp *self);
    void (*spe_A_times)(qcp *self, const abip_float *x, abip_float *y);//y += Ax, where A is the reformulated constraint matrix of ABIP
    void (*spe_AT_times)(qcp *self, const abip_float *x, abip_float *y);//y += A'x, where A is the reformulated constraint matrix of ABIP
    abip_float (*inner_conv_check)(qcp *self, ABIPWork *w);
    /*--------------------------------*/

    /*-----scaling-------*/
    abip_float *D;
    abip_float *E;
    abip_float sc_b;
    abip_float sc_c;
    /*-------------*/

};

abip_int init_qcp(qcp **QCP, ABIPData *d, ABIPSettings *stgs);

void scaling_qcp_data(qcp *self, ABIPCone *k);
void un_scaling_qcp_sol(qcp *self, ABIPSolution *sol);
void calc_qcp_residuals(qcp *self, ABIPWork *w, ABIPResiduals *r, abip_int ipm_iter, abip_int admm_iter);
abip_int init_qcp_linsys_work(qcp *self);
abip_int solve_qcp_linsys(qcp *self, abip_float *b, abip_float *pcg_warm_start, abip_int iter, abip_float error_ratio);
void free_qcp_linsys_work(qcp *self);
void qcp_A_times(qcp *self, const abip_float *x, abip_float *y);//y += Ax, where A is the reformulated constraint matrix of ABIP
void qcp_AT_times(qcp *self, const abip_float *x, abip_float *y);//y += A'x, where A is the reformulated constraint matrix of ABIP
abip_float qcp_inner_conv_check(qcp *self, ABIPWork *w);
#ifdef __cplusplus
}
#endif
#endif
```

4.157 include/svm_config.h File Reference

```
#include "glbopts.h"
#include "abip.h"
#include "amatrix.h"
#include "linsys.h"
```

Classes

- struct Svm

Typedefs

- typedef struct Svm svm

Functions

- abip_int init_svm (svm **gen_svm, ABIPData *d, ABIPSettings *stgs)

  Initialize the svm socp formulation structure.
- void scaling_svm_data (svm *self, ABIPCone *k)

  Customized scaling procedure for the svm socp formulation.
- void un_scaling_svm_sol (svm *self, ABIPSolution *sol)

  Get the unscaled solution of the original svm problem.
- void calc_svm_residuals (svm *self, ABIPWork *w, ABIPResiduals *r, abip_int ipm_iter, abip_int admm_iter)

  Calculate the residuals of the svm socp formulation.
- abip_int init_svm_linsys_work (svm *self)

  Initialize the linear system solver work space for the svm socp formulation.
- abip_int solve_svm_linsys (svm *self, abip_float *b, abip_float *pcg_warm_start, abip_int iter, abip_float error_ratio)

  Customized linear system solver for the svm socp formulation.
- void free_svm_linsys_work (svm *self)

  Free the linear system solver work space for the svm socp formulation.
- void svm_A_times (svm *self, const abip_float *x, abip_float *y)

  Customized matrix-vector multiplication for the svm socp formulation with A untransposed.
- void svm_AT_times (svm *self, const abip_float *x, abip_float *y)

  Customized matrix-vector multiplication for the svm socp formulation with A transposed.
- abip_float svm_inner_conv_check (svm *self, ABIPWork *w)

  Check whether the inner loop of the svm socp formulation has converged.

4.157.1 Typedef Documentation

4.157.1.1 svm

```
typedef struct Svm svm
```

Definition at line 12 of file svm_config.h.

4.157.2 Function Documentation

4.157.2.1 calc_svm_residuals()

```
void calc_svm_residuals (
            svm * self,
            ABIPWork * w,
            ABIPResiduals * r,
            abip_int ipm_iter,
            abip_int admm_iter )
```

Calculate the residuals of the svm socp formulation.

Definition at line 445 of file svm_config.c.

4.157.2.2 free_svm_linsys_work()

```
void free_svm_linsys_work (
            svm * self )
```

Free the linear system solver work space for the svm socp formulation.

Definition at line 812 of file svm_config.c.

4.157.2.3 init_svm()

```
abip_int init_svm (
            svm ** gen_svm,
            ABIPData * d,
            ABIPSettings * stgs )
```

Initialize the svm socp formulation structure.

Definition at line 8 of file svm_config.c.

4.157.2.4 init_svm_linsys_work()

```
abip_int init_svm_linsys_work (
            svm * self )
```

Initialize the linear system solver work space for the svm socp formulation.

Definition at line 702 of file svm_config.c.

4.157.2.5 scaling_svm_data()

```
void scaling_svm_data (
            svm * self,
            ABIPCone * k )
```

Customized scaling procedure for the svm socp formulation.

Definition at line 283 of file svm_config.c.

4.157.2.6 solve_svm_linsys()

```
abip_int solve_svm_linsys (
            svm * self,
            abip_float * b,
            abip_float * pcg_warm_start,
            abip_int iter,
            abip_float error_ratio )
```

Customized linear system solver for the svm socp formulation.

Definition at line 728 of file svm_config.c.

4.157.2.7 svm_A_times()

```
void svm_A_times (
            svm * self,
            const abip_float * x,
            abip_float * y )
```

Customized matrix-vector multiplication for the svm socp formulation with A untransposed.

Definition at line 175 of file svm_config.c.

4.157.2.8 svm_AT_times()

```
void svm_AT_times (
            svm * self,
            const abip_float * x,
            abip_float * y )
```

Customized matrix-vector multiplication for the svm socp formulation with A transposed.

Definition at line 202 of file svm_config.c.

4.157.2.9 svm_inner_conv_check()

```
abip_float svm_inner_conv_check (
            svm * self,
            ABIPWork * w )
```

Check whether the inner loop of the svm socp formulation has converged.

Definition at line 234 of file svm_config.c.

4.157.2.10 un_scaling_svm_sol()

```
void un_scaling_svm_sol (
            svm * self,
            ABIPSolution * sol )
```

Get the unscaled solution of the original svm problem.

Definition at line 413 of file svm_config.c.

4.158 svm_config.h

Go to the documentation of this file.

```
#ifndef SVM_CONFIG_H_GUARD
#define SVM_CONFIG_H_GUARD

#ifdef __cplusplus
extern "C" {
#endif

#include "glbopts.h"
#include "abip.h"
#include "amatrix.h"
#include "linsys.h"
typedef struct Svm svm;

struct Svm{

    /*----------common parts-------------*/
    enum problem_type pro_type;
    abip_int m;  //rows of input data A
    abip_int n;  //cols of input data A
    abip_int p;  //rows of ABIP constraint matrix A
    abip_int q;  //cols of ABIP constraint matrix A
    ABIPLinSysWork *L;
    ABIPSettings *stgs;
    ABIPData *data; //original data
    abip_float sparsity;

    abip_float *rho_dr; // non-identity DR scaling

    /* scaled data */
    ABIPMatrix *A;
    ABIPMatrix *Q;
    abip_float *b;
    abip_float *c;
    /*-------------*/

    void (*scaling_data)(svm *self, ABIPCone *k);
    void (*un_scaling_sol)(svm *self, ABIPSolution *sol);
    void (*calc_residuals)(svm *self, ABIPWork *w, ABIPResiduals *r, abip_int ipm_iter, abip_int
  admm_iter);
    abip_int (*init_spe_linsys_work)(svm *self);
    abip_int (*solve_spe_linsys)(svm *self, abip_float *b, abip_float *pcg_warm_start, abip_int iter,
  abip_float error_ratio);
    void (*free_spe_linsys_work)(svm *self);
```

```
    void (*spe_A_times)(svm *self, const abip_float *x, abip_float *y);//y += Ax, where A is the reformulated constraint matrix of ABIP
    void (*spe_AT_times)(svm *self, const abip_float *x, abip_float *y);//y += A'x, where A is the reformulated constraint matrix of ABIP
    abip_float (*inner_conv_check)(svm *self, ABIPWork *w);
    /*--------------------------------*/

    abip_float lambda; // 'C' in svm
    /*-----scaling-------*/
    abip_float *sc_D;
    abip_float *sc_E;
    abip_float *sc_F;
    abip_float sc_b;
    abip_float sc_c;
    abip_float sc;
    abip_float sc_cone1;
    abip_float sc_cone2;
    /*-------------*/
    ABIPMatrix *wA;
    abip_float *wy;
    abip_float *wB;
    abip_float *wC;
    abip_float *wD;
    abip_float *wE;
    abip_float *wF;
    abip_float *wG;
    abip_float *wH;
    ABIPMatrix *wX;
};

abip_int init_svm(svm **gen_svm, ABIPData *d, ABIPSettings *stgs);

void scaling_svm_data(svm *self, ABIPCone *k);
void un_scaling_svm_sol(svm *self, ABIPSolution *sol);
void calc_svm_residuals(svm *self, ABIPWork *w, ABIPResiduals *r, abip_int ipm_iter, abip_int adm_iter);
abip_int init_svm_linsys_work(svm *self);
abip_int solve_svm_linsys(svm *self, abip_float *b, abip_float *pcg_warm_start, abip_int iter, abip_float error_ratio);
void free_svm_linsys_work(svm *self);
void svm_A_times(svm *self, const abip_float *x, abip_float *y);//y += Ax, where A is the reformulated constraint matrix of ABIP
void svm_AT_times(svm *self, const abip_float *x, abip_float *y);//y += A'x, where A is the reformulated constraint matrix of ABIP
abip_float svm_inner_conv_check(svm *self, ABIPWork *w);

#ifdef __cplusplus
}
#endif
#endif
```

4.159 include/svm_qp_config.h File Reference

```
#include "glbopts.h"
#include "abip.h"
#include "amatrix.h"
#include "linsys.h"
```

Classes

- struct SVMqp

Typedefs

- typedef struct SVMqp svmqp

Functions

- abip_int init_svmqp (svmqp **gen_svm, ABIPData *d, ABIPSettings *stgs)

  Initialize the svm qp formulation structure.
- void scaling_svmqp_data (svmqp *self, ABIPCone *k)

  Customized scaling procedure for the svm qp formulation.
- void un_scaling_svmqp_sol (svmqp *self, ABIPSolution *sol)

  Get the unscaled solution of the original svm problem.
- void calc_svmqp_residuals (svmqp *self, ABIPWork *w, ABIPResiduals *r, abip_int ipm_iter, abip_int admm_iter)

  Calculate the residuals of the svm qp formulation.
- abip_int init_svmqp_linsys_work (svmqp *self)

  Initialize the linear system solver work space for the svm qp formulation.
- abip_int solve_svmqp_linsys (svmqp *self, abip_float *b, abip_float *pcg_warm_start, abip_int iter, abip_float error_ratio)

  Customized linear system solver for the svm qp formulation.
- void free_svmqp_linsys_work (svmqp *self)

  Free the linear system solver work space for the svm qp formulation.
- void svmqp_A_times (svmqp *self, const abip_float *x, abip_float *y)

  Customized matrix-vector multiplication for the svm qp formulation with A untransposed.
- void svmqp_AT_times (svmqp *self, const abip_float *x, abip_float *y)

  Customized matrix-vector multiplication for the svm qp formulation with A transposed.
- abip_float svmqp_inner_conv_check (svmqp *self, ABIPWork *w)

  Check whether the inner loop of the svm qp formulation has converged.

4.159.1 Typedef Documentation

4.159.1.1 svmqp

```
typedef struct SVMqp svmqp
```

Definition at line 12 of file svm_qp_config.h.

4.159.2 Function Documentation

4.159.2.1 calc_svmqp_residuals()

```
void calc_svmqp_residuals (
            svmqp * self,
            ABIPWork * w,
            ABIPResiduals * r,
            abip_int ipm_iter,
            abip_int admm_iter )
```

Calculate the residuals of the svm qp formulation.

Definition at line 628 of file svm_qp_config.c.

4.159.2.2 free_svmqp_linsys_work()

```
void free_svmqp_linsys_work (
            svmqp * self )
```

Free the linear system solver work space for the svm qp formulation.

Definition at line 1002 of file svm_qp_config.c.

4.159.2.3 init_svmqp()

```
abip_int init_svmqp (
            svmqp ** gen_svm,
            ABIPData * d,
            ABIPSettings * stgs )
```

Initialize the svm qp formulation structure.

Definition at line 8 of file svm_qp_config.c.

4.159.2.4 init_svmqp_linsys_work()

```
abip_int init_svmqp_linsys_work (
            svmqp * self )
```

Initialize the linear system solver work space for the svm qp formulation.

Definition at line 868 of file svm_qp_config.c.

4.159.2.5 scaling_svmqp_data()

```
void scaling_svmqp_data (
            svmqp * self,
            ABIPCone * k )
```

Customized scaling procedure for the svm qp formulation.

Definition at line 196 of file svm_qp_config.c.

4.159.2.6 solve_svmqp_linsys()

```
abip_int solve_svmqp_linsys (
            svmqp * self,
            abip_float * b,
            abip_float * pcg_warm_start,
            abip_int iter,
            abip_float error_ratio )
```

Customized linear system solver for the svm qp formulation.

Definition at line 894 of file svm_qp_config.c.

4.159.2.7 svmqp_A_times()

```
void svmqp_A_times (
            svmqp * self,
            const abip_float * x,
            abip_float * y )
```

Customized matrix-vector multiplication for the svm qp formulation with A untransposed.

Definition at line 128 of file svm_qp_config.c.

4.159.2.8 svmqp_AT_times()

```
void svmqp_AT_times (
            svmqp * self,
            const abip_float * x,
            abip_float * y )
```

Customized matrix-vector multiplication for the svm qp formulation with A transposed.

Definition at line 141 of file svm_qp_config.c.

4.159.2.9 svmqp_inner_conv_check()

```
abip_float svmqp_inner_conv_check (
            svmqp * self,
            ABIPWork * w )
```

Check whether the inner loop of the svm qp formulation has converged.

Definition at line 152 of file svm_qp_config.c.

4.159.2.10 un_scaling_svmqp_sol()

```
void un_scaling_svmqp_sol (
            svmqp * self,
            ABIPSolution * sol )
```

Get the unscaled solution of the original svm problem.

Definition at line 597 of file svm_qp_config.c.

4.160 svm_qp_config.h

Go to the documentation of this file.

```
#ifndef SVM_QP_CONFIG_H_GUARD
#define SVM_QP_CONFIG_H_GUARD

#ifdef __cplusplus
extern "C" {
#endif

#include "glbopts.h"
#include "abip.h"
#include "amatrix.h"
#include "linsys.h"
typedef struct SVMqp svmqp;

struct SVMqp{

    /*----------common parts-------------*/
    enum problem_type pro_type;
    abip_int m;  //rows of input data A
    abip_int n;  //cols of input data A
    abip_int p;  //rows of ABIP constraint matrix A
    abip_int q;  //cols of ABIP constraint matrix A
    ABIPLinSysWork *L;
    ABIPSettings *stgs;
    ABIPData *data; //original data
    abip_float sparsity;

    abip_float *rho_dr; // non-identity DR scaling

    /* scaled data */
    ABIPMatrix *A;
    ABIPMatrix *Q;
    abip_float *b;
    abip_float *c;
    /*-------------*/

    void (*scaling_data)(svmqp *self, ABIPCone *k);
    void (*un_scaling_sol)(svmqp *self, ABIPSolution *sol);
    void (*calc_residuals)(svmqp *self, ABIPWork *w, ABIPResiduals *r, abip_int ipm_iter, abip_int admm_iter);
    abip_int (*init_spe_linsys_work)(svmqp *self);
    abip_int (*solve_spe_linsys)(svmqp *self, abip_float *b, abip_float *pcg_warm_start, abip_int iter, abip_float error_ratio);
    void (*free_spe_linsys_work)(svmqp *self);
    void (*spe_A_times)(svmqp *self, const abip_float *x, abip_float *y);//y += Ax, where A is the reformulated constraint matrix of ABIP
    void (*spe_AT_times)(svmqp *self, const abip_float *x, abip_float *y);//y += A'x, where A is the reformulated constraint matrix of ABIP
    abip_float (*inner_conv_check)(svmqp *self, ABIPWork *w);
    /*--------------------------------*/

    abip_float lambda;
    /*-----scaling-------*/
    abip_float *D;
    abip_float *E;
    abip_float *F;
    abip_float *H;
    abip_float sc_b;
    abip_float sc_c;
};

abip_int init_svmqp(svmqp **gen_svm, ABIPData *d, ABIPSettings *stgs);

void scaling_svmqp_data(svmqp *self, ABIPCone *k);
```

```
void un_scaling_svmqp_sol(svmqp *self, ABIPSolution *sol);
void calc_svmqp_residuals(svmqp *self, ABIPWork *w, ABIPResiduals *r, abip_int ipm_iter, abip_int admm_iter);
abip_int init_svmqp_linsys_work(svmqp *self);
abip_int solve_svmqp_linsys(svmqp *self, abip_float *b, abip_float *pcg_warm_start, abip_int iter, abip_float error_ratio);
void free_svmqp_linsys_work(svmqp *self);
void svmqp_A_times(svmqp *self, const abip_float *x, abip_float *y);//y += Ax, where A is the reformulated constraint matrix of ABIP
void svmqp_AT_times(svmqp *self, const abip_float *x, abip_float *y);//y += A'x, where A is the reformulated constraint matrix of ABIP
abip_float svmqp_inner_conv_check(svmqp *self, ABIPWork *w);

#ifdef __cplusplus
}
#endif
#endif
```

4.161 include/util.h File Reference

```
#include "abip.h"
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
```

Functions

- struct ABIP (timer)
- void ABIP() tic (ABIP(timer) ∗t)
- abip_float ABIP() toc (ABIP(timer) ∗t)
- abip_float ABIP() str_toc (char ∗str, ABIP(timer) ∗t)
- abip_float ABIP() tocq (ABIP(timer) ∗t)
- void ABIP() print_data (const ABIPData ∗d)
- void ABIP() print_work (const ABIPWork ∗w)
- void ABIP() print_array (const abip_float ∗arr, abip_int n, const char ∗name)
- void ABIP() set_default_settings (ABIPData ∗d)

  Default parameter settings.
- void ABIP() free_info (ABIPInfo ∗info)
- void ABIP() free_sol (ABIPSolution ∗sol)
- void ABIP() free_data (ABIPData ∗d)
- void ABIP() free_cone (ABIPCone ∗k)

4.161.1 Function Documentation

4.161.1.1 ABIP()

```
ABIP (
            timer )
```

Definition at line 1 of file util.h.

4.161.1.2 free_cone()

```
void ABIP() free_cone (
            ABIPCone * k )
```

Definition at line 148 of file util.c.

4.161.1.3 free_data()

```
void ABIP() free_data (
            ABIPData * d )
```

Definition at line 160 of file util.c.

4.161.1.4 free_info()

```
void ABIP() free_info (
            ABIPInfo * info )
```

Definition at line 142 of file util.c.

4.161.1.5 free_sol()

```
void ABIP() free_sol (
            ABIPSolution * sol )
```

Definition at line 182 of file util.c.

4.161.1.6 print_array()

```
void ABIP() print_array (
            const abip_float * arr,
            abip_int n,
            const char * name )
```

Definition at line 119 of file util.c.

4.161.1.7 print_data()

```
void ABIP() print_data (
             const ABIPData * d )
```

Definition at line 100 of file util.c.

4.161.1.8 print_work()

```
void ABIP() print_work (
             const ABIPWork * w )
```

Definition at line 80 of file util.c.

4.161.1.9 set_default_settings()

```
void ABIP() set_default_settings (
             ABIPData * d )
```

Default parameter settings.

Definition at line 203 of file util.c.

4.161.1.10 str_toc()

```
abip_float ABIP() str_toc (
             char * str,
             ABIP(timer) * t )
```

Definition at line 74 of file util.c.

4.161.1.11 tic()

```
void ABIP() tic (
             ABIP(timer) * t )
```

Definition at line 48 of file util.c.

4.161.1.12 toc()

```
abip_float ABIP() toc (
            ABIP(timer) * t )
```

Definition at line 68 of file util.c.

4.161.1.13 tocq()

```
abip_float ABIP() tocq (
            ABIP(timer) * t )
```

Definition at line 50 of file util.c.

4.162 util.h

Go to the documentation of this file.

```
#ifndef UTIL_H_GUARD
#define UTIL_H_GUARD

#ifdef __cplusplus
extern "C" {
#endif

#include "abip.h"
#include <stdlib.h>
#include <stdio.h>

#if (defined NOTIMER)
typedef void *ABIP(timer);

#elif(defined _WIN32 || defined _WIN64 || defined _WINDLL)

#include <windows.h>
typedef struct ABIP(timer)
{
      LARGE_INTEGER tic;
      LARGE_INTEGER toc;
      LARGE_INTEGER freq;
} ABIP(timer);

#elif(defined __APPLE__)

#include <mach/mach_time.h>
typedef struct ABIP(timer)
{
      uint64_t tic;
      uint64_t toc;
      mach_timebase_info_data_t tinfo;
} ABIP(timer);

#else

#include <time.h>
typedef struct ABIP(timer)
{
      struct timespec tic;
      struct timespec toc;
} ABIP(timer);

#endif

#if EXTRA_VERBOSE > 1
extern ABIP(timer) global_timer;
#endif

void ABIP(tic)(ABIP(timer) *t);
abip_float ABIP(toc)(ABIP(timer) *t);
abip_float ABIP(str_toc)(char *str, ABIP(timer) *t);
```

```
abip_float ABIP(tocq)(ABIP(timer) *t);

void ABIP(print_data)(const ABIPData *d);
void ABIP(print_work)(const ABIPWork *w);
void ABIP(print_array)(const abip_float *arr, abip_int n, const char *name);
void ABIP(set_default_settings)(ABIPData *d);
void ABIP(free_info)(ABIPInfo *info);
void ABIP(free_sol)(ABIPSolution *sol);
void ABIP(free_data)(ABIPData *d);
void ABIP(free_cone)(ABIPCone *k);
#ifdef __cplusplus
}
#endif
#endif
```

4.163 make_abip_qcp.m File Reference

Functions

- else error ('Unsupported platform.\n')
- if (debug==0) debugcommand
- fprintf ("%s\n", mexcommand)
- eval (replace(mexcommand, "Program Files (x86)", "'Program Files (x86)'"))
- addpath (pmex)
- if (mex_type=="ml") fprintf("Successfull compiled mex function abip_ml\n")
- fprintf ("Usage:[sol,info] = abip_ml(data,settings)\n ")
- fprintf ("data struct contais X,y,lambda\n")
- fprintf ("data struct contais A,Q,c,rl,ru,lb,ub and L(optional)\n")
- fprintf ("data struct contais A,b,c\n")
- fprintf ("data struct contais A,Q,b,c\n")

Variables

- mex_type = 'qcp'
- debug = 0
- platform = convertCharsToStrings(computer('arch'))
- link = ' '
- MKLROOT = getenv('MKLROOT')
- mkl_lib_path = fullfile(MKLROOT, 'lib')
- lib_path = join(['-L', mkl_lib_path])
- If use MKL
- If use we suggest you set the environmental variables by oneapi
- If use we suggest you set the environmental variables by it is typically placed at opt intel oneapi setvars sh alternatively
- If use we suggest you set the environmental variables by it is typically placed at opt intel oneapi setvars sh you can set the lib_path to your MKL path by your self
- If use we suggest you set the environmental variables by it is typically placed at opt intel oneapi setvars sh you can set the lib_path to your MKL path by your For example
- If use we suggest you set the environmental variables by it is typically placed at opt intel oneapi setvars sh you can set the lib_path to your MKL path by your For in linux
- If use we suggest you set the environmental variables by it is typically placed at opt intel oneapi setvars sh you can set the lib_path to your MKL path by your For in you may find it at opt intel oneapi mkl lib intel64
- end mexfname = "abip_" + mex_type
- psrc = fullfile('.', 'source')
- pmex = fullfile('.', 'mex')
- mex_file = fullfile(pmex, ['abip_' mex_type '_mex.c'])
- src_files = dir([psrc '/*.c'])

- srclist = [ ]
- for i
- end src = fullfile(psrc, srclist)
- pcs = fullfile('.', 'csparse', 'Source')
- cs_files = dir([pcs '/∗.c'])
- cslist = [ ]
- end cs = fullfile(pcs, cslist)
- pldl = fullfile('.', 'qdldl', 'src')
- ldl_files = dir([pldl '/∗.c'])
- ldllist = [ ]
- end ldl = fullfile(pldl, ldllist)
- pamd = fullfile('.', 'amd')
- amd_files = dir([pamd '/∗.c'])
- amdlist = [ ]
- end amd = fullfile(pamd, amdlist)
- pinc = "include"
- cs_include = fullfile("csparse", "Include")
- ldl_include = fullfile("qdldl", "include")
- mkl_include = fullfile(mkl_path, "include")
- amd_include = "amd"
- inc = [pinc,mkl_include,cs_include,ldl_include, amd_include]
- else debugcommand = "-g "
- end mexcommand = "mex " + debugcommand + " -output " + join([mexfname, lib_path, src, inc, link])

4.163.1 Function Documentation

4.163.1.1 addpath()

```
addpath (
            pmex )
```

4.163.1.2 error()

```
else error (
            'Unsupported platform.\n' )
```

4.163.1.3 eval()

```
eval (
            replace(mexcommand, "Program Files (x86)", "'Program Files (x86)'") )
```

4.163.1.4 fprintf() [1/6]

```
fprintf (
            "%s\n" ,
            mexcommand  )
```

4.163.1.5 fprintf() [2/6]

```
fprintf (
            "data struct contais A,
            b ,
            c\n"  )
```

4.163.1.6 fprintf() [3/6]

```
fprintf (
            "data struct contais A,
            Q ,
            b ,
            c\n"  )
```

4.163.1.7 fprintf() [4/6]

```
fprintf (
            "data struct contais A,
            Q ,
            c ,
            rl ,
            ru ,
            lb ,
            ub and L(optional)\n"  )
```

4.163.1.8 fprintf() [5/6]

```
fprintf (
            "data struct contais X,
            y ,
            lambda\n"  )
```

4.163.1.9 fprintf() [6/6]

```
fprintf ( )
```

4.163.1.10 if() [1/2]

```
if (
            debug  = =0 )
```

4.163.1.11 if() [2/2]

```
end if (
            mex_type  = ="ml" )
```

4.163.2 Variable Documentation

4.163.2.1 alternatively

```
If use we suggest you set the environmental variables by it is typically placed at opt intel
oneapi setvars sh alternatively
```

Definition at line 12 of file make_abip_qcp.m.

4.163.2.2 amd

```
end amd = fullfile(pamd, amdlist)
```

Definition at line 72 of file make_abip_qcp.m.

4.163.2.3 amd_files

```
amd_files = dir( [pamd '/*.c'] )
```

Definition at line 67 of file make_abip_qcp.m.

4.163.2.4 amd_include

```
amd_include = "amd"
```

Definition at line 84 of file make_abip_qcp.m.

4.163.2.5 amdlist

```
amdlist = [ ]
```

Definition at line 68 of file make_abip_qcp.m.

4.163.2.6 cs

```
end cs = fullfile(pcs, cslist)
```

Definition at line 54 of file make_abip_qcp.m.

4.163.2.7 cs_files

```
cs_files = dir( [pcs '/*.c'] )
```

Definition at line 49 of file make_abip_qcp.m.

4.163.2.8 cs_include

```
cs_include = fullfile("csparse", "Include")
```

Definition at line 78 of file make_abip_qcp.m.

4.163.2.9 cslist

```
cslist = [ ]
```

Definition at line 50 of file make_abip_qcp.m.

4.163.2.10 debug

```
debug = 0
```

Definition at line 2 of file make_abip_qcp.m.

4.163.2.11 debugcommand

```
else debugcommand = "-g "
```

Definition at line 92 of file make_abip_qcp.m.

4.163.2.12 example

```
If use we suggest you set the environmental variables by it is typically placed at opt intel
oneapi setvars sh you can set the lib_path to your MKL path by your For example
```

Definition at line 13 of file make_abip_qcp.m.

4.163.2.13 i

```
for i
```

Initial value:

```
= 1:length(src_files)
    srclist = [srclist,convertCharsToStrings(src_files(i).name)]
```

Definition at line 40 of file make_abip_qcp.m.

4.163.2.14 inc

```
inc = [pinc,mkl_include,cs_include,ldl_include, amd_include]
```

Definition at line 86 of file make_abip_qcp.m.

4.163.2.15 intel64

```
If use we suggest you set the environmental variables by it is typically placed at opt intel
oneapi setvars sh you can set the lib_path to your MKL path by your For in you may find it at
opt intel oneapi mkl lib intel64
```

Definition at line 13 of file make_abip_qcp.m.

4.163.2.16 ldl

```
end ldl = fullfile(pldl, ldllist)
```

Definition at line 63 of file make_abip_qcp.m.

4.163.2.17 ldl_files

```
ldl_files = dir( [pldl '/*.c'] )
```

Definition at line 58 of file make_abip_qcp.m.

4.163.2.18 ldl_include

```
ldl_include = fullfile("qdldl", "include")
```

Definition at line 80 of file make_abip_qcp.m.

4.163.2.19 ldllist

```
ldllist = []
```

Definition at line 59 of file make_abip_qcp.m.

4.163.2.20 lib_path

```
lib_path = join(['-L', mkl_lib_path])
```

Definition at line 8 of file make_abip_qcp.m.

4.163.2.21 link

```
link = ' '
```

Definition at line 5 of file make_abip_qcp.m.

4.163.2.22 linux

```
If use we suggest you set the environmental variables by it is typically placed at opt intel
oneapi setvars sh you can set the lib_path to your MKL path by your For in linux
```

Definition at line 13 of file make_abip_qcp.m.

4.163.2.23 mex_file

```
mex_file = fullfile(pmex, ['abip_' mex_type '_mex.c'])
```

Definition at line 36 of file make_abip_qcp.m.

4.163.2.24 mex_type

```
mex_type = 'qcp'
```

Definition at line 1 of file make_abip_qcp.m.

4.163.2.25 mexcommand

```
end mexcommand = "mex " + debugcommand + " -output " + join([mexfname, lib_path, src, inc,
link])
```

Definition at line 94 of file make_abip_qcp.m.

4.163.2.26 mexfname

```
end mexfname = "abip_" + mex_type
```

Definition at line 31 of file make_abip_qcp.m.

4.163.2.27 MKL

```
If use MKL
```

Definition at line 10 of file make_abip_qcp.m.

4.163.2.28 mkl_include

```
mkl_include = fullfile(mkl_path, "include")
```

Definition at line 82 of file make_abip_qcp.m.

4.163.2.29 mkl_lib_path

```
mkl_lib_path = fullfile(MKLROOT, 'lib')
```

Definition at line 7 of file make_abip_qcp.m.

4.163.2.30 MKLROOT

```
MKLROOT = getenv('MKLROOT')
```

Definition at line 6 of file make_abip_qcp.m.

4.163.2.31 oneapi

```
If use we suggest you set the environmental variables by oneapi
```

Definition at line 10 of file make_abip_qcp.m.

4.163.2.32 pamd

```
pamd = fullfile('.', 'amd')
```

Definition at line 65 of file make_abip_qcp.m.

4.163.2.33 pcs

```
pcs = fullfile('.', 'csparse', 'Source')
```

Definition at line 47 of file make_abip_qcp.m.

4.163.2.34 pinc

```
pinc = "include"
```

Definition at line 76 of file make_abip_qcp.m.

4.163.2.35 platform

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

Definition at line 4 of file make_abip_qcp.m.

4.163.2.36 pldl

```
pldl = fullfile('.', 'qdldl', 'src')
```

Definition at line 56 of file make_abip_qcp.m.

4.163.2.37 pmex

```
pmex = fullfile('.', 'mex')
```

Definition at line 34 of file make_abip_qcp.m.

4.163.2.38 psrc

```
psrc = fullfile('.', 'source')
```

Definition at line 32 of file make_abip_qcp.m.

4.163.2.39 self

```
If use we suggest you set the environmental variables by it is typically placed at opt intel
oneapi setvars sh you can set the lib_path to your MKL path by your self
```

Definition at line 12 of file make_abip_qcp.m.

4.163.2.40 src

```
src = fullfile(psrc, srclist)
```

Definition at line 45 of file make_abip_qcp.m.

4.163.2.41 src_files

```
src_files = dir( [psrc '/*.c'] )
```

Definition at line 37 of file make_abip_qcp.m.

4.163.2.42 srclist

```
srclist = []
```

Definition at line 38 of file make_abip_qcp.m.

4.164 make_abip_qcp.m

Go to the documentation of this file.

```
00001 mex_type = 'qcp';
00002 debug = 0;
00003
00004 platform = convertCharsToStrings(computer('arch'));
00005 link = ' ';
00006 MKLROOT = getenv('MKLROOT');
00007 mkl_lib_path = fullfile(MKLROOT, 'lib');
00008 lib_path = join(['-L', mkl_lib_path]);
00009 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
00010 % If use MKL, we suggest you set the environmental variables by oneapi,
00011 %   it is typically placed at /opt/intel/oneapi/setvars.sh
00012 %   alternatively, you can set the lib_path to your MKL path by your self,
00013 % For example, in linux, you may find it at /opt/intel/oneapi/mkl/2021.2.0/lib/intel64,
00014 %   then you may set therein.
00015 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
00016 if platform == "win64"
00017     fprintf('Linking MKL in Windows \n');
00018     link = [link, ' -lmkl_intel_lp64',...
00019         ' -lmkl_core', ' -lmkl_sequential '];
00020 elseif platform == "maci64"
00021     fprintf('Linking MKL in MacOS \n');
00022     link = [link, join(mkl_lib_path + ["/libmkl_intel_lp64.a","/libmkl_core.a", "/libmkl_sequential.a
       "])];
00023 elseif platform == "glnxa64"
00024     fprintf('Linking MKL in Linux \n');
00025     link = [link, ' -lmkl_intel_ilp64',...
00026         ' -lmkl_core', ' -lmkl_sequential '];
00027 else
00028     error('Unsupported platform.\n');
00029 end
00030
00031 mexfname = "abip_" + mex_type;
00032 psrc = fullfile('.', 'source');
00033
00034 pmex = fullfile('.', 'mex');
00035
00036 mex_file = fullfile(pmex, ['abip_'  mex_type  '_mex.c']);
00037 src_files = dir( [psrc '/*.c'] );
00038 srclist = [];
00039
```

```
for i = 1:length(src_files)
    srclist = [srclist,convertCharsToStrings(src_files(i).name)];
end


src = fullfile(psrc, srclist);

pcs = fullfile('.', 'csparse', 'Source');

cs_files = dir( [pcs '/*.c'] );
cslist = [];
for i = 1:length(cs_files)
    cslist = [cslist,convertCharsToStrings(cs_files(i).name)];
end
cs = fullfile(pcs, cslist);

pldl = fullfile('.', 'qdldl', 'src');

ldl_files = dir( [pldl '/*.c'] );
ldllist = [];
for i = 1:length(ldl_files)
    ldllist = [ldllist,convertCharsToStrings(ldl_files(i).name)];
end
ldl = fullfile(pldl, ldllist);

pamd = fullfile('.', 'amd');

amd_files = dir( [pamd '/*.c'] );
amdlist = [];
for i = 1:length(amd_files)
    amdlist = [amdlist,convertCharsToStrings(amd_files(i).name)];
end
amd = fullfile(pamd, amdlist);

src = [src,cs,ldl,mex_file,amd];

pinc = "include";

cs_include = fullfile("csparse", "Include");

ldl_include = fullfile("qdldl", "include");

mkl_include = fullfile(mkl_path, "include");

amd_include = "amd";

inc = [pinc,mkl_include,cs_include,ldl_include, amd_include];
inc = join("-I" + inc);

if(debug == 0)
    debugcommand = "-O";
else
    debugcommand = "-g ";
end
mexcommand = "mex " + debugcommand + " -output " + join([mexfname, lib_path, src, inc, link]);
fprintf("%s\n",mexcommand);
eval(replace(mexcommand, "Program Files (x86)", "'Program Files (x86)'"));

addpath(pmex);

if(mex_type == "ml")
    fprintf("Successfull compiled mex function abip_ml\n");
    fprintf("Usage:[sol,info] = abip_ml(data,settings)\n ");
    fprintf("data struct contais X,y,lambda\n");
end
if(mex_type == "qp")
    fprintf("Successfull compiled mex function abip_qp\n");
    fprintf("Usage:[sol,info] = abip_qp(data,settings) \n ");
    fprintf("data struct contais A,Q,c,rl,ru,lb,ub and L(optional)\n");
end
if(mex_type == "socp")
    fprintf("Successfull compiled mex function abip_socp\n");
    fprintf("Usage:[sol,info] = abip_socp(data,cone,settings)\n ");
    fprintf("data struct contais A,b,c\n");
end
if(mex_type == "qcp")
    fprintf("Successfull compiled mex function abip_qcp\n");
    fprintf("Usage:[sol,info] = abip_qcp(data,cone,settings)\n ");
    fprintf("data struct contais A,Q,b,c\n");
end
```

4.165 mex/abip_ml_mex.c File Reference

```
#include "abip.h"
#include "linsys.h"
#include "matrix.h"
#include "mex.h"
#include "cones.h"
```

Functions

- void mexFunction (int nlhs, mxArray ∗plhs[ ], int nrhs, const mxArray ∗prhs[ ])

4.165.1 Function Documentation

4.165.1.1 mexFunction()

```
void mexFunction (
            int nlhs,
            mxArray * plhs[ ],
            int nrhs,
            const mxArray * prhs[ ] )
```

Definition at line 93 of file abip_ml_mex.c.

4.166 abip_ml_mex.c

Go to the documentation of this file.

```
#include "abip.h"
#include "linsys.h"
#include "matrix.h"
#include "mex.h"
#include "cones.h"


static void free_mex(ABIPData *d, ABIPCone *k) {

   /* if (k) {
        abip_free(k);
    }*/
    if (d) {

        if (d->stgs) {
            abip_free(d->stgs);
        }

        if (d->b) {
            abip_free(d->b);
        }
        if (d->c) {
            abip_free(d->c);
        }

        if (d->A) {

            if (d->A->p) {
                abip_free(d->A->p);
            }
```

```
            if (d->A->i) {
                abip_free(d->A->i);
            }
            if (d->A->x) {
                abip_free(d->A->x);
            }
            abip_free(d->A);
        }
        abip_free(d);
    }
}


/* this memory must be freed */
static abip_int *cast_to_abip_int_arr(mwIndex *arr, abip_int len) {
    abip_int i;
    abip_int *arr_out = (abip_int *)abip_malloc(sizeof(abip_int) * len);
    for (i = 0; i < len; i++) {
        arr_out[i] = (abip_int)arr[i];
    }
    return arr_out;
}


/* this memory must be freed */
static abip_float *cast_to_abip_float_arr(double *arr, abip_int len) {
    abip_int i;
    abip_float *arr_out = (abip_float *)abip_malloc(sizeof(abip_float) * len);
    for (i = 0; i < len; i++) {
        arr_out[i] = (abip_float)arr[i];
    }
    return arr_out;
}

static double *cast_to_double_arr(abip_float *arr, abip_int len) {
    abip_int i;
    double *arr_out = (double *)abip_malloc(sizeof(double) * len);
    for (i = 0; i < len; i++) {
        arr_out[i] = (double)arr[i];
    }
    return arr_out;
}

static void set_output_field(mxArray **pout, abip_float *out, abip_int len) {
    *pout = mxCreateDoubleMatrix(0, 0, mxREAL);
#if SFLOAT > 0
    mxSetPr(*pout, cast_to_double_arr(out, len));
    abip_free(out);
#else
    mxSetPr(*pout, out);
#endif
    mxSetM(*pout, len);
    mxSetN(*pout, 1);
}




/*the input of matlab is data, cone, settings, output is sol, info
  data is struct containing X,y,lambda;
  matlab usage: [sol,info] = abip(data,settings)
*/
void  mexFunction(int  nlhs, mxArray* plhs[], int  nrhs, const  mxArray* prhs[])
{

    const mxArray *data;
    const mxArray *A_mex;
    const mxArray *b_mex;
    const mxArray *lambda_mex;


    const mxArray *settings;
    mxArray *tmp;

    const mwSize one[1] = {1};
    const int num_info_fields = 14;
    const char *info_fields[] = {"ipm_iter", "admm_iter", "status", "pobj", "dobj","res_pri",
 "res_dual",
                                   "gap", "status_val", "setup_time", "solve_time", "runtime",
 "lin_sys_time_per_iter",
                                   "avg_cg_iters"};

    const int svm_num_sol_fields = 3;
    const char *svm_sol_fields[] = {"w", "b", "xi"};

    const int lasso_num_sol_fields = 1;
    const char *lasso_sol_fields[] = {"x"};
```

```


    /* get data*/
    ABIPData* d = (ABIPData*)abip_malloc(sizeof(ABIPData));
    data = prhs[0];
    A_mex = (mxArray *)mxGetField(data, 0, "X");
    if (A_mex == ABIP_NULL) {
        abip_free(d);
        mexErrMsgTxt("ABIPData struct must contain a `X` entry.");
    }
    if (!mxIsSparse(A_mex)) {
        abip_free(d);
        mexErrMsgTxt("Input matrix X must be in sparse format (pass in sparse(X))");
    }
    b_mex = (mxArray *)mxGetField(data, 0, "y");
    if (b_mex == ABIP_NULL) {
        abip_free(d);
        mexErrMsgTxt("ABIPData struct must contain a `y` entry.");
    }
    if (mxIsSparse(b_mex)) {
        abip_free(d);
        mexErrMsgTxt("Input vector y must be in dense format (pass in full(y))");
    }
    lambda_mex = (mxArray *)mxGetField(data, 0, "lambda");
    if (lambda_mex == ABIP_NULL) {
        abip_free(d);
        mexErrMsgTxt("ABIPData struct must contain a `lambda` entry.");
    }


    d->lambda = (abip_float)*mxGetPr(lambda_mex);
    d->m = mxGetM(A_mex);
    d->n = mxGetN(A_mex);
    ABIPMatrix *A = (ABIPMatrix *)abip_malloc(sizeof(ABIPMatrix));
    A->m = d->m;
    A->n = d->n;

    d->b = cast_to_abip_float_arr(mxGetPr(b_mex), d->m);

    A->p = cast_to_abip_int_arr(mxGetJc(A_mex), A->n + 1);
    A->i = cast_to_abip_int_arr(mxGetIr(A_mex), A->p[A->n]);
    A->x = cast_to_abip_float_arr(mxGetPr(A_mex), A->p[A->n]);

    d->A = A;
    d->c = ABIP_NULL;

    /*get settings*/
    settings = prhs[1];
    d->stgs = (ABIPSettings*)abip_malloc(sizeof(ABIPSettings));
    ABIP(set_default_settings)(d);

    tmp = mxGetField(settings, 0, "alpha");
    if (tmp != ABIP_NULL) {
        d->stgs->alpha = (abip_float)*mxGetPr(tmp);
    }

    tmp = mxGetField(settings, 0, "cg_rate");
    if (tmp != ABIP_NULL) {
        d->stgs->cg_rate = (abip_float)*mxGetPr(tmp);
    }

    tmp = mxGetField(settings, 0, "eps");
    if (tmp != ABIP_NULL) {
        d->stgs->eps = (abip_float)*mxGetPr(tmp);
        d->stgs->eps_p = d->stgs->eps;
        d->stgs->eps_d = d->stgs->eps;
        d->stgs->eps_g = d->stgs->eps;
        d->stgs->eps_inf = d->stgs->eps;
        d->stgs->eps_unb = d->stgs->eps;
    }
    tmp = mxGetField(settings, 0, "eps_p");
    if (tmp != ABIP_NULL) {
        d->stgs->eps_p = (abip_float)*mxGetPr(tmp);
    }
    tmp = mxGetField(settings, 0, "eps_d");
    if (tmp != ABIP_NULL) {
        d->stgs->eps_d = (abip_float)*mxGetPr(tmp);
    }
    tmp = mxGetField(settings, 0, "eps_g");
    if (tmp != ABIP_NULL) {
        d->stgs->eps_g = (abip_float)*mxGetPr(tmp);
    }
        tmp = mxGetField(settings, 0, "eps_inf");
    if (tmp != ABIP_NULL) {
        d->stgs->eps_inf = (abip_float)*mxGetPr(tmp);
    }     tmp = mxGetField(settings, 0, "eps_unb");
    if (tmp != ABIP_NULL) {
```

```
        d->stgs->eps_unb = (abip_float)*mxGetPr(tmp);
    }

    tmp = mxGetField(settings, 0, "max_admm_iters");
    if (tmp != ABIP_NULL) {
        d->stgs->max_admm_iters = (abip_int)*mxGetPr(tmp);
    }

    tmp = mxGetField(settings, 0, "max_ipm_iters");
    if (tmp != ABIP_NULL) {
        d->stgs->max_ipm_iters = (abip_int)*mxGetPr(tmp);
    }

    tmp = mxGetField(settings, 0, "normalize");
    if (tmp != ABIP_NULL) {
        d->stgs->normalize = (abip_int)*mxGetPr(tmp);
    }

    tmp = mxGetField(settings, 0, "rho_y");
    if (tmp != ABIP_NULL) {
        d->stgs->rho_y = (abip_float)*mxGetPr(tmp);
    }

    tmp = mxGetField(settings, 0, "rho_x");
    if (tmp != ABIP_NULL) {
        d->stgs->rho_x = (abip_float)*mxGetPr(tmp);
    }

    tmp = mxGetField(settings, 0, "rho_tau");
    if (tmp != ABIP_NULL) {
        d->stgs->rho_tau = (abip_float)*mxGetPr(tmp);
    }

    tmp = mxGetField(settings, 0, "scale");
    if (tmp != ABIP_NULL) {
        d->stgs->scale = (abip_int)*mxGetPr(tmp);
    }

    tmp = mxGetField(settings, 0, "scale_bc");
    if (tmp != ABIP_NULL) {
        d->stgs->scale_bc = (abip_int)*mxGetPr(tmp);
    }

    tmp = mxGetField(settings, 0, "scale_E");
    if (tmp != ABIP_NULL) {
        d->stgs->scale_E = (abip_int)*mxGetPr(tmp);
    }

    tmp = mxGetField(settings, 0, "use_indirect");
    if (tmp != ABIP_NULL) {
        d->stgs->use_indirect = (abip_int)*mxGetPr(tmp);
    }

    tmp = mxGetField(settings, 0, "verbose");
    if (tmp != ABIP_NULL) {
        d->stgs->verbose = (abip_int)*mxGetPr(tmp);
    }

    tmp = mxGetField(settings, 0, "linsys_solver");
    if (tmp != ABIP_NULL) {
        d->stgs->linsys_solver = (abip_int)*mxGetPr(tmp);
    }

    tmp = mxGetField(settings, 0, "prob_type");
    if (tmp != ABIP_NULL) {
        d->stgs->prob_type = (abip_int)*mxGetPr(tmp);
        if(d->stgs->prob_type != LASSO && d->stgs->prob_type != SVM && d->stgs->prob_type != SVMQP){
            mexErrMsgTxt("Invalid problem type");
        }
    }
    else{
        mexErrMsgTxt("Please input the machine learning problem type");
    }

    tmp = mxGetField(settings, 0, "inner_check_period");
    if (tmp != ABIP_NULL) {
        d->stgs->inner_check_period = (abip_int)*mxGetPr(tmp);
    }

    tmp = mxGetField(settings, 0, "outer_check_period");
    if (tmp != ABIP_NULL) {
        d->stgs->outer_check_period = (abip_int)*mxGetPr(tmp);
    }

    tmp = mxGetField(settings, 0, "err_dif");
    if (tmp != ABIP_NULL) {
        d->stgs->err_dif = (abip_float)*mxGetPr(tmp);
```

```
    }

    tmp = mxGetField(settings, 0, "time_limit");
    if (tmp != ABIP_NULL) {
        d->stgs->time_limit = (abip_float)*mxGetPr(tmp);
    }
    tmp = mxGetField(settings, 0, "psi");
    if (tmp != ABIP_NULL) {
        d->stgs->psi = (abip_float)*mxGetPr(tmp);
    }

    tmp = mxGetField(settings, 0, "origin_scaling");
    if (tmp != ABIP_NULL) {
        d->stgs->origin_scaling = (abip_int)*mxGetPr(tmp);
    }

    tmp = mxGetField(settings, 0, "ruiz_scaling");
    if (tmp != ABIP_NULL) {
        d->stgs->ruiz_scaling = (abip_int)*mxGetPr(tmp);
    }

    tmp = mxGetField(settings, 0, "pc_scaling");
    if (tmp != ABIP_NULL) {
        d->stgs->pc_scaling = (abip_int)*mxGetPr(tmp);
    }


    ABIPCone* k = (ABIPCone*)abip_malloc(sizeof(ABIPCone));

    k->q = ABIP_NULL;
    k->qsize = 0;

    k->rqsize = 1;
    k->rq = (abip_int *)abip_malloc(sizeof(abip_int));

    k->f = 0;
    k->z = 0;

    if(d->stgs->prob_type == LASSO) {

        k->rq[0] = 2 + d->m;
        k->l = 2 * d->n;
    }
    else if(d->stgs->prob_type == SVM) {

        k->rq[0] = 2 + d->n;
        k->l = 2 + 2 * d->m + 2 * d->n;
    }
    else if(d->stgs->prob_type == SVMQP){
        k->rqsize = 0;
        k->rq = ABIP_NULL;
        k->f = d->n + 1;
        k->l = 2 * d->m;
    }
    else{
        mexErrMsgTxt("This type of machine learning problem is not supported yet");
    }

    ABIPSolution* sol = (ABIPSolution*)abip_malloc(sizeof(ABIPSolution));
    ABIPInfo* info = (ABIPInfo*)abip_malloc(sizeof(ABIPInfo));

    sol->x = ABIP_NULL;
    sol->y = ABIP_NULL;
    sol->s = ABIP_NULL;



    abip_int status = abip(d,sol,info,k);

    /* output sol */

    //SVM
    if(d->stgs->prob_type == 2 || d->stgs->prob_type == 4){

        plhs[0] = mxCreateStructArray(1, one, svm_num_sol_fields, svm_sol_fields);

        set_output_field(&tmp, sol->x, d->n);
        mxSetField(plhs[0], 0, "w", tmp);

        set_output_field(&tmp, sol->y, 1);
        mxSetField(plhs[0], 0, "b", tmp);

        set_output_field(&tmp, sol->s, d->m);
        mxSetField(plhs[0], 0, "xi", tmp);
    }
    //LASSO
    else{
```

```
        plhs[0] = mxCreateStructArray(1, one, lasso_num_sol_fields, lasso_sol_fields);

        set_output_field(&tmp, sol->x, d->n);
        mxSetField(plhs[0], 0, "x", tmp);
    }

    /* output info */
    plhs[1] = mxCreateStructArray(1, one, num_info_fields, info_fields);

    /* if you add/remove fields here update the info_fields above */
    mxSetField(plhs[1], 0, "status", mxCreateString(info->status));

    tmp = mxCreateDoubleMatrix(1, 1, mxREAL);
    mxSetField(plhs[1], 0, "ipm_iter", tmp);
    *mxGetPr(tmp) = (abip_float)info->ipm_iter;

    tmp = mxCreateDoubleMatrix(1, 1, mxREAL);
    mxSetField(plhs[1], 0, "admm_iter", tmp);
    *mxGetPr(tmp) = (abip_float)info->admm_iter;

    tmp = mxCreateDoubleMatrix(1, 1, mxREAL);
    mxSetField(plhs[1], 0, "status_val", tmp);
    *mxGetPr(tmp) = (abip_float)info->status_val;

    tmp = mxCreateDoubleMatrix(1, 1, mxREAL);
    mxSetField(plhs[1], 0, "pobj", tmp);
    *mxGetPr(tmp) = info->pobj;

    tmp = mxCreateDoubleMatrix(1, 1, mxREAL);
    mxSetField(plhs[1], 0, "dobj", tmp);
    *mxGetPr(tmp) = info->dobj;

    tmp = mxCreateDoubleMatrix(1, 1, mxREAL);
    mxSetField(plhs[1], 0, "res_pri", tmp);
    *mxGetPr(tmp) = info->res_pri;

    tmp = mxCreateDoubleMatrix(1, 1, mxREAL);
    mxSetField(plhs[1], 0, "res_dual", tmp);
    *mxGetPr(tmp) = info->res_dual;

    tmp = mxCreateDoubleMatrix(1, 1, mxREAL);
    mxSetField(plhs[1], 0, "gap", tmp);
    *mxGetPr(tmp) = info->rel_gap;


    /*return value in secs */
    tmp = mxCreateDoubleMatrix(1, 1, mxREAL);
    mxSetField(plhs[1], 0, "setup_time", tmp);
    *mxGetPr(tmp) = info->setup_time / 1e3;

    /*return value in secs */
    tmp = mxCreateDoubleMatrix(1, 1, mxREAL);
    mxSetField(plhs[1], 0, "solve_time", tmp);
    *mxGetPr(tmp) = info->solve_time / 1e3;

    /*return value in secs */
    tmp = mxCreateDoubleMatrix(1, 1, mxREAL);
    mxSetField(plhs[1], 0, "runtime", tmp);
    *mxGetPr(tmp) = (info->solve_time + info->setup_time) / 1e3;

    /*return value in secs */
    tmp = mxCreateDoubleMatrix(1, 1, mxREAL);
    mxSetField(plhs[1], 0, "lin_sys_time_per_iter", tmp);
    *mxGetPr(tmp) = info->avg_linsys_time / 1e3;

    //average cg iters per admm iter
    tmp = mxCreateDoubleMatrix(1, 1, mxREAL);
    mxSetField(plhs[1], 0, "avg_cg_iters", tmp);
    *mxGetPr(tmp) = info->avg_cg_iters;

    free_mex(d, k);
    return;
}
```

4.167 mex/abip_qcp_mex.c File Reference

```
#include "abip.h"
#include "linsys.h"
#include "matrix.h"
#include "mex.h"
```

```
#include "cones.h"
```

Functions

- void mexFunction (int nlhs, mxArray ∗plhs[ ], int nrhs, const mxArray ∗prhs[ ])

4.167.1 Function Documentation

4.167.1.1 mexFunction()

```
void mexFunction (
            int nlhs,
            mxArray * plhs[ ],
            int nrhs,
            const mxArray * prhs[ ] )
```

Definition at line 109 of file abip_qcp_mex.c.

4.168 abip_qcp_mex.c

Go to the documentation of this file.

```
00001 #include "abip.h"
00002 #include "linsys.h"
00003 #include "matrix.h"
00004 #include "mex.h"
00005 #include "cones.h"
00006
00007 /*
00008 This is mex file for creating matlab routine:
00009
00010 function [results, info] = abip_qp(data, cones, params)
00011
00012     min        1/2x'Qx + c'x
00013 subject to          Ax = b
00014                      x in K
00015
00016 where A \in R^m*n, b \in R^m, c \in R^n
00017
00018 data:
00019 A:      m*n matrix used as above
00020 b:      m dimensional column vector used as above
00021 c:      n dimensional column vector used as above
00022
00023 K:      covex cone, now SOC, RSOC, free cone, zero cone and positive orthant are supported
00024 */
00025
00026 static void free_mex(ABIPData *d, ABIPCone *k) {
00027
00028
00029    if (k) {
00030
00031         if(k->q){
00032             abip_free(k->q);
00033         }
00034         if(k->rq){
00035             abip_free(k->rq);
00036         }
00037         abip_free(k);
00038     }
00039     if (d) {
00040
```

```
        if(d->A){

        abip_free(d->A->x);

        abip_free(d->A->i);
        abip_free(d->A->p);

        }

        if (d->b) {
            abip_free(d->b);
        }
        if (d->c) {
            abip_free(d->c);
        }

        abip_free(d);
    }
}



#if !(DLONG > 0)
/* this memory must be freed */
static abip_int *cast_to_abip_int_arr(mwIndex *arr, abip_int len) {
    abip_int i;
    abip_int *arr_out = (abip_int *)abip_malloc(sizeof(abip_int) * len);
    for (i = 0; i < len; i++) {
        arr_out[i] = (abip_int)arr[i];
    }
    return arr_out;
}
#endif


/* this memory must be freed */
static abip_float *cast_to_abip_float_arr(double *arr, abip_int len) {
    abip_int i;
    abip_float *arr_out = (abip_float *)abip_malloc(sizeof(abip_float) * len);
    for (i = 0; i < len; i++) {
        arr_out[i] = (abip_float)arr[i];
    }
    return arr_out;
}

static double *cast_to_double_arr(abip_float *arr, abip_int len) {
    abip_int i;
    double *arr_out = (double *)abip_malloc(sizeof(double) * len);
    for (i = 0; i < len; i++) {
        arr_out[i] = (double)arr[i];
    }
    return arr_out;
}


static void set_output_field(mxArray **pout, abip_float *out, abip_int len) {
    *pout = mxCreateDoubleMatrix(0, 0, mxREAL);
#if SFLOAT > 0
    mxSetPr(*pout, cast_to_double_arr(out, len));
    abip_free(out);
#else
    mxSetPr(*pout, out);
#endif
    mxSetM(*pout, len);
    mxSetN(*pout, 1);
}


void  mexFunction(int  nlhs, mxArray* plhs[], int  nrhs, const  mxArray* prhs[])
{

    const mxArray *data;
    const mxArray *A_mex;
    const mxArray *Q_mex;
    const mxArray *b_mex;
    const mxArray *c_mex;
    const mxArray *cone;
    const mxArray *settings;
    const mxArray *kq;
    const mxArray *krq;
    const mxArray *kf;
    const mxArray *kz;
    const mxArray *kl;
    const double *q_mex;
    const double *rq_mex;
    const size_t *q_dims;
    const size_t *rq_dims;
```

```
    abip_int ns;

    mxArray *tmp;

    const mwSize one[1] = {1};
    const int num_info_fields = 14;
    const char *info_fields[] = {"ipm_iter", "admm_iter", "status", "pobj", "dobj","res_pri",
"res_dual",
                                       "gap", "status_val", "setup_time", "solve_time", "runtime",
"lin_sys_time_per_iter",
                                       "avg_cg_iters"};

    const int num_sol_fields = 3;
    const char *sol_fields[] = {"x", "y", "s"};


    /* get data*/
    ABIPData* d = (ABIPData*)abip_malloc(sizeof(ABIPData));
    data = prhs[0];


    b_mex = (mxArray *)mxGetField(data, 0, "b");
    if (b_mex == ABIP_NULL) {
        d->b = ABIP_NULL;
        printf("ABIPData doesn't contain 'b'\n");
    }
    else if (mxIsSparse(b_mex)) {
        abip_free(d);
        mexErrMsgTxt("Input vector b must be in dense format (pass in full(b))");
    }
    else{

        d->b = cast_to_abip_float_arr(mxGetPr(b_mex), MAX(mxGetM(b_mex), mxGetN(b_mex)));

    }

    c_mex = (mxArray *)mxGetField(data, 0, "c");
    if (c_mex == ABIP_NULL) {
        abip_free(d);
        mexErrMsgTxt("ABIPData struct must contain a 'c' entry.");
    }
    else if (mxIsSparse(c_mex)) {
        abip_free(d);
        mexErrMsgTxt("Input vector c must be in dense format (pass in full(c))");
    }
    else{

        d->c = cast_to_abip_float_arr(mxGetPr(c_mex), MAX(mxGetM(c_mex), mxGetN(c_mex)));

    }

    A_mex = (mxArray *)mxGetField(data, 0, "A");
    if (A_mex == ABIP_NULL) {
        d->A = ABIP_NULL;
        d->m = 0;
        d->n = MAX(mxGetM(c_mex), mxGetN(c_mex));

        printf("ABIPData doesn't contain 'A'\n");
    }
    else if (!mxIsSparse(A_mex)) {
        abip_free(d);
        mexErrMsgTxt("Input matrix A must be in sparse format (pass in sparse(A))");
    }
    else{
        d->A = (ABIPMatrix*)abip_malloc(sizeof(ABIPMatrix));
        ABIPMatrix *A = d->A;

        A->m = mxGetM(A_mex);
        A->n = mxGetN(A_mex);

        d->m = A->m;
        d->n = A->n;


        A->p = cast_to_abip_int_arr(mxGetJc(A_mex), A->n + 1);
        A->i = cast_to_abip_int_arr(mxGetIr(A_mex), A->p[A->n]);

        A->x = cast_to_abip_float_arr(mxGetPr(A_mex), A->p[A->n]);

    }

    Q_mex = (mxArray *)mxGetField(data, 0, "Q");
    if (Q_mex == ABIP_NULL){
        d->Q = ABIP_NULL;
        printf("ABIPData doesn't contain 'Q'\n");
    }
    else if (!mxIsSparse(Q_mex)) {
```

```
        abip_free(d);
        mexErrMsgTxt("Input matrix Q must be in sparse format (pass in sparse(Q))");
    }
    else{
        d->Q = (ABIPMatrix*)abip_malloc(sizeof(ABIPMatrix));
        ABIPMatrix *Q = d->Q;

        Q->m = mxGetM(Q_mex);
        Q->n = mxGetN(Q_mex);


        Q->p = cast_to_abip_int_arr(mxGetJc(Q_mex), Q->n + 1);
        Q->i = cast_to_abip_int_arr(mxGetIr(Q_mex), Q->p[Q->n]);
        Q->x = cast_to_abip_float_arr(mxGetPr(Q_mex), Q->p[Q->n]);

    }

    /*get cone*/
    cone = prhs[1];

    ABIPCone *k = (ABIPCone*)abip_malloc(sizeof(ABIPCone));

    kq = mxGetField(cone, 0, "q");
    if (kq && !mxIsEmpty(kq)) {
        q_mex = mxGetPr(kq);
        ns = (abip_int)mxGetNumberOfDimensions(kq);
        q_dims = mxGetDimensions(kq);
        k->qsize = (abip_int)q_dims[0];
        if (ns > 1 && q_dims[0] == 1) {
        k->qsize = (abip_int)q_dims[1];
        }
        k->q = (abip_int *)mxMalloc(sizeof(abip_int) * k->qsize);
        for (abip_int i = 0; i < k->qsize; i++) {
        k->q[i] = (abip_int)q_mex[i];
        }
    } else {
        k->qsize = 0;
        k->q = ABIP_NULL;
    }

    krq = mxGetField(cone, 0, "rq");
    if (krq && !mxIsEmpty(krq)) {
        rq_mex = mxGetPr(krq);
        ns = (abip_int)mxGetNumberOfDimensions(krq);
        rq_dims = mxGetDimensions(krq);
        k->rqsize = (abip_int)rq_dims[0];
        if (ns > 1 && rq_dims[0] == 1) {
        k->rqsize = (abip_int)rq_dims[1];
        }
        k->rq = (abip_int *)mxMalloc(sizeof(abip_int) * k->rqsize);
        for (abip_int i = 0; i < k->rqsize; i++) {
        k->rq[i] = (abip_int)rq_mex[i];
        }
    } else {
        k->rqsize = 0;
        k->rq = ABIP_NULL;
    }

    kf = mxGetField(cone, 0, "f");
    if (kf && !mxIsEmpty(kf)) {
        k->f = (abip_int)*mxGetPr(kf);
    } else {
        k->f = 0;
    }

    kz = mxGetField(cone, 0, "z");
    if (kz && !mxIsEmpty(kz)) {
        k->z = (abip_int)*mxGetPr(kz);
    } else {
        k->z = 0;
    }

    kl = mxGetField(cone, 0, "l");
    if (kl && !mxIsEmpty(kl)) {
        k->l = (abip_int)*mxGetPr(kl);
    } else {
        k->l = 0;
    }

    /*get settings*/
    settings = prhs[2];
    d->stgs = (ABIPSettings*)abip_malloc(sizeof(ABIPSettings));
    ABIP(set_default_settings)(d);

    tmp = mxGetField(settings, 0, "alpha");
    if (tmp != ABIP_NULL) {
        d->stgs->alpha = (abip_float)*mxGetPr(tmp);
```

```
    }

    tmp = mxGetField(settings, 0, "cg_rate");
    if (tmp != ABIP_NULL) {
        d->stgs->cg_rate = (abip_float)*mxGetPr(tmp);
    }

    tmp = mxGetField(settings, 0, "eps");
    if (tmp != ABIP_NULL) {
        d->stgs->eps = (abip_float)*mxGetPr(tmp);
        d->stgs->eps_p = d->stgs->eps;
        d->stgs->eps_d = d->stgs->eps;
        d->stgs->eps_g = d->stgs->eps;
        d->stgs->eps_inf = d->stgs->eps;
        d->stgs->eps_unb = d->stgs->eps;
    }
    tmp = mxGetField(settings, 0, "eps_p");
    if (tmp != ABIP_NULL) {
        d->stgs->eps_p = (abip_float)*mxGetPr(tmp);
    }
    tmp = mxGetField(settings, 0, "eps_d");
    if (tmp != ABIP_NULL) {
        d->stgs->eps_d = (abip_float)*mxGetPr(tmp);
    }
    tmp = mxGetField(settings, 0, "eps_g");
    if (tmp != ABIP_NULL) {
        d->stgs->eps_g = (abip_float)*mxGetPr(tmp);
    }
        tmp = mxGetField(settings, 0, "eps_inf");
    if (tmp != ABIP_NULL) {
        d->stgs->eps_inf = (abip_float)*mxGetPr(tmp);
    }    tmp = mxGetField(settings, 0, "eps_unb");
    if (tmp != ABIP_NULL) {
        d->stgs->eps_unb = (abip_float)*mxGetPr(tmp);
    }

    tmp = mxGetField(settings, 0, "max_admm_iters");
    if (tmp != ABIP_NULL) {
        d->stgs->max_admm_iters = (abip_int)*mxGetPr(tmp);
    }

    tmp = mxGetField(settings, 0, "max_ipm_iters");
    if (tmp != ABIP_NULL) {
        d->stgs->max_ipm_iters = (abip_int)*mxGetPr(tmp);
    }

    tmp = mxGetField(settings, 0, "normalize");
    if (tmp != ABIP_NULL) {
        d->stgs->normalize = (abip_int)*mxGetPr(tmp);
    }

    tmp = mxGetField(settings, 0, "rho_y");
    if (tmp != ABIP_NULL) {
        d->stgs->rho_y = (abip_float)*mxGetPr(tmp);
    }

    tmp = mxGetField(settings, 0, "rho_x");
    if (tmp != ABIP_NULL) {
        d->stgs->rho_x = (abip_float)*mxGetPr(tmp);
    }

    tmp = mxGetField(settings, 0, "rho_tau");
    if (tmp != ABIP_NULL) {
        d->stgs->rho_tau = (abip_float)*mxGetPr(tmp);
    }

    tmp = mxGetField(settings, 0, "scale");
    if (tmp != ABIP_NULL) {
        d->stgs->scale = (abip_int)*mxGetPr(tmp);
    }

    tmp = mxGetField(settings, 0, "scale_bc");
    if (tmp != ABIP_NULL) {
        d->stgs->scale_bc = (abip_int)*mxGetPr(tmp);
    }

    tmp = mxGetField(settings, 0, "scale_E");
    if (tmp != ABIP_NULL) {
        d->stgs->scale_E = (abip_int)*mxGetPr(tmp);
    }

    tmp = mxGetField(settings, 0, "use_indirect");
    if (tmp != ABIP_NULL) {
        d->stgs->use_indirect = (abip_int)*mxGetPr(tmp);
    }

    tmp = mxGetField(settings, 0, "verbose");
```

```
    if (tmp != ABIP_NULL) {
        d->stgs->verbose = (abip_int)*mxGetPr(tmp);
    }

    tmp = mxGetField(settings, 0, "linsys_solver");
    if (tmp != ABIP_NULL) {
        d->stgs->linsys_solver = (abip_int)*mxGetPr(tmp);
    }

    tmp = mxGetField(settings, 0, "inner_check_period");
    if (tmp != ABIP_NULL) {
        d->stgs->inner_check_period = (abip_int)*mxGetPr(tmp);
    }

    tmp = mxGetField(settings, 0, "outer_check_period");
    if (tmp != ABIP_NULL) {
        d->stgs->outer_check_period = (abip_int)*mxGetPr(tmp);
    }

    tmp = mxGetField(settings, 0, "err_dif");
    if (tmp != ABIP_NULL) {
        d->stgs->err_dif = (abip_float)*mxGetPr(tmp);
    }

    tmp = mxGetField(settings, 0, "time_limit");
    if (tmp != ABIP_NULL) {
        d->stgs->time_limit = (abip_float)*mxGetPr(tmp);
    }

    tmp = mxGetField(settings, 0, "psi");
    if (tmp != ABIP_NULL) {
        d->stgs->psi = (abip_float)*mxGetPr(tmp);
    }

    tmp = mxGetField(settings, 0, "origin_scaling");
    if (tmp != ABIP_NULL) {
        d->stgs->origin_scaling = (abip_int)*mxGetPr(tmp);
    }

    tmp = mxGetField(settings, 0, "ruiz_scaling");
    if (tmp != ABIP_NULL) {
        d->stgs->ruiz_scaling = (abip_int)*mxGetPr(tmp);
    }

    tmp = mxGetField(settings, 0, "pc_scaling");
    if (tmp != ABIP_NULL) {
        d->stgs->pc_scaling = (abip_int)*mxGetPr(tmp);
    }

    d->stgs->prob_type = QCP; //QCP

    ABIPSolution* sol = (ABIPSolution*)abip_malloc(sizeof(ABIPSolution));
    ABIPInfo* info = (ABIPInfo*)abip_malloc(sizeof(ABIPInfo));

    sol->x = ABIP_NULL;
    sol->y = ABIP_NULL;
    sol->s = ABIP_NULL;

    abip_int status = abip(d,sol,info,k);

    /* output sol */
    plhs[0] = mxCreateStructArray(1, one, num_sol_fields, sol_fields);

    set_output_field(&tmp, sol->x, d->n);
    mxSetField(plhs[0], 0, "x", tmp);

    set_output_field(&tmp, sol->y, d->m);
    mxSetField(plhs[0], 0, "y", tmp);

    set_output_field(&tmp, sol->s, d->n);
    mxSetField(plhs[0], 0, "s", tmp);

    /* output info */
    plhs[1] = mxCreateStructArray(1, one, num_info_fields, info_fields);

    /* if you add/remove fields here update the info_fields above */
    mxSetField(plhs[1], 0, "status", mxCreateString(info->status));

    tmp = mxCreateDoubleMatrix(1, 1, mxREAL);
    mxSetField(plhs[1], 0, "ipm_iter", tmp);
    *mxGetPr(tmp) = (abip_float)info->ipm_iter;

    tmp = mxCreateDoubleMatrix(1, 1, mxREAL);
    mxSetField(plhs[1], 0, "admm_iter", tmp);
    *mxGetPr(tmp) = (abip_float)info->admm_iter;

    tmp = mxCreateDoubleMatrix(1, 1, mxREAL);
```

```
    mxSetField(plhs[1], 0, "status_val", tmp);
    *mxGetPr(tmp) = (abip_float)info->status_val;

    tmp = mxCreateDoubleMatrix(1, 1, mxREAL);
    mxSetField(plhs[1], 0, "pobj", tmp);
    *mxGetPr(tmp) = info->pobj;

    tmp = mxCreateDoubleMatrix(1, 1, mxREAL);
    mxSetField(plhs[1], 0, "dobj", tmp);
    *mxGetPr(tmp) = info->dobj;

    tmp = mxCreateDoubleMatrix(1, 1, mxREAL);
    mxSetField(plhs[1], 0, "res_pri", tmp);
    *mxGetPr(tmp) = info->res_pri;

    tmp = mxCreateDoubleMatrix(1, 1, mxREAL);
    mxSetField(plhs[1], 0, "res_dual", tmp);
    *mxGetPr(tmp) = info->res_dual;

    tmp = mxCreateDoubleMatrix(1, 1, mxREAL);
    mxSetField(plhs[1], 0, "gap", tmp);
    *mxGetPr(tmp) = info->rel_gap;

    /*return value in secs */
    tmp = mxCreateDoubleMatrix(1, 1, mxREAL);
    mxSetField(plhs[1], 0, "setup_time", tmp);
    *mxGetPr(tmp) = info->setup_time / 1e3;

    /*return value in secs */
    tmp = mxCreateDoubleMatrix(1, 1, mxREAL);
    mxSetField(plhs[1], 0, "solve_time", tmp);
    *mxGetPr(tmp) = info->solve_time / 1e3;

    /*return value in secs */
    tmp = mxCreateDoubleMatrix(1, 1, mxREAL);
    mxSetField(plhs[1], 0, "runtime", tmp);
    *mxGetPr(tmp) = (info->setup_time + info->solve_time) / 1e3;

    /*return value in secs */
    tmp = mxCreateDoubleMatrix(1, 1, mxREAL);
    mxSetField(plhs[1], 0, "lin_sys_time_per_iter", tmp);
    *mxGetPr(tmp) = info->avg_linsys_time / 1e3;

    //average cg iters per admm iter
    tmp = mxCreateDoubleMatrix(1, 1, mxREAL);
    mxSetField(plhs[1], 0, "avg_cg_iters", tmp);
    *mxGetPr(tmp) = info->avg_cg_iters;


    free_mex(d, k);
    return;
}
```

4.169 qdldl/include/qdldl.h File Reference

```
#include "qdldl_types.h"
```

Functions

- QDLDL_int QDLDL_etree (const QDLDL_int n, const QDLDL_int ∗Ap, const QDLDL_int ∗Ai, QDLDL_int ∗work, QDLDL_int ∗Lnz, QDLDL_int ∗etree)
- QDLDL_int QDLDL_factor (const QDLDL_int n, const QDLDL_int ∗Ap, const QDLDL_int ∗Ai, const QDLDL_float ∗Ax, QDLDL_int ∗Lp, QDLDL_int ∗Li, QDLDL_float ∗Lx, QDLDL_float ∗D, QDLDL_float ∗Dinv, const QDLDL_int ∗Lnz, const QDLDL_int ∗etree, QDLDL_bool ∗bwork, QDLDL_int ∗iwork, QDLDL_float ∗fwork)
- void QDLDL_solve (const QDLDL_int n, const QDLDL_int ∗Lp, const QDLDL_int ∗Li, const QDLDL_float ∗Lx, const QDLDL_float ∗Dinv, QDLDL_float ∗x)
- void QDLDL_Lsolve (const QDLDL_int n, const QDLDL_int ∗Lp, const QDLDL_int ∗Li, const QDLDL_float ∗Lx, QDLDL_float ∗x)
- void QDLDL_Ltsolve (const QDLDL_int n, const QDLDL_int ∗Lp, const QDLDL_int ∗Li, const QDLDL_float ∗Lx, QDLDL_float ∗x)

4.169.1 Function Documentation

4.169.1.1 QDLDL_etree()

```
QDLDL_int QDLDL_etree (
            const QDLDL_int n,
            const QDLDL_int * Ap,
            const QDLDL_int * Ai,
            QDLDL_int * work,
            QDLDL_int * Lnz,
            QDLDL_int * etree )
```

Compute the elimination tree for a quasidefinite matrix in compressed sparse column form, where the input matrix is assumed to contain data for the upper triangular part of A only, and there are no duplicate indices.

Returns an elimination tree for the factorization A = LDL^T and a count of the nonzeros in each column of L that are strictly below the diagonal.

Does not use MALLOC. It is assumed that the arrays work, Lnz, and etree will be allocated with a number of elements equal to n.

The data in (n,Ap,Ai) are from a square matrix A in CSC format, and should include the upper triangular part of A only.

This function is only intended for factorisation of QD matrices specified by their upper triangular part. An error is returned if any column has data below the diagonal or s completely empty.

For matrices with a non-empty column but a zero on the corresponding diagonal, this function will *not* return an error, as it may still be possible to factor such a matrix in LDL form. No promises are made in this case though...

Parameters

n	number of columns in CSC matrix A (assumed square)
Ap	column pointers (size n+1) for columns of A
Ai	row indices of A. Has Ap[n] elements
work	work vector (size n) (no meaning on return)
Lnz	count of nonzeros in each column of L (size n) below diagonal
etree	elimination tree (size n)

Returns

total sum of Lnz (i.e. total nonzeros in L below diagonal). Returns -1 if the input is not triu or has an empty column. Returns -2 if the return value overflows QDLDL_int.

Definition at line 11 of file qdldl.c.

4.169.1.2 QDLDL_factor()

```
QDLDL_int QDLDL_factor (
            const QDLDL_int n,
            const QDLDL_int * Ap,
            const QDLDL_int * Ai,
            const QDLDL_float * Ax,
            QDLDL_int * Lp,
            QDLDL_int * Li,
            QDLDL_float * Lx,
            QDLDL_float * D,
            QDLDL_float * Dinv,
            const QDLDL_int * Lnz,
            const QDLDL_int * etree,
            QDLDL_bool * bwork,
            QDLDL_int * iwork,
            QDLDL_float * fwork )
```

Compute an LDL decomposition for a quasidefinite matrix in compressed sparse column form, where the input matrix is assumed to contain data for the upper triangular part of A only, and there are no duplicate indices.

Returns factors L, D and Dinv = 1./D.

Does not use MALLOC. It is assumed that L will be a compressed sparse column matrix with data (n,Lp,Li,Lx) with sufficient space allocated, with a number of nonzeros equal to the count given as a return value by QDLDL_etree

Parameters

n	number of columns in L and A (both square)
Ap	column pointers (size n+1) for columns of A (not modified)
Ai	row indices of A. Has Ap[n] elements (not modified)
Ax	data of A. Has Ap[n] elements (not modified)
Lp	column pointers (size n+1) for columns of L
Li	row indices of L. Has Lp[n] elements
Lx	data of L. Has Lp[n] elements
D	vectorized factor D. Length is n
Dinv	reciprocal of D. Length is n
Lnz	count of nonzeros in each column of L below diagonal, as given by QDLDL_etree (not modified)
etree	elimination tree as as given by QDLDL_etree (not modified)
bwork	working array of bools. Length is n
iwork	working array of integers. Length is 3∗n
fwork	working array of floats. Length is n

Returns

Returns a count of the number of positive elements in D. Returns -1 and exits immediately if any element of D evaluates exactly to zero (matrix is not quasidefinite or otherwise LDL factorisable)

Definition at line 72 of file qdldl.c.

4.169.1.3 QDLDL_Lsolve()

```
void QDLDL_Lsolve (
            const QDLDL_int n,
            const QDLDL_int * Lp,
            const QDLDL_int * Li,
            const QDLDL_float * Lx,
            QDLDL_float * x )
```

Solves (L+I)x = b

It is assumed that L will be a compressed sparse column matrix with data (n,Lp,Li,Lx).

Parameters

n	number of columns in L
Lp	column pointers (size n+1) for columns of L
Li	row indices of L. Has Lp[n] elements
Lx	data of L. Has Lp[n] elements
x	initialized to b. Equal to x on return

Definition at line 236 of file qdldl.c.

4.169.1.4 QDLDL_Ltsolve()

```
void QDLDL_Ltsolve (
            const QDLDL_int n,
            const QDLDL_int * Lp,
            const QDLDL_int * Li,
            const QDLDL_float * Lx,
            QDLDL_float * x )
```

Solves (L+I)'x = b

It is assumed that L will be a compressed sparse column matrix with data (n,Lp,Li,Lx).

Parameters

n	number of columns in L
Lp	column pointers (size n+1) for columns of L
Li	row indices of L. Has Lp[n] elements
Lx	data of L. Has Lp[n] elements
x	initialized to b. Equal to x on return

Definition at line 252 of file qdldl.c.

4.169.1.5 QDLDL_solve()

```
void QDLDL_solve (
            const QDLDL_int n,
            const QDLDL_int * Lp,
            const QDLDL_int * Li,
            const QDLDL_float * Lx,
            const QDLDL_float * Dinv,
            QDLDL_float * x )
```

Solves LDL'x = b

It is assumed that L will be a compressed sparse column matrix with data (n,Lp,Li,Lx).

Parameters

n	number of columns in L
Lp	column pointers (size n+1) for columns of L
Li	row indices of L. Has Lp[n] elements
Lx	data of L. Has Lp[n] elements
Dinv	reciprocal of D. Length is n
x	initialized to b. Equal to x on return

Definition at line 269 of file qdldl.c.

4.170 qdldl.h

Go to the documentation of this file.

```
#ifndef QDLDL_H
#define QDLDL_H

// Include qdldl type options
#include "qdldl_types.h"

# ifdef __cplusplus
extern "C" {
# endif // ifdef __cplusplus

 QDLDL_int QDLDL_etree(const QDLDL_int   n,
                       const QDLDL_int* Ap,
                       const QDLDL_int* Ai,
                       QDLDL_int* work,
                       QDLDL_int* Lnz,
                       QDLDL_int* etree);


QDLDL_int QDLDL_factor(const QDLDL_int    n,
                  const QDLDL_int*   Ap,
                  const QDLDL_int*   Ai,
                  const QDLDL_float* Ax,
                  QDLDL_int*    Lp,
                  QDLDL_int*    Li,
                  QDLDL_float* Lx,
                  QDLDL_float* D,
                  QDLDL_float* Dinv,
                  const QDLDL_int* Lnz,
                  const QDLDL_int* etree,
                  QDLDL_bool* bwork,
                  QDLDL_int* iwork,
                  QDLDL_float* fwork);


void QDLDL_solve(const QDLDL_int    n,
                 const QDLDL_int*   Lp,
                 const QDLDL_int*   Li,
```

```
                   const QDLDL_float* Lx,
                   const QDLDL_float* Dinv,
                   QDLDL_float* x);


void QDLDL_Lsolve(const QDLDL_int    n,
                  const QDLDL_int*   Lp,
                  const QDLDL_int*   Li,
                  const QDLDL_float* Lx,
                  QDLDL_float* x);


void QDLDL_Ltsolve(const QDLDL_int    n,
                   const QDLDL_int*   Lp,
                   const QDLDL_int*   Li,
                   const QDLDL_float* Lx,
                   QDLDL_float* x);

# ifdef __cplusplus
}
# endif // ifdef __cplusplus

#endif // ifndef QDLDL_H
```

4.171 qdldl/include/qdldl_types.h File Reference

```
#include <limits.h>
#include "glbopts.h"
```

Macros

- #define QDLDL_INT_MAX INT_MAX

Typedefs

- typedef abip_int QDLDL_int
- typedef abip_float QDLDL_float
- typedef abip_int QDLDL_bool

4.171.1 Macro Definition Documentation

4.171.1.1 QDLDL_INT_MAX

```
#define QDLDL_INT_MAX INT_MAX
```

Definition at line 18 of file qdldl_types.h.

4.171.2 Typedef Documentation

4.171.2.1 QDLDL_bool

```
typedef abip_int QDLDL_bool
```

Definition at line 15 of file qdldl_types.h.

4.171.2.2 QDLDL_float

```
typedef abip_float QDLDL_float
```

Definition at line 14 of file qdldl_types.h.

4.171.2.3 QDLDL_int

```
typedef abip_int QDLDL_int
```

Definition at line 13 of file qdldl_types.h.

4.172 qdldl_types.h

Go to the documentation of this file.

```
00001 #ifndef QDLDL_TYPES_H
00002 # define QDLDL_TYPES_H
00003
00004 # ifdef __cplusplus
00005 extern "C" {
00006 # endif /* ifdef __cplusplus */
00007
00008 #include <limits.h> //for the QDLDL_INT_TYPE_MAX
00009 #include"glbopts.h"
00010
00011 // QDLDL integer and float types
00012
00013 typedef abip_int   QDLDL_int;   /* for indices */
00014 typedef abip_float  QDLDL_float; /* for numerical values  */
00015 typedef abip_int   QDLDL_bool;  /* for boolean values  */
00016
00017 //Maximum value of the signed type QDLDL_int.
00018 #define QDLDL_INT_MAX INT_MAX
00019
00020 # ifdef __cplusplus
00021 }
00022 # endif /* ifdef __cplusplus */
00023
00024 #endif /* ifndef QDLDL_TYPES_H */
```

4.173 qdldl/src/qdldl.c File Reference

```
#include "qdldl.h"
```

Macros

- #define QDLDL_UNKNOWN (-1)
- #define QDLDL_USED (1)
- #define QDLDL_UNUSED (0)

Functions

- QDLDL_int QDLDL_etree (const QDLDL_int n, const QDLDL_int ∗Ap, const QDLDL_int ∗Ai, QDLDL_int ∗work, QDLDL_int ∗Lnz, QDLDL_int ∗etree)
- QDLDL_int QDLDL_factor (const QDLDL_int n, const QDLDL_int ∗Ap, const QDLDL_int ∗Ai, const QDLDL_float ∗Ax, QDLDL_int ∗Lp, QDLDL_int ∗Li, QDLDL_float ∗Lx, QDLDL_float ∗D, QDLDL_float ∗Dinv, const QDLDL_int ∗Lnz, const QDLDL_int ∗etree, QDLDL_bool ∗bwork, QDLDL_int ∗iwork, QDLDL_float ∗fwork)
- void QDLDL_Lsolve (const QDLDL_int n, const QDLDL_int ∗Lp, const QDLDL_int ∗Li, const QDLDL_float ∗Lx, QDLDL_float ∗x)
- void QDLDL_Ltsolve (const QDLDL_int n, const QDLDL_int ∗Lp, const QDLDL_int ∗Li, const QDLDL_float ∗Lx, QDLDL_float ∗x)
- void QDLDL_solve (const QDLDL_int n, const QDLDL_int ∗Lp, const QDLDL_int ∗Li, const QDLDL_float ∗Lx, const QDLDL_float ∗Dinv, QDLDL_float ∗x)

4.173.1 Macro Definition Documentation

4.173.1.1 QDLDL_UNKNOWN

```
#define QDLDL_UNKNOWN (-1)
```

Definition at line 3 of file qdldl.c.

4.173.1.2 QDLDL_UNUSED

```
#define QDLDL_UNUSED (0)
```

Definition at line 5 of file qdldl.c.

4.173.1.3 QDLDL_USED

```
#define QDLDL_USED (1)
```

Definition at line 4 of file qdldl.c.

4.173.2 Function Documentation

4.173.2.1 QDLDL_etree()

```
QDLDL_int QDLDL_etree (
            const QDLDL_int n,
            const QDLDL_int * Ap,
            const QDLDL_int * Ai,
            QDLDL_int * work,
            QDLDL_int * Lnz,
            QDLDL_int * etree )
```

Compute the elimination tree for a quasidefinite matrix in compressed sparse column form, where the input matrix is assumed to contain data for the upper triangular part of A only, and there are no duplicate indices.

Returns an elimination tree for the factorization A = LDL^T and a count of the nonzeros in each column of L that are strictly below the diagonal.

Does not use MALLOC. It is assumed that the arrays work, Lnz, and etree will be allocated with a number of elements equal to n.

The data in (n,Ap,Ai) are from a square matrix A in CSC format, and should include the upper triangular part of A only.

This function is only intended for factorisation of QD matrices specified by their upper triangular part. An error is returned if any column has data below the diagonal or s completely empty.

For matrices with a non-empty column but a zero on the corresponding diagonal, this function will *not* return an error, as it may still be possible to factor such a matrix in LDL form. No promises are made in this case though...

Parameters

n	number of columns in CSC matrix A (assumed square)
Ap	column pointers (size n+1) for columns of A
Ai	row indices of A. Has Ap[n] elements
work	work vector (size n) (no meaning on return)
Lnz	count of nonzeros in each column of L (size n) below diagonal
etree	elimination tree (size n)

Returns

total sum of Lnz (i.e. total nonzeros in L below diagonal). Returns -1 if the input is not triu or has an empty column. Returns -2 if the return value overflows QDLDL_int.

Definition at line 11 of file qdldl.c.

4.173.2.2 QDLDL_factor()

```
QDLDL_int QDLDL_factor (
            const QDLDL_int n,
            const QDLDL_int * Ap,
            const QDLDL_int * Ai,
            const QDLDL_float * Ax,
            QDLDL_int * Lp,
            QDLDL_int * Li,
            QDLDL_float * Lx,
            QDLDL_float * D,
            QDLDL_float * Dinv,
            const QDLDL_int * Lnz,
            const QDLDL_int * etree,
            QDLDL_bool * bwork,
            QDLDL_int * iwork,
            QDLDL_float * fwork )
```

Compute an LDL decomposition for a quasidefinite matrix in compressed sparse column form, where the input matrix is assumed to contain data for the upper triangular part of A only, and there are no duplicate indices.

Returns factors L, D and Dinv = 1./D.

Does not use MALLOC. It is assumed that L will be a compressed sparse column matrix with data (n,Lp,Li,Lx) with sufficient space allocated, with a number of nonzeros equal to the count given as a return value by QDLDL_etree

Parameters

n	number of columns in L and A (both square)
Ap	column pointers (size n+1) for columns of A (not modified)
Ai	row indices of A. Has Ap[n] elements (not modified)
Ax	data of A. Has Ap[n] elements (not modified)
Lp	column pointers (size n+1) for columns of L
Li	row indices of L. Has Lp[n] elements
Lx	data of L. Has Lp[n] elements
D	vectorized factor D. Length is n
Dinv	reciprocal of D. Length is n
Lnz	count of nonzeros in each column of L below diagonal, as given by QDLDL_etree (not modified)
etree	elimination tree as as given by QDLDL_etree (not modified)
bwork	working array of bools. Length is n
iwork	working array of integers. Length is 3∗n
fwork	working array of floats. Length is n

Returns

Returns a count of the number of positive elements in D. Returns -1 and exits immediately if any element of D evaluates exactly to zero (matrix is not quasidefinite or otherwise LDL factorisable)

Definition at line 72 of file qdldl.c.

4.173.2.3 QDLDL_Lsolve()

```
void QDLDL_Lsolve (
            const QDLDL_int n,
            const QDLDL_int * Lp,
            const QDLDL_int * Li,
            const QDLDL_float * Lx,
            QDLDL_float * x )
```

Solves (L+I)x = b

It is assumed that L will be a compressed sparse column matrix with data (n,Lp,Li,Lx).

Parameters

n	number of columns in L
Lp	column pointers (size n+1) for columns of L
Li	row indices of L. Has Lp[n] elements
Lx	data of L. Has Lp[n] elements
x	initialized to b. Equal to x on return

Definition at line 236 of file qdldl.c.

4.173.2.4 QDLDL_Ltsolve()

```
void QDLDL_Ltsolve (
            const QDLDL_int n,
            const QDLDL_int * Lp,
            const QDLDL_int * Li,
            const QDLDL_float * Lx,
            QDLDL_float * x )
```

Solves (L+I)'x = b

It is assumed that L will be a compressed sparse column matrix with data (n,Lp,Li,Lx).

Parameters

n	number of columns in L
Lp	column pointers (size n+1) for columns of L
Li	row indices of L. Has Lp[n] elements
Lx	data of L. Has Lp[n] elements
x	initialized to b. Equal to x on return

Definition at line 252 of file qdldl.c.

4.173.2.5 QDLDL_solve()

```
void QDLDL_solve (
        const QDLDL_int n,
        const QDLDL_int * Lp,
        const QDLDL_int * Li,
        const QDLDL_float * Lx,
        const QDLDL_float * Dinv,
        QDLDL_float * x )
```

Solves LDL'x = b

It is assumed that L will be a compressed sparse column matrix with data (n,Lp,Li,Lx).

Parameters

n	number of columns in L
Lp	column pointers (size n+1) for columns of L
Li	row indices of L. Has Lp[n] elements
Lx	data of L. Has Lp[n] elements
Dinv	reciprocal of D. Length is n
x	initialized to b. Equal to x on return

Definition at line 269 of file qdldl.c.

4.174 qdldl.c

Go to the documentation of this file.

```
#include "qdldl.h"

#define QDLDL_UNKNOWN (-1)
#define QDLDL_USED (1)
#define QDLDL_UNUSED (0)

/* Compute the elimination tree for a quasidefinite matrix
   in compressed sparse column form.
*/

QDLDL_int QDLDL_etree(const QDLDL_int  n,
                      const QDLDL_int* Ap,
                      const QDLDL_int* Ai,
                      QDLDL_int* work,
                      QDLDL_int* Lnz,
                      QDLDL_int* etree){

  QDLDL_int sumLnz;
  QDLDL_int i,j,p;


  for(i = 0; i < n; i++){
  // zero out Lnz and work.  Set all etree values to unknown
    work[i]  = 0;
    Lnz[i]   = 0;
    etree[i] = QDLDL_UNKNOWN;

    //Abort if A doesn't have at least one entry
    //one entry in every column
    if(Ap[i] == Ap[i+1]){
      return -1;
    }

  }

  for(j = 0; j < n; j++){
    work[j] = j;
```

```
    for(p = Ap[j]; p < Ap[j+1]; p++){
      i = Ai[p];
      if(i > j){return -1;}; //abort if entries on lower triangle
      while(work[i] != j){
        if(etree[i] == QDLDL_UNKNOWN){
          etree[i] = j;
        }
        Lnz[i]++;        //nonzeros in this column
        work[i] = j;
        i = etree[i];
      }
    }
  }

  //compute the total nonzeros in L.  This much
  //space is required to store Li and Lx.  Return
  //error code -2 if the nonzero count will overflow
  //its unteger type.
  sumLnz  = 0;
  for(i = 0; i < n; i++){
    if(sumLnz > QDLDL_INT_MAX - Lnz[i]){
      sumLnz = -2;
      break;
    }
    else{
      sumLnz += Lnz[i];
    }
  }

  return sumLnz;
}



QDLDL_int QDLDL_factor(const QDLDL_int    n,
                  const QDLDL_int*   Ap,
                  const QDLDL_int*   Ai,
                  const QDLDL_float* Ax,
                  QDLDL_int*   Lp,
                  QDLDL_int*   Li,
                  QDLDL_float* Lx,
                  QDLDL_float* D,
                  QDLDL_float* Dinv,
                  const QDLDL_int* Lnz,
                  const QDLDL_int* etree,
                  QDLDL_bool*  bwork,
                  QDLDL_int*   iwork,
                  QDLDL_float* fwork){

  QDLDL_int i,j,k,nnzY, bidx, cidx, nextIdx, nnzE, tmpIdx;
  QDLDL_int *yIdx, *elimBuffer, *LNextSpaceInCol;
  QDLDL_float *yVals;
  QDLDL_float yVals_cidx;
  QDLDL_bool  *yMarkers;
  QDLDL_int   positiveValuesInD = 0;

  //partition working memory into pieces
  yMarkers        = bwork;
  yIdx            = iwork;
  elimBuffer      = iwork + n;
  LNextSpaceInCol = iwork + n*2;
  yVals           = fwork;


  Lp[0] = 0; //first column starts at index zero

  for(i = 0; i < n; i++){

    //compute L column indices
    Lp[i+1] = Lp[i] + Lnz[i];   //cumsum, total at the end

    // set all Yidx to be 'unused' initially
    //in each column of L, the next available space
    //to start is just the first space in the column
    yMarkers[i]  = QDLDL_UNUSED;
    yVals[i]     = 0.0;
    D[i]         = 0.0;
    LNextSpaceInCol[i] = Lp[i];
  }

  // First element of the diagonal D.
  D[0]     = Ax[0];
  if(D[0] == 0.0){return -1;}
  if(D[0]  > 0.0){positiveValuesInD++;}
  Dinv[0] = 1/D[0];

  //Start from 1 here. The upper LH corner is trivially 0
```

```
    //in L b/c we are only computing the subdiagonal elements
    for(k = 1; k < n; k++){

      //NB : For each k, we compute a solution to
      //y = L(0:(k-1),0:k-1))\b, where b is the kth
      //column of A that sits above the diagonal.
      //The solution y is then the kth row of L,
      //with an implied '1' at the diagonal entry.

      //number of nonzeros in this row of L
      nnzY = 0;  //number of elements in this row

      //This loop determines where nonzeros
      //will go in the kth row of L, but doesn't
      //compute the actual values
      tmpIdx = Ap[k+1];

      for(i = Ap[k]; i < tmpIdx; i++){

        bidx = Ai[i];    // we are working on this element of b

        //Initialize D[k] as the element of this column
        //corresponding to the diagonal place.  Don't use
        //this element as part of the elimination step
        //that computes the k^th row of L
        if(bidx == k){
          D[k] = Ax[i];
          continue;
        }

        yVals[bidx] = Ax[i];    // initialise y(bidx) = b(bidx)

        // use the forward elimination tree to figure
        // out which elements must be eliminated after
        // this element of b
        nextIdx = bidx;

        if(yMarkers[nextIdx] == QDLDL_UNUSED){   //this y term not already visited

          yMarkers[nextIdx] = QDLDL_USED;     //I touched this one
          elimBuffer[0]     = nextIdx;  // It goes at the start of the current list
          nnzE              = 1;         //length of unvisited elimination path from here

          nextIdx = etree[bidx];

          while(nextIdx != QDLDL_UNKNOWN && nextIdx < k){
            if(yMarkers[nextIdx] == QDLDL_USED) break;

            yMarkers[nextIdx] = QDLDL_USED;   //I touched this one
            elimBuffer[nnzE] = nextIdx; //It goes in the current list
            nnzE++;                     //the list is one longer than before
            nextIdx = etree[nextIdx];   //one step further along tree

          } //end while

          // now I put the buffered elimination list into
          // my current ordering in reverse order
          while(nnzE){
            yIdx[nnzY++] = elimBuffer[--nnzE];
          } //end while
        } //end if

      } //end for i

      //This for loop places nonzeros values in the k^th row
      for(i = (nnzY-1); i >=0; i--){

        //which column are we working on?
        cidx = yIdx[i];

        // loop along the elements in this
        // column of L and subtract to solve to y
        tmpIdx = LNextSpaceInCol[cidx];
        yVals_cidx = yVals[cidx];
        for(j = Lp[cidx]; j < tmpIdx; j++){
          yVals[Li[j]] -= Lx[j]*yVals_cidx;
        }

        //Now I have the cidx^th element of y = L\b.
        //so compute the corresponding element of
        //this row of L and put it into the right place
        Li[tmpIdx] = k;
        Lx[tmpIdx] = yVals_cidx *Dinv[cidx];

        //D[k] -= yVals[cidx]*yVals[cidx]*Dinv[cidx];
        D[k] -= yVals_cidx*Lx[tmpIdx];
        LNextSpaceInCol[cidx]++;
```

```

      //reset the yvalues and indices back to zero and QDLDL_UNUSED
      //once I'm done with them
      yVals[cidx]      = 0.0;
      yMarkers[cidx]   = QDLDL_UNUSED;

    } //end for i

    //Maintain a count of the positive entries
    //in D.  If we hit a zero, we can't factor
    //this matrix, so abort
    if(D[k] == 0.0){return -1;}
    if(D[k]  > 0.0){positiveValuesInD++;}

    //compute the inverse of the diagonal
    Dinv[k]= 1/D[k];

  } //end for k

  return positiveValuesInD;

}

// Solves (L+I)x = b
void QDLDL_Lsolve(const QDLDL_int    n,
                  const QDLDL_int*   Lp,
                  const QDLDL_int*   Li,
                  const QDLDL_float* Lx,
                  QDLDL_float* x){

  QDLDL_int i,j;
  for(i = 0; i < n; i++){
    QDLDL_float val = x[i];
    for(j = Lp[i]; j < Lp[i+1]; j++){
      x[Li[j]] -= Lx[j]*val;
    }
  }
}

// Solves (L+I)'x = b
void QDLDL_Ltsolve(const QDLDL_int    n,
                   const QDLDL_int*   Lp,
                   const QDLDL_int*   Li,
                   const QDLDL_float* Lx,
                   QDLDL_float* x){

  QDLDL_int i,j;
  for(i = n-1; i>=0; i--){
    QDLDL_float val = x[i];
    for(j = Lp[i]; j < Lp[i+1]; j++){
      val -= Lx[j]*x[Li[j]];
    }
    x[i] = val;
  }
}

// Solves Ax = b where A has given LDL factors
void QDLDL_solve(const QDLDL_int       n,
                    const QDLDL_int*   Lp,
                    const QDLDL_int*   Li,
                    const QDLDL_float* Lx,
                    const QDLDL_float* Dinv,
                    QDLDL_float* x){

  QDLDL_int i;

  QDLDL_Lsolve(n,Lp,Li,Lx,x);
  for(i = 0; i < n; i++) x[i] *= Dinv[i];
  QDLDL_Ltsolve(n,Lp,Li,Lx,x);
}
```

4.175 source/abip.c File Reference

```
#include "abip.h"
#include <stdio.h>
#include "cones.h"
#include "cs.h"
#include "ctrlc.h"
#include "glbopts.h"
```

```
#include "lasso_config.h"
#include "linalg.h"
#include "linsys.h"
#include "mkl.h"
#include "mkl_lapacke.h"
#include "qcp_config.h"
#include "svm_config.h"
#include "svm_qp_config.h"
#include "util.h"
```

Macros

- #define _CRT_SECURE_NO_WARNINGS

Functions

- ABIP (timer)
- abip_int update_work (const ABIPData ∗d, ABIPWork ∗w, const ABIPSolution ∗sol, const ABIPCone ∗c, spe_problem ∗spe)
- abip_float adjust_barrier (ABIPWork ∗w, ABIPResiduals ∗r, spe_problem ∗spe)
- abip_int ABIP() solve (ABIPWork ∗w, const ABIPData ∗d, ABIPSolution ∗sol, ABIPInfo ∗info, ABIPCone ∗c, spe_problem ∗s)

  Main solve iteration of the solver.
- void ABIP() finish (ABIPWork ∗w, spe_problem ∗spe)

  Free the memory allocated for the solver.
- ABIPWork ∗ABIP() init (const ABIPData ∗d, ABIPInfo ∗info, spe_problem ∗s, ABIPCone ∗c)

  Initialize and allocate memory for the solver.
- abip_int ABIP() init_problem (spe_problem ∗∗s, ABIPData ∗d, ABIPSettings ∗stgs, enum problem_type special_problem)

  Specify the problem type and initialize the problem.
- abip_int abip (const ABIPData ∗d, ABIPSolution ∗sol, ABIPInfo ∗info, ABIPCone ∗K)

  the main function to call the abip conic solver

4.175.1 Macro Definition Documentation

4.175.1.1 _CRT_SECURE_NO_WARNINGS

```
#define _CRT_SECURE_NO_WARNINGS
```

Definition at line 1 of file abip.c.

4.175.2 Function Documentation

4.175.2.1 abip()

```
abip_int abip (
            const ABIPData * d,
            ABIPSolution * sol,
            ABIPInfo * info,
            ABIPCone * K )
```

the main function to call the abip conic solver

Definition at line 1335 of file abip.c.

4.175.2.2 ABIP()

```
ABIP (
            timer )
```

Definition at line 20 of file abip.c.

4.175.2.3 adjust_barrier()

```
abip_float adjust_barrier (
            ABIPWork * w,
            ABIPResiduals * r,
            spe_problem * spe )
```

Definition at line 994 of file abip.c.

4.175.2.4 finish()

```
void ABIP() finish (
            ABIPWork * w,
            spe_problem * spe )
```

Free the memory allocated for the solver.

Definition at line 1254 of file abip.c.

4.175.2.5 init()

```
ABIPWork *ABIP() init (
             const ABIPData * d,
             ABIPInfo * info,
             spe_problem * s,
             ABIPCone * c )
```

Initialize and allocate memory for the solver.

Definition at line 1271 of file abip.c.

4.175.2.6 init_problem()

```
abip_int ABIP() init_problem (
             spe_problem ** s,
             ABIPData * d,
             ABIPSettings * stgs,
             enum problem_type special_problem )
```

Specify the problem type and initialize the problem.

Definition at line 1316 of file abip.c.

4.175.2.7 solve()

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

Main solve iteration of the solver.

Definition at line 1076 of file abip.c.

4.175.2.8 update_work()

```
abip_int update_work (
             const ABIPData * d,
             ABIPWork * w,
             const ABIPSolution * sol,
             const ABIPCone * c,
             spe_problem * spe )
```

Definition at line 912 of file abip.c.

4.176 abip.c

Go to the documentation of this file.

```
#define _CRT_SECURE_NO_WARNINGS
#include "abip.h"

#include <stdio.h>

#include "cones.h"
#include "cs.h"
#include "ctrlc.h"
#include "glbopts.h"
#include "lasso_config.h"
#include "linalg.h"
#include "linsys.h"
#include "mkl.h"
#include "mkl_lapacke.h"
#include "qcp_config.h"
#include "svm_config.h"
#include "svm_qp_config.h"
#include "util.h"

ABIP(timer) global_timer;

/* printing header */
static const char *HEADER[] = {
    " ipm iter ", " admm iter ", "     mu ",  " pri res ", " dua res ",
    " rel gap ",  " pri obj ",   " dua obj ", " kap/tau ", " time (s)",
};

static const abip_int HSPACE = 9;
static const abip_int HEADER_LEN = 10;
static const abip_int LINE_LEN = 150;

static abip_int abip_isnan(abip_float x) {
  DEBUG_FUNC
  RETURN(x == NAN || x != x);
}

static void free_work(ABIPWork *w) {
  DEBUG_FUNC

  if (!w) {
    RETURN;
  }

  if (w->u) {
    abip_free(w->u);
  }

  if (w->v) {
    abip_free(w->v);
  }

  if (w->u_t) {
    abip_free(w->u_t);
  }

  if (w->rel_ut) {
    abip_free(w->rel_ut);
  }

  if (w->A) {
    ABIP(free_A_matrix)(w->A);
  }

  abip_free(w);

  RETURN;
}

static void print_init_header(spe_problem *spe) {
  DEBUG_FUNC

  abip_int i;
  ABIPSettings *stgs = spe->stgs;
  char *lin_sys_method = ABIP(get_lin_sys_method)(spe);

  for (i = 0; i < LINE_LEN; ++i) {
    abip_printf("-");
  }

  abip_printf(
      "\n\tABIP v%s - First-Order Interior-Point Solver for Conic "
      "Programming\n\t(c)  Jinsong Liu & LEAVES Group, 2021-2024\n",
```

```
      ABIP(version)());

  for (i = 0; i < LINE_LEN; ++i) {
    abip_printf("-");
  }

  abip_printf("\n");

  if (lin_sys_method) {
    abip_printf("Lin-sys: %s\n", lin_sys_method);
    abip_free(lin_sys_method);
  }

  if (stgs->normalize) {
    abip_printf(
        "eps_p = %.2e, eps_d = %.2e, eps_g = %.2e, alpha = %.2f, max_ipm_iters "
        "= %i, max_admm_iters = %i, normalize = %i\n"
        "rho_y = %.2e\n",
        stgs->eps_p, stgs->eps_d, stgs->eps_g, stgs->alpha,
        (int)stgs->max_ipm_iters, (int)stgs->max_admm_iters,
        (int)stgs->normalize, stgs->rho_y);
  } else {
    abip_printf(
        "eps_p = %.2e, eps_d = %.2e, eps_g = %.2e, alpha = %.2f, max_ipm_iters "
        "= %i, max_admm_iters = %i, normalize = %i\n"
        "rho_y = %.2e\n",
        stgs->eps_p, stgs->eps_d, stgs->eps_g, stgs->alpha,
        (int)stgs->max_ipm_iters, (int)stgs->max_admm_iters,
        (int)stgs->normalize, stgs->rho_y);
  }

  abip_printf("constraints m = %i, Variables n = %i\n", (int)spe->p,
              (int)spe->q);

#ifdef MATLAB_MEX_FILE

  mexEvalString("drawnow;");

#endif

  RETURN;
}

static void populate_on_failure(abip_int m, abip_int n, ABIPSolution *sol,
                                ABIPInfo *info, abip_int status_val,
                                const char *msg) {
  DEBUG_FUNC

  if (info) {
    info->res_pri = NAN;
    info->res_dual = NAN;
    info->rel_gap = NAN;
    info->res_infeas = NAN;
    info->res_unbdd = NAN;

    info->pobj = NAN;
    info->dobj = NAN;

    info->ipm_iter = -1;
    info->admm_iter = -1;
    info->status_val = status_val;
    info->solve_time = NAN;
    strcpy(info->status, msg);
  }

  if (sol) {
    if (n > 0) {
      if (!sol->x) {
        sol->x = (abip_float *)abip_malloc(sizeof(abip_float) * n);
      }
      ABIP(scale_array)(sol->x, NAN, n);

      if (!sol->s) {
        sol->s = (abip_float *)abip_malloc(sizeof(abip_float) * n);
      }
      ABIP(scale_array)(sol->s, NAN, m);
    }

    if (m > 0) {
      if (!sol->y) {
        sol->y = (abip_float *)abip_malloc(sizeof(abip_float) * m);
      }
      ABIP(scale_array)(sol->y, NAN, m);
    }
  }

  RETURN;
```

```
}

static abip_int failure(ABIPWork *w, abip_int m, abip_int n, ABIPSolution *sol,
                        ABIPInfo *info, abip_int stint, const char *msg,
                        const char *ststr) {
  DEBUG_FUNC

  abip_int status = stint;
  populate_on_failure(m, n, sol, info, status, ststr);

  abip_printf("Failure:%s\n", msg);
  abip_end_interrupt_listener();

  RETURN status;
}

static abip_int projection(ABIPWork *w, abip_int iter, spe_problem *spe,
                           ABIPResiduals *r) {
  DEBUG_FUNC

  abip_int status;

  if (spe->Q != ABIP_NULL || spe->stgs->linsys_solver != 3) {
    abip_int n = w->n;
    abip_int m = w->m;
    abip_int l = n + m + 1;
    abip_float a = w->a;

    abip_float *mu = (abip_float *)abip_malloc((m + n) * sizeof(abip_float));
    memcpy(mu, w->u, (m + n) * sizeof(abip_float));
    ABIP(add_scaled_array)(mu, w->v, m + n, 1);
    ABIP(c_dot)(mu, spe->rho_dr, m + n);
    abip_float eta = spe->rho_dr[m + n] * (w->u[m + n] + w->v[m + n]);

    abip_float *warm_start =
        (abip_float *)abip_malloc((m + n) * sizeof(abip_float));
    memcpy(warm_start, w->u, (m + n) * sizeof(abip_float));

    abip_float *tem = (abip_float *)abip_malloc((m + n) * sizeof(abip_float));

    abip_float pcg_tol = 0;
    if (spe->stgs->linsys_solver == 3) {  // create warm start for pcg
      ABIP(add_scaled_array)(warm_start, w->r, m + n, w->u[m + n]);

      pcg_tol = MIN(r->Ax_b_norm, r->Qx_ATy_c_s_norm);
      pcg_tol = 0.2 * MIN(pcg_tol, ABIP(norm_inf)(warm_start, n) /
                                    POWF((abip_float)iter + 1, 1.5));

      pcg_tol = MAX(pcg_tol, 1e-12);
    }

    abip_float *p = (abip_float *)abip_malloc((m + n) * sizeof(abip_float));
    memcpy(p, mu, (m + n) * sizeof(abip_float));

    status = spe->solve_spe_linsys(spe, p, warm_start, iter, pcg_tol);

    memcpy(tem, p, (m + n) * sizeof(abip_float));
    ABIP(c_dot)(tem, spe->rho_dr, m + n);

    abip_float b =
        ABIP(dot)(w->r, mu, m + n) - 2 * ABIP(dot)(w->r, tem, m + n) - eta;

    abip_float *Qp = (abip_float *)abip_malloc(n * sizeof(abip_float));
    memset(Qp, 0, n * sizeof(abip_float));

    if (spe->Q != ABIP_NULL) {
      ABIP(accum_by_A)(spe->Q, &p[m], Qp);
    }

    abip_float c = -ABIP(dot)(&p[m], Qp, n);

    if (iter > 0) {
      w->u_t[m + n] = (-b + SQRTF(MAX(0, b * b - 4 * a * c))) / (2 * a);
    } else {
      w->u_t[m + n] = 1;
    }

    memcpy(w->u_t, p, (m + n) * sizeof(abip_float));
    ABIP(add_scaled_array)(w->u_t, w->r, m + n, -w->u_t[m + n]);

    abip_free(mu);
    abip_free(warm_start);
    abip_free(p);
    abip_free(tem);
    abip_free(Qp);
  } else {
    abip_int n = w->n;
```

```
    abip_int m = w->m;
    abip_int l = n + m + 1;
    abip_float a = w->a;

    abip_float *mu = (abip_float *)abip_malloc((m + n) * sizeof(abip_float));
    memcpy(mu, w->u, (m + n) * sizeof(abip_float));
    ABIP(add_scaled_array)(mu, w->v, m + n, 1);
    ABIP(c_dot)(mu, spe->rho_dr, m + n);

    abip_float eta = spe->rho_dr[m + n] * (w->u[m + n] + w->v[m + n]);

    abip_float *w3 = (abip_float *)abip_malloc((m + n) * sizeof(abip_float));
    memcpy(w3, mu, (m + n) * sizeof(abip_float));
    ABIP(add_scaled_array)(w3, spe->b, m, eta);
    ABIP(add_scaled_array)(&w3[m], spe->c, n, -eta);

    abip_float *warm_start =
        (abip_float *)abip_malloc((m + n) * sizeof(abip_float));
    memcpy(warm_start, w->u, (m + n) * sizeof(abip_float));

    abip_float pcg_tol = 0;
    if (spe->stgs->linsys_solver == 3) {  // create warm start for pcg

      ABIP(add_scaled_array)(warm_start, w->r, m + n, w->u[m + n]);

      pcg_tol = MIN(r->Ax_b_norm, r->Qx_ATy_c_s_norm);
      pcg_tol = 0.2 * MIN(pcg_tol, ABIP(norm_inf)(warm_start, n) /
                                    POWF((abip_float)iter + 1, 1.5));

      pcg_tol = MAX(pcg_tol, 1e-12);
    }

    memset(warm_start, 0, (m + n) * sizeof(abip_float));

    status = spe->solve_spe_linsys(spe, w3, warm_start, iter, r->error_ratio);

    abip_float coef =
        -(-ABIP(dot)(spe->b, w3, m) + ABIP(dot)(spe->c, &w3[m], n)) /
        (1 - ABIP(dot)(spe->b, w->r, m) + ABIP(dot)(spe->c, &w->r[m], n));

    memcpy(w->u_t, w3, (m + n) * sizeof(abip_float));
    ABIP(add_scaled_array)(w->u_t, w->r, m + n, coef);
    if (iter > 0) {
      w->u_t[m + n] =
          eta - ABIP(dot)(spe->b, w->u_t, m) + ABIP(dot)(spe->c, &w->u_t[m], n);
    } else {
      w->u_t[m + n] = 1;
    }

    abip_free(mu);
    abip_free(w3);
    abip_free(warm_start);
  }

  RETURN status;
}

static void update_dual_vars(ABIPWork *w) {
  DEBUG_FUNC

  abip_int l = w->m + w->n + 1;

  memcpy(w->v, w->u, l * sizeof(abip_float));

  ABIP(add_scaled_array)(w->v, w->rel_ut, l, -1);

  RETURN;
}

static void solve_barrier_subproblem(ABIPWork *w, const ABIPCone *c,
                                     spe_problem *spe) {
  DEBUG_FUNC

  abip_int n = w->n;
  abip_int m = w->m;
  abip_int l = m + n + 1;
  abip_float lambda = w->mu / w->beta;

  // over relaxation: rel_ut = alpha * ut + (1 - alpha) * u
  memcpy(w->rel_ut, w->u_t, l * sizeof(abip_float));
  ABIP(scale_array)(w->rel_ut, spe->stgs->alpha, l);
  ABIP(add_scaled_array)(w->rel_ut, w->u, l, 1 - spe->stgs->alpha);

  abip_float *tmp = (abip_float *)abip_malloc(l * sizeof(abip_float));

  ABIP(add_scaled_array)(w->rel_ut, w->v, l, -1);
  memcpy(tmp, w->rel_ut, l * sizeof(abip_float));
```

```
  abip_float *y = (abip_float *)abip_malloc(m * sizeof(abip_float));
  memcpy(y, tmp, m * sizeof(abip_float));

  abip_float tau = (tmp[l - 1] + SQRTF(tmp[l - 1] * tmp[l - 1] +
                                        4 * lambda / spe->rho_dr[l - 1])) /
                    2;

  memcpy(w->u, y, m * sizeof(abip_float));
  w->u[l - 1] = tau;

  abip_int count = 0;
  abip_int i;

  /*soc*/
  if (c->qsize && c->q) {
    for (i = 0; i < c->qsize; ++i) {
      if (c->q[i] == 0) {
        continue;
      }
      if (c->q[i] == 1) {
        ABIP(positive_orthant_barrier_subproblem)
        (&w->u[m + count], &tmp[m + count], lambda / spe->rho_dr[m + count], 1);
      } else {
        ABIP(soc_barrier_subproblem)
        (&w->u[m + count], &tmp[m + count], lambda / spe->rho_dr[m + count],
         c->q[i]);
      }
      count += c->q[i];
    }
  }

  /*rsoc*/
  if (c->rqsize && c->rq) {
    for (i = 0; i < c->rqsize; ++i) {
      if (c->rq[i] < 3) {
        continue;
      } else {
        ABIP(rsoc_barrier_subproblem)
        (&w->u[m + count], &tmp[m + count], lambda / spe->rho_dr[m + count],
         c->rq[i]);
      }
      count += c->rq[i];
    }
  }

  /*free cone*/
  if (c->f) {
    ABIP(free_barrier_subproblem)
    (&w->u[m + count], &tmp[m + count], lambda / spe->rho_dr[m + count], c->f);
    count += c->f;
  }

  /*zero cone*/
  if (c->z) {
    ABIP(zero_barrier_subproblem)
    (&w->u[m + count], &tmp[m + count], lambda / spe->rho_dr[m + count], c->z);
    count += c->z;
  }

  /*positive orthant*/
  if (c->l) {
    ABIP(positive_orthant_barrier_subproblem)
    (&w->u[m + count], &tmp[m + count], lambda / spe->rho_dr[m + count], c->l);
    count += c->l;
  }

  abip_free(tmp);
  abip_free(y);
}

static abip_int indeterminate(ABIPWork *w, ABIPSolution *sol, ABIPInfo *info) {
  DEBUG_FUNC

  strcpy(info->status, "Indeterminate");

  ABIP(scale_array)(sol->x, NAN, w->n);
  ABIP(scale_array)(sol->y, NAN, w->m);
  ABIP(scale_array)(sol->s, NAN, w->n);

  RETURN ABIP_INDETERMINATE;
}

static abip_int solved(ABIPWork *w, ABIPSolution *sol, ABIPInfo *info,
                       abip_float tau) {
  DEBUG_FUNC

```

```
  ABIP(scale_array)(sol->x, SAFEDIV_POS(1.0, tau), w->n);
  ABIP(scale_array)(sol->y, SAFEDIV_POS(1.0, tau), w->m);
  ABIP(scale_array)(sol->s, SAFEDIV_POS(1.0, tau), w->n);

  if ((info->status_val == 0) || (info->status_val == 2)) {
    strcpy(info->status, "Solved/Inaccurate");
    RETURN ABIP_SOLVED_INACCURATE;
  }

  strcpy(info->status, "Solved");

  RETURN ABIP_SOLVED;
}
static void sety(ABIPWork *w, ABIPSolution *sol) {
  DEBUG_FUNC

  if (!sol->y) {
    sol->y = (abip_float *)abip_malloc(sizeof(abip_float) * w->m);
  }

  memcpy(sol->y, w->u, w->m * sizeof(abip_float));

  RETURN;
}

static void setx(ABIPWork *w, ABIPSolution *sol) {
  DEBUG_FUNC

  if (!sol->x) {
    sol->x = (abip_float *)abip_malloc(sizeof(abip_float) * w->n);
  }

  memcpy(sol->x, &(w->u[w->m]), w->n * sizeof(abip_float));

  RETURN;
}

static void sets(ABIPWork *w, ABIPSolution *sol) {
  DEBUG_FUNC

  if (!sol->s) {
    sol->s = (abip_float *)abip_malloc(sizeof(abip_float) * w->n);
  }

  memcpy(sol->s, &(w->v[w->m]), w->n * sizeof(abip_float));

  RETURN;
}

static abip_int infeasible(ABIPWork *w, ABIPSolution *sol, ABIPInfo *info,
                           abip_float bt_y) {
  DEBUG_FUNC

  ABIP(scale_array)(sol->y, 1 / bt_y, w->m);
  ABIP(scale_array)(sol->s, 1 / bt_y, w->n);
  ABIP(scale_array)(sol->x, NAN, w->n);

  if (info->status_val == 0) {
    strcpy(info->status, "Infeasible/Inaccurate");
    RETURN ABIP_INFEASIBLE_INACCURATE;
  }

  strcpy(info->status, "Infeasible");
  RETURN ABIP_INFEASIBLE;
}

static abip_int unbounded(ABIPWork *w, ABIPSolution *sol, ABIPInfo *info,
                          abip_float ct_x) {
  DEBUG_FUNC

  ABIP(scale_array)(sol->x, -1 / ct_x, w->n);
  ABIP(scale_array)(sol->y, NAN, w->m);
  ABIP(scale_array)(sol->s, NAN, w->n);

  if (info->status_val == 0) {
    strcpy(info->status, "Unbounded/Inaccurate");
    RETURN ABIP_UNBOUNDED_INACCURATE;
  }

  strcpy(info->status, "Unbounded");
  RETURN ABIP_UNBOUNDED;
}

static abip_int is_solved_status(abip_int status) {
  RETURN status == ABIP_SOLVED || status == ABIP_SOLVED_INACCURATE;
}

```

```
static abip_int is_infeasible_status(abip_int status) {
  RETURN status == ABIP_INFEASIBLE || status == ABIP_INFEASIBLE_INACCURATE;
}

static abip_int is_unbounded_status(abip_int status) {
  RETURN status == ABIP_UNBOUNDED || status == ABIP_UNBOUNDED_INACCURATE;
}

static void get_info(ABIPWork *w, ABIPSolution *sol, ABIPInfo *info,
                     ABIPResiduals *r, abip_int ipm_iter, abip_int admm_iter) {
  DEBUG_FUNC

  info->ipm_iter = ipm_iter + 1;
  info->admm_iter = admm_iter;

  info->res_infeas = r->res_infeas;
  info->res_unbdd = r->res_unbdd;

  if (is_solved_status(info->status_val)) {
    info->rel_gap = r->rel_gap;
    info->res_pri = r->res_pri;
    info->res_dual = r->res_dual;
    info->pobj = r->pobj;
    info->dobj = r->dobj;
  } else if (is_unbounded_status(info->status_val)) {
    info->rel_gap = NAN;
    info->res_pri = NAN;
    info->res_dual = NAN;
    info->pobj = -INFINITY;
    info->dobj = -INFINITY;
  } else if (is_infeasible_status(info->status_val)) {
    info->rel_gap = NAN;
    info->res_pri = NAN;
    info->res_dual = NAN;
    info->pobj = INFINITY;
    info->dobj = INFINITY;
  }

  RETURN;
}

static void get_solution(ABIPWork *w, spe_problem *spe, ABIPSolution *sol,
                         ABIPInfo *info, ABIPResiduals *r, abip_int ipm_iter,
                         abip_int admm_iter) {
  DEBUG_FUNC

  abip_int l = w->m + w->n + 1;

  setx(w, sol);
  sety(w, sol);
  sets(w, sol);

  if (info->status_val == ABIP_UNFINISHED) {
    info->status_val = solved(w, sol, info, r->tau);
  } else if (is_solved_status(info->status_val)) {
    info->status_val = solved(w, sol, info, r->tau);
  } else if (is_infeasible_status(info->status_val)) {
    info->status_val = infeasible(w, sol, info, r->dobj * r->tau);
  } else {
    info->status_val = unbounded(w, sol, info, r->pobj * r->tau);
  }

  if (spe->stgs->normalize) {
    spe->un_scaling_sol(spe, sol);
  }

  get_info(w, sol, info, r, ipm_iter, admm_iter);

  RETURN;
}

static void print_summary(ABIPWork *w, spe_problem *spe, abip_int i, abip_int j,
                          abip_int k, ABIPResiduals *r,
                          ABIP(timer) * solve_timer) {
  DEBUG_FUNC

  if (!spe->stgs->verbose) {
    RETURN;
  }

  abip_printf("%*i|", (int)strlen(HEADER[0]), (int)i + 1);
  abip_printf("%*i|", (int)strlen(HEADER[1]), (int)j + 1);
  abip_printf("%*.2e|", (int)strlen(HEADER[2]), w->mu);

  abip_printf("%*.2e|", (int)HSPACE, r->res_pri / spe->stgs->eps_p);
  abip_printf("%*.2e|", (int)HSPACE, r->res_dual / spe->stgs->eps_d);
  abip_printf("%*.2e|", (int)HSPACE, r->rel_gap / spe->stgs->eps_g);
```

```
  abip_printf("%*.2e|", (int)HSPACE, r->pobj);
  abip_printf("%*.2e|", (int)HSPACE, r->dobj);
  abip_printf("%*.2e|", (int)HSPACE, r->tau);

  abip_printf("%*.2e ", (int)HSPACE, ABIP(tocq)(solve_timer) / 1e3);
  abip_printf("\n");

#if EXTRA_VERBOSE > 0

  abip_printf("Norm u = %4f, ", ABIP(norm)(w->u, w->n + w->m + 2));
  abip_printf("Norm u_t = %4f, ", ABIP(norm)(w->u_t, w->n + w->m + 2));
  abip_printf("Norm v = %4f, ", ABIP(norm)(w->v, w->n + w->m + 2));
  abip_printf("tau = %4f, ", r->tau);
  abip_printf("kappa = %4f, ", r->kap);
  abip_printf("|u - u_t| = %1.2e, ",
              ABIP(norm_diff)(w->u, w->u_t, w->n + w->m + 2));
  abip_printf("res_infeas = %1.2e, ", r->res_infeas);
  abip_printf("res_unbdd = %1.2e\n", r->res_unbdd);

#endif

#ifdef MATLAB_MEX_FILE

  mexEvalString("drawnow;");

#endif

  RETURN;
}

static void print_header(ABIPWork *w) {
  DEBUG_FUNC

  abip_int i;

  for (i = 0; i < LINE_LEN; ++i) {
    abip_printf("-");
  }
  abip_printf("\n");

  for (i = 0; i < HEADER_LEN - 1; ++i) {
    abip_printf("%s|", HEADER[i]);
  }
  abip_printf("%s\n", HEADER[HEADER_LEN - 1]);

  for (i = 0; i < LINE_LEN; ++i) {
    abip_printf("-");
  }
  abip_printf("\n");

#ifdef MATLAB_MEX_FILE

  mexEvalString("drawnow;");

#endif

  RETURN;
}

static void print_footer(const ABIPData *d, spe_problem *spe, ABIPSolution *sol,
                         ABIPWork *w, ABIPInfo *info, abip_int k) {
  DEBUG_FUNC

  abip_int i;

  char *lin_sys_str = ABIP(get_lin_sys_summary)(spe, info);

  for (i = 0; i < LINE_LEN; ++i) {
    abip_printf("-");
  }

  abip_printf("\n");

  abip_printf("Status: %s\n", info->status);

  if (info->ipm_iter + 1 == spe->stgs->max_ipm_iters) {
    abip_printf("Hit max_ipm_iters, solution may be inaccurate\n");
  }

  if (info->admm_iter + 1 >= spe->stgs->max_admm_iters) {
    abip_printf("Hit max_admm_iters, solution may be inaccurate\n");
  }

  abip_printf("Timing: Solve time: %1.2es\n", info->solve_time / 1e3);
  abip_printf("         per iter: %1.2es\n", info->solve_time / (k * 1e3));
  abip_printf("       Total time: %1.2es\n",
              (info->solve_time + info->setup_time) / 1e3);
```

```
  if (lin_sys_str) {
    abip_printf("%s", lin_sys_str);
    abip_free(lin_sys_str);
  }

  for (i = 0; i < LINE_LEN; ++i) {
    abip_printf("-");
  }

  abip_printf("\n");

  if (is_infeasible_status(info->status_val)) {
    abip_printf("Certificate of primal infeasibility:\n");
    abip_printf("|A'y + s|_2 * |b|_2 = %.4e\n", info->res_infeas);
    abip_printf("b'y = %.4f\n", ABIP(dot)(d->b, sol->y, d->m));
  } else if (is_unbounded_status(info->status_val)) {
    abip_printf("Certificate of dual infeasibility:\n");
    abip_printf("|Ax|_2 * |c|_2 = %.4e\n", info->res_unbdd);
    abip_printf("c'x = %.4f\n", ABIP(dot)(d->c, sol->x, d->n));
  } else {
    abip_printf("Error metrics:\n");
    abip_printf(
        "primal res: |Ax - b|_inf / (1 + max(|Ax|_inf, |b|_inf)) = %.4e\n",
        info->res_pri);
    abip_printf(
        "dual res: |Qx - A'y + c - s|_inf / (1 + max(|Qx|_inf + |c|_inf)) = "
        "%.4e\n",
        info->res_dual);
    abip_printf(
        "rel gap: |x'Qx + c'x - b'y| / (1 + max(|x'Qx| + |c'x| + |b'y|)) = "
        "%.4e\n",
        info->rel_gap);

    for (i = 0; i < LINE_LEN; ++i) {
      abip_printf("-");
    }

    abip_printf("\n");
    abip_printf("1/2x'Qx + c'x = %.4e, -1/2x'Qx + b'y = %.4e\n", info->pobj,
                info->dobj);
  }

  for (i = 0; i < LINE_LEN; ++i) {
    abip_printf("=");
  }

  abip_printf("\n");

#ifdef MATLAB_MEX_FILE

  mexEvalString("drawnow;");

#endif

  RETURN;
}

static abip_int has_converged(ABIPWork *w, spe_problem *spe, ABIPResiduals *r,
                              abip_int ipm_iter, abip_int admm_iter) {
  DEBUG_FUNC

  abip_float eps_p = spe->stgs->eps_p;
  abip_float eps_d = spe->stgs->eps_d;
  abip_float eps_g = spe->stgs->eps_g;
  abip_float eps_inf = spe->stgs->eps_inf;
  abip_float eps_unb = spe->stgs->eps_unb;

  if (r->res_pri < eps_p && r->res_dual < eps_d && r->rel_gap < eps_g) {
    RETURN ABIP_SOLVED;
  }

  if (r->res_dif < spe->stgs->err_dif * MAX(MAX(eps_p, eps_d), eps_g)) {
    RETURN ABIP_SOLVED_INACCURATE;
  }

  if (r->res_unbdd < eps_unb && ipm_iter > 0 && admm_iter > 0) {
    RETURN ABIP_UNBOUNDED;
  }

  if (r->res_infeas < eps_inf && ipm_iter > 0 && admm_iter > 0) {
    RETURN ABIP_INFEASIBLE;
  }

  RETURN 0;
}
```

```
static abip_int validate(const ABIPData *d, const ABIPCone *k,
                         spe_problem *spe) {
  DEBUG_FUNC

  ABIPSettings *stgs = d->stgs;

  if (d->n <= 0) {
    abip_printf("n must be greater than 0; n = %li\n", (long)d->n);
    RETURN - 1;
  }

  if (spe->p > spe->q) {
    abip_printf("WARN: m larger than n, problem likely degenerate\n");
    RETURN - 1;
  }

  if (ABIP(validate_lin_sys)(d->A) < 0) {
    abip_printf("invalid linear system input data\n");
    RETURN - 1;
  }

  if (ABIP(validate_cones)(spe, k) < 0) {
    abip_printf("cone validation error\n");
    return -1;
  }

  if (stgs->max_ipm_iters <= 0) {
    abip_printf("max_ipm_iters must be positive\n");
    RETURN - 1;
  }

  if (stgs->max_admm_iters <= 0) {
    abip_printf("max_admm_iters must be positive\n");
    RETURN - 1;
  }

  if (stgs->eps_p <= 0 || stgs->eps_d <= 0 || stgs->eps_g <= 0 ||
      stgs->eps_inf <= 0 || stgs->eps_unb <= 0) {
    abip_printf("eps tolerance must be positive\n");
    RETURN - 1;
  }

  if (stgs->alpha <= 0 || stgs->alpha >= 2) {
    abip_printf("alpha must be in (0,2)\n");
    RETURN - 1;
  }

  if (stgs->rho_y <= 0) {
    abip_printf("rho_y must be positive (1e-3 works well).\n");
    RETURN - 1;
  }

  RETURN 0;
}

static ABIPWork *init_work(spe_problem *s, ABIPCone *k) {
  DEBUG_FUNC

  ABIPWork *w = (ABIPWork *)abip_calloc(1, sizeof(ABIPWork));
  abip_int l = s->p + s->q + 1;

  if (s->stgs->verbose) {
    print_init_header(s);
  }

  if (!w) {
    abip_printf("ERROR: allocating work failure\n");
    RETURN ABIP_NULL;
  }

  w->sigma = SIGMA;
  w->gamma = GAMMA;

  w->mu = 1.0;
  w->beta = 1.0;

  w->m = s->p;
  w->n = s->q;

  w->u = (abip_float *)abip_malloc(l * sizeof(abip_float));
  w->v = (abip_float *)abip_malloc(l * sizeof(abip_float));
  memset(w->u, 0, l * sizeof(abip_float));
  memset(w->v, 0, l * sizeof(abip_float));

  w->v_origin = (abip_float *)abip_malloc(l * sizeof(abip_float));
  w->u_t = (abip_float *)abip_malloc(l * sizeof(abip_float));
  w->rel_ut = (abip_float *)abip_malloc(l * sizeof(abip_float));
```

```
  w->r = (abip_float *)abip_malloc((w->n + w->m) * sizeof(abip_float));

  if (!w->u || !w->v || !w->u_t || !w->rel_ut || !w->v_origin || !w->r) {
    abip_printf("ERROR: work memory allocation failure\n");
    RETURN ABIP_NULL;
  }

  w->nm_inf_b = ABIP(norm_inf)(s->data->b, s->p);
  w->nm_inf_c = ABIP(norm_inf)(s->data->c, s->q);

  s->scaling_data(s, k);

  if (s->init_spe_linsys_work(s)) {
    abip_printf("ERROR: init_lin_sys_work failure\n");
    RETURN ABIP_NULL;
  }

  RETURN w;
}

static abip_int pre_calculate(ABIPWork *w, spe_problem *spe) {
  DEBUG_FUNC

  abip_int m = w->m;
  abip_int n = w->n;

  abip_float *zeros = (abip_float *)abip_malloc((m + n) * sizeof(abip_float));
  memset(zeros, 0, (m + n) * sizeof(abip_float));

  memcpy(w->r, spe->b, m * sizeof(abip_float));
  ABIP(scale_array)(w->r, -1, m);
  memcpy(&w->r[m], spe->c, n * sizeof(abip_float));

  spe->solve_spe_linsys(spe, w->r, ABIP_NULL, -1, 1e-12);

  abip_float *tem = (abip_float *)abip_malloc((m + n) * sizeof(abip_float));
  memcpy(tem, w->r, (m + n) * sizeof(abip_float));
  ABIP(c_dot)(tem, spe->rho_dr, m + n);
  w->a = spe->rho_dr[m + n] + ABIP(dot)(tem, w->r, m + n);

  abip_free(tem);
  abip_free(zeros);

  RETURN 0;
}

abip_int update_work(const ABIPData *d, ABIPWork *w, const ABIPSolution *sol,
                     const ABIPCone *c, spe_problem *spe

) {
  DEBUG_FUNC

  abip_int n = w->n;
  abip_int m = w->m;

  // initialize x,y
  abip_float *x = (abip_float *)abip_malloc(n * sizeof(abip_float));
  abip_float *y = (abip_float *)abip_malloc(m * sizeof(abip_float));
  memset(y, 0, m * sizeof(abip_float));
  /*-----initialize x----------*/
  abip_int i;
  abip_int count = 0;

  // soc
  if (c->qsize && c->q) {
    for (i = 0; i < c->qsize; ++i) {
      if (c->q[i] == 0) {
        continue;
      }
      memset(&x[count], 0, c->q[i] * sizeof(abip_float));
      x[count] = 1;
      count += c->q[i];
    }
  }

  // rsoc
  if (c->rqsize && c->rq) {
    for (i = 0; i < c->rqsize; ++i) {
      if (c->rq[i] < 3) {
        continue;
      }
      memset(&x[count], 0, c->rq[i] * sizeof(abip_float));
      x[count] = 1;
      x[count + 1] = 1;
      count += c->rq[i];
    }
  }
```

```
  // free cone
  if (c->f) {
    for (i = count; i < count + c->f; i++) {
      x[i] = 0;
    }
    count += c->f;
  }

  // zero cone
  if (c->z) {
    for (i = count; i < count + c->z; i++) {
      x[i] = 0;
    }
    count += c->z;
  }

  // positive orthant
  if (c->l) {
    for (i = count; i < count + c->l; i++) {
      x[i] = 1;
    }
    count += c->l;
  }

  // initialize u,v
  memcpy(w->u, y, m * sizeof(abip_float));
  memcpy(&w->u[m], x, n * sizeof(abip_float));
  w->u[m + n] = 1.0;
  // w->u[m+n+1] = 1;
  memcpy(w->v, y, m * sizeof(abip_float));
  memcpy(&w->v[m], x, n * sizeof(abip_float));
  w->v[m + n] = 1.0;

  pre_calculate(w, spe);

  abip_free(x);
  abip_free(y);
  return 0;
}

abip_float adjust_barrier(ABIPWork *w, ABIPResiduals *r, spe_problem *spe) {
  abip_float error_ratio = r->error_ratio;

  abip_float sigma = 0.8;
  abip_float ratio =
      w->mu / MIN(MIN(spe->stgs->eps_p, spe->stgs->eps_d), spe->stgs->eps_g);
  abip_float gamma;

  if (ratio > 50 && ratio <= 100) {
    gamma = 1.5;
  } else if (ratio > 10 && ratio <= 50) {
    gamma = 1.3;
  } else if (ratio > 5 && ratio <= 10) {
    gamma = 1.2;
  } else if (ratio > 1 && ratio <= 5) {
    gamma = 1.1;
  } else if (ratio > 0.5 && ratio <= 1) {
    gamma = 1;
  } else if (ratio > 0.1 && ratio <= 0.5) {
    gamma = 0.9;
  } else if (ratio > 0.05 && ratio <= 0.1) {
    gamma = 0.9;
  } else if (ratio > 0.01 && ratio <= 0.05) {
    gamma = 0.8;
  } else if (ratio > 0.005 && ratio <= 0.01) {
    gamma = 0.8;
  } else if (ratio > 0.001 && ratio <= 0.005) {
    gamma = 0.7;
  } else if (ratio > 0.0005 && ratio <= 0.001) {
    gamma = 0.7;
  } else if (ratio > 0.0001 && ratio <= 0.0005) {
    gamma = 0.6;
  } else if (ratio > 0.00005 && ratio <= 0.0001) {
    gamma = 0.6;
  } else {
    gamma = 0.5;
  }

  abip_float mix_ratio = error_ratio;

  if (mix_ratio > 22) {
    gamma = gamma * 4.4;
  } else if (mix_ratio > 18 && mix_ratio <= 22) {
    gamma = gamma * 4.2;
  } else if (mix_ratio > 15 && mix_ratio <= 18) {
    gamma = gamma * 4;
```

```
  } else if (mix_ratio > 12 && mix_ratio <= 15) {
    gamma = gamma * 3.8;
  } else if (mix_ratio > 8 && mix_ratio <= 12) {
    gamma = gamma * 3.6;
  } else if (mix_ratio > 6 && mix_ratio <= 8) {
    sigma = 0.81;
    gamma = gamma * 3.4;
  } else if (mix_ratio > 4 && mix_ratio <= 6) {
    sigma = 0.82;
    gamma = gamma * 3.4;
  } else if (mix_ratio > 3 && mix_ratio <= 4) {
    sigma = 0.83;
    gamma = gamma * 3.2;
  } else if (mix_ratio > 3 && mix_ratio <= 4) {
    sigma = 0.84;
    gamma = gamma * 3;
  } else if (mix_ratio > 2 && mix_ratio <= 3) {
    sigma = 0.85;
    gamma = gamma * 2.8;
  } else if (mix_ratio > 1.5 && mix_ratio <= 2) {
    sigma = 0.85;
    gamma = gamma * 2.6;
  } else if (mix_ratio < 1.5) {
    sigma = 0.85;
    gamma = gamma * 2.4;
  }

  sigma = sigma * 0.2;

  w->mu = sigma * w->mu;
  return gamma * POWF(w->mu, spe->stgs->psi);
}

abip_int ABIP(solve)(ABIPWork *w, const ABIPData *d, ABIPSolution *sol,
                     ABIPInfo *info, ABIPCone *c, spe_problem *s) {
  DEBUG_FUNC

  abip_int i;
  abip_int j;
  abip_int k;
  ABIP(timer) solve_timer;
  ABIP(timer) lin_timer;
  ABIP(timer) barrier_timer;
  ABIP(timer) res_timer;
  ABIP(timer) P_timer;
  ABIP(timer) uw_timer;
  abip_float lin_time = 0;
  abip_float barrier_time = 0;
  abip_float res_time = 0;
  abip_float P_time = 0;
  abip_float uw_time = 0;

  abip_float time_limit_left = 1e3 * s->stgs->time_limit - info->setup_time;

  ABIPResiduals *r = (ABIPResiduals *)abip_malloc(sizeof(ABIPResiduals));

  abip_int l = w->m + w->n + 1;

  if (!d || !sol || !info || !w) {
    abip_printf("ERROR: ABIP_NULL input\n");
    RETURN ABIP_FAILED;
  }

  abip_start_interrupt_listener();
  ABIP(tic)(&solve_timer);

  info->status_val = ABIP_UNFINISHED;
  r->last_ipm_iter = -1;
  r->last_admm_iter = -1;
  r->res_pri = 1e8;
  r->res_dual = 1e8;
  r->rel_gap = 1e8;
  r->error_ratio = 1e8;

  abip_float tol_inner = 4 * POWF(w->mu, s->stgs->psi);

  ABIP(tic)(&uw_timer);
  update_work(d, w, sol, c, s);
  uw_time += ABIP(tocq)(&uw_timer) / 1e3;

  if (s->stgs->verbose) {
    print_header(w);
  }

  k = 0;

  for (i = 0; i < s->stgs->max_ipm_iters; ++i) {
```

```
    for (j = 0; j < s->stgs->max_admm_iters; ++j) {
      ABIP(tic)(&lin_timer);
      if (projection(w, k, s, r) < 0) {
        RETURN failure(w, w->m, w->n, sol, info, ABIP_FAILED,
                       "error in project_lin_sys", "Failure");
      }
      lin_time += ABIP(tocq)(&lin_timer) / 1e3;
      ABIP(tic)(&barrier_timer);
      solve_barrier_subproblem(w, c, s);
      barrier_time += ABIP(tocq)(&barrier_timer) / 1e3;

      update_dual_vars(w);

      memcpy(w->v_origin, w->v, l * sizeof(abip_float));
      ABIP(c_dot)(w->v_origin, s->rho_dr, l);

      k += 1;

      ABIP(tic)(&P_timer);

      abip_float err_inner = s->inner_conv_check(s, w);

      if (err_inner < tol_inner || ABIP(tocq)(&solve_timer) > time_limit_left) {
        P_time += ABIP(tocq)(&P_timer) / 1e3;

        break;
      }
      P_time += ABIP(tocq)(&P_timer) / 1e3;

      if (abip_is_interrupted()) {
        RETURN failure(w, w->m, w->n, sol, info, ABIP_SIGINT, "Interrupted",
                       "Interrupted");
      }

#if EXTRA_VERBOSE > 0
      abip_printf("primal error: %.4f, dual error: %.4f, gap: %.4f\n",
                  r.err_pri / w->eps_p, r.err_dual / w->eps_d,
                  r.gap / w->eps_g);
#endif

      if ((j + 1) % s->stgs->inner_check_period == 0 || r->error_ratio <= 8) {
        ABIP(tic)(&res_timer);
        s->calc_residuals(s, w, r, i, k);
        res_time += ABIP(tocq)(&res_timer) / 1e3;

        if ((j + 1) % s->stgs->inner_check_period == 0) {
          print_summary(w, s, i, j, k, r, &solve_timer);
        }

        if ((info->status_val = has_converged(w, s, r, i, k)) != 0 ||
            k + 1 >= s->stgs->max_admm_iters * s->stgs->max_ipm_iters ||
            i + 1 >= s->stgs->max_ipm_iters ||
            ABIP(tocq)(&solve_timer) >
                time_limit_left)  // max running time is time_limit s
        {
          if (s->stgs->verbose && k > 0) {
            printf("\nin last admm iter:\n");
            print_summary(w, s, i, j, k, r, &solve_timer);
            abip_printf("total admm iter is %i\n", k);
          }

          get_solution(w, s, sol, info, r, i, k);
          info->solve_time = ABIP(tocq)(&solve_timer);

          if (s->stgs->verbose) {
            print_footer(d, s, sol, w, info, k);
            printf(
                "\ntotal time of project_lin_sys: %.2es\ntotal time of "
                "solve_barrier_subproblem: %.2es\ntotal time of calculate res: "
                "%.2es\ntotal time of calculate err_inner: %.2es\ntotal time "
                "of updating work: %.2es\n",
                lin_time, barrier_time, res_time, P_time, uw_time);
          }

          abip_end_interrupt_listener();

          RETURN info->status_val;
        }
      }

    }  // inner for

    if (s->sparsity || (i + 1) % s->stgs->outer_check_period == 0) {
      ABIP(tic)(&res_timer);
      s->calc_residuals(s, w, r, i, k);
      res_time += ABIP(tocq)(&res_timer) / 1e3;
      print_summary(w, s, i, j, k, r, &solve_timer);
```

```
      if ((info->status_val = has_converged(w, s, r, i, k)) != 0 ||
          k + 1 >= s->stgs->max_admm_iters * s->stgs->max_ipm_iters ||
          i + 1 >= s->stgs->max_ipm_iters ||
          ABIP(tocq)(&solve_timer) > time_limit_left) {
        if (s->stgs->verbose && k > 0) {
          printf("\nin last admm iter:\n");
          print_summary(w, s, i, j, k, r, &solve_timer);
          abip_printf("total admm iter is %i\n", k);
        }

        get_solution(w, s, sol, info, r, i, k);
        info->solve_time = ABIP(tocq)(&solve_timer);

        if (s->stgs->verbose) {
          print_footer(d, s, sol, w, info, k);
          printf(
              "\ntotal time of project_lin_sys: %.2es\ntotal time of "
              "solve_barrier_subproblem: %.2es\ntotal time of calculate res: "
              "%.2es\ntotal time of calculate err_inner: %.2es\n",
              lin_time, barrier_time, res_time, P_time);
        }

        abip_end_interrupt_listener();

        RETURN info->status_val;
      }
    }

    tol_inner = adjust_barrier(w, r, s);
  }

  RETURN info->status_val;
}

void ABIP(finish)(ABIPWork *w, spe_problem *spe) {
  DEBUG_FUNC

  if (w) {
    free_work(w);
  }

  if (spe->L) {
    spe->free_spe_linsys_work(spe);
  }

  RETURN;
}

ABIPWork *ABIP(init)(const ABIPData *d, ABIPInfo *info, spe_problem *s,
                     ABIPCone *c) {
  DEBUG_FUNC

#if EXTRA_VERBOSE > 1
  ABIP(tic)(&global_timer);
#endif

  ABIPWork *w;
  ABIP(timer) init_timer;
  abip_start_interrupt_listener();

  if (!d || !info) {
    abip_printf("ERROR: Missing ABIPData or ABIPInfo input\n");
    RETURN ABIP_NULL;
  }

#if EXTRA_VERBOSE > 0
  ABIP(print_data)(d);
#endif

#ifndef NOVALIDATE
  if (validate(d, c, s) < 0) {
    abip_printf("ERROR: Validation returned failure\n");
    RETURN ABIP_NULL;
  }
#endif

  ABIP(tic)(&init_timer);

  w = init_work(s, c);
  info->setup_time = ABIP(tocq)(&init_timer);

  if (d->stgs->verbose) {
    abip_printf("Setup time: %1.2es\n", info->setup_time / 1e3);
  }

  abip_end_interrupt_listener();

```

```
  RETURN w;
}

abip_int ABIP(init_problem)(spe_problem **s, ABIPData *d, ABIPSettings *stgs,
                            enum problem_type special_problem) {
  switch (special_problem) {
    case LASSO:
      return init_lasso((lasso **)s, d, stgs);
    case SVM:
      return init_svm((svm **)s, d, stgs);
    case QCP:
      return init_qcp((qcp **)s, d, stgs);
    case SVMQP:
      return init_svmqp((svmqp **)s, d, stgs);
    default:
      return init_qcp((qcp **)s, d, stgs);
  }
}

abip_int abip(const ABIPData *d, ABIPSolution *sol, ABIPInfo *info,
              ABIPCone *K) {
  DEBUG_FUNC

  spe_problem *s = (spe_problem *)abip_malloc(sizeof(spe_problem));
  enum problem_type prob_type;
  if (d->stgs->prob_type == 0)
    prob_type = LASSO;
  else if (d->stgs->prob_type == 1)
    prob_type = SVM;
  else if (d->stgs->prob_type == 2)
    prob_type = QCP;
  else if (d->stgs->prob_type == 3)
    prob_type = SVMQP;

  ABIP(init_problem)(&s, d, d->stgs, prob_type);
  abip_int status;
  ABIPWork *w = ABIP(init)(d, info, s,
                           K);  // call init_work() to call init_lin_sys_work()
                                // to perform LDL' factorization

#if EXTRA_VERBOSE > 0
  abip_printf("size of abip_int = %lu, size of abip_float = %lu\n",
              sizeof(abip_int), sizeof(abip_float));
#endif

  if (w) {
    ABIP(solve)(w, d, sol, info, K, s);
    status = info->status_val;
    ABIP(finish)(w, s);
  } else {
    status = failure(ABIP_NULL, d ? d->m : -1, d ? d->n : -1, sol, info,
                     ABIP_FAILED, "could not initialize work", "Failure");
  }

  RETURN status;
}
```

4.177 source/abip_version.c File Reference

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

Functions

- const char *ABIP() version (void)

4.177.1 Function Documentation

4.177.1.1 version()

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

Definition at line 3 of file abip_version.c.

4.178 abip_version.c

Go to the documentation of this file.

```
00001 #include "glbopts.h"
00002
00003 const char *ABIP(version)(void)
00004 {
00005     return ABIP_VERSION;
00006 }
```

4.179 source/cones.c File Reference

```
#include "cones.h"
```

Macros

- #define _CRT_SECURE_NO_WARNINGS

Functions

- abip_int ABIP() get_cone_dims (const ABIPCone *k)

  Calculate the total number of dimensions of all the cones.
- abip_int ABIP() validate_cones (spe_problem *spe, const ABIPCone *k)

  Check if the cone dimensions are valid.
- char *ABIP() get_cone_header (const ABIPCone *k)

  Get the number of variables and blocks of each cone.
- void ABIP() soc_barrier_subproblem (abip_float *x, abip_float *tmp, abip_float lambda, abip_int n)

  Barrier subproblem for the second order cone.
- void ABIP() rsoc_barrier_subproblem (abip_float *x, abip_float *tmp, abip_float lambda, abip_int n)

  Barrier subproblem for the rotated second order cone.
- void ABIP() free_barrier_subproblem (abip_float *x, abip_float *tmp, abip_float lambda, abip_int n)

  Barrier subproblem for the free cone.
- void ABIP() zero_barrier_subproblem (abip_float *x, abip_float *tmp, abip_float lambda, abip_int n)

  Barrier subproblem for the zero cone.
- void ABIP() positive_orthant_barrier_subproblem (abip_float *x, abip_float *tmp, abip_float lambda, abip_int n)

  Barrier subproblem for the positive orthant cone.

4.179.1 Macro Definition Documentation

4.179.1.1 _CRT_SECURE_NO_WARNINGS

```
#define _CRT_SECURE_NO_WARNINGS
```

Definition at line 1 of file cones.c.

4.179.2 Function Documentation

4.179.2.1 free_barrier_subproblem()

```
void ABIP() free_barrier_subproblem (
            abip_float * x,
            abip_float * tmp,
            abip_float lambda,
            abip_int n )
```

Barrier subproblem for the free cone.

Definition at line 255 of file cones.c.

4.179.2.2 get_cone_dims()

```
abip_int ABIP() get_cone_dims (
            const ABIPCone * k )
```

Calculate the total number of dimensions of all the cones.

Definition at line 8 of file cones.c.

4.179.2.3 get_cone_header()

```
char *ABIP() get_cone_header (
            const ABIPCone * k )
```

Get the number of variables and blocks of each cone.

Definition at line 87 of file cones.c.

4.179.2.4 positive_orthant_barrier_subproblem()

```
void ABIP() positive_orthant_barrier_subproblem (
            abip_float * x,
            abip_float * tmp,
            abip_float lambda,
            abip_int n )
```

Barrier subproblem for the positive orthant cone.

Definition at line 279 of file cones.c.

4.179.2.5 rsoc_barrier_subproblem()

```
void ABIP() rsoc_barrier_subproblem (
            abip_float * x,
            abip_float * tmp,
            abip_float lambda,
            abip_int n )
```

Barrier subproblem for the rotated second order cone.

Definition at line 169 of file cones.c.

4.179.2.6 soc_barrier_subproblem()

```
void ABIP() soc_barrier_subproblem (
            abip_float * x,
            abip_float * tmp,
            abip_float lambda,
            abip_int n )
```

Barrier subproblem for the second order cone.

Definition at line 130 of file cones.c.

4.179.2.7 validate_cones()

```
abip_int ABIP() validate_cones (
            spe_problem * spe,
            const ABIPCone * k )
```

Check if the cone dimensions are valid.

Definition at line 37 of file cones.c.

4.179.2.8 zero_barrier_subproblem()

```
void ABIP() zero_barrier_subproblem (
            abip_float * x,
            abip_float * tmp,
            abip_float lambda,
            abip_int n )
```

Barrier subproblem for the zero cone.

Definition at line 267 of file cones.c.

4.180 cones.c

Go to the documentation of this file.

```
#define _CRT_SECURE_NO_WARNINGS
#include "cones.h"

/* c = l + f + q[i] + s[i] */
abip_int ABIP(get_cone_dims)(const ABIPCone* k) {
  abip_int i, c = 0;

  if (k->l) {
    c += k->l;
  }
  if (k->z) {
    c += k->z;
  }
  if (k->f) {
    c += k->f;
  }
  if (k->qsize && k->q) {
    for (i = 0; i < k->qsize; ++i) {
      c += k->q[i];
    }
  }
  if (k->rqsize && k->rq) {
    for (i = 0; i < k->rqsize; ++i) {
      c += k->rq[i];
    }
  }

  return c;
}

abip_int ABIP(validate_cones)(spe_problem* spe, const ABIPCone* k) {
  abip_int i;
  if (ABIP(get_cone_dims)(k) != spe->q) {
    abip_printf("cone dimensions %li not equal to num rows in A = n = %li\n",
                (long)ABIP(get_cone_dims)(k), (long)spe->q);
    return -1;
  }
  if (k->l && k->l < 0) {
    abip_printf("lp cone error\n");
    return -1;
  }
  if (k->f && k->f < 0) {
    abip_printf("free cone error\n");
    return -1;
  }
  if (k->z && k->z < 0) {
    abip_printf("zero cone error\n");
    return -1;
  }
  if (k->qsize && k->q) {
    if (k->qsize < 0) {
      abip_printf("soc cone error\n");
      return -1;
    }
    for (i = 0; i < k->qsize; ++i) {
      if (k->q[i] < 0) {
        abip_printf("soc cone error\n");
        return -1;
      }
    }
```

```
  }
  if (k->rqsize && k->rq) {
    if (k->rqsize < 0) {
      abip_printf("rsoc cone error\n");
      return -1;
    }
    for (i = 0; i < k->rqsize; ++i) {
      if (k->rq[i] < 0) {
        abip_printf("rsoc cone error\n");
        return -1;
      }
    }
  }

  return 0;
}

char* ABIP(get_cone_header)(const ABIPCone* k) {
  abip_int i, soc_vars, rsoc_vars;
  char* tmp = (char*)abip_malloc(sizeof(char) * 512);
  sprintf(tmp, "Cones:");

  soc_vars = 0;
  if (k->qsize && k->q) {
    for (i = 0; i < k->qsize; i++) {
      soc_vars += k->q[i];
    }
    sprintf(tmp + strlen(tmp), "\tsoc vars: %li, soc blks: %li\n",
            (long)soc_vars, (long)k->qsize);
  }

  rsoc_vars = 0;
  if (k->rqsize && k->rq) {
    for (i = 0; i < k->rqsize; i++) {
      rsoc_vars += k->rq[i];
    }
    sprintf(tmp + strlen(tmp), "\trsoc vars: %li, rsoc blks: %li\n",
            (long)rsoc_vars, (long)k->rqsize);
  }

  if (k->f) {
    sprintf(tmp + strlen(tmp), "\tfree vars: %li\n", (long)k->f);
  }

  if (k->z) {
    sprintf(tmp + strlen(tmp), "\tzero vars: %li\n", (long)k->z);
  }

  if (k->l) {
    sprintf(tmp + strlen(tmp), "\tlinear vars: %li\n", (long)k->l);
  }
  return tmp;
}

/*  x in second order cone
    K = {(t,x)| t>||x||}
*/
void ABIP(soc_barrier_subproblem)(abip_float* x, abip_float* tmp,
                                  abip_float lambda, abip_int n) {
  abip_float a = tmp[0];
  abip_float tol = 1e-9;
  abip_float* b = (abip_float*)abip_malloc((n - 1) * sizeof(abip_float));
  memcpy(b, &tmp[1], (n - 1) * sizeof(abip_float));
  abip_float b_norm_sq = ABIP(norm_sq)(b, n - 1);
  if (ABS(a) <= tol) {
    x[0] = SQRTF(2 * lambda + b_norm_sq / 4);
    memcpy(&x[1], b, (n - 1) * sizeof(abip_float));
    ABIP(scale_array)(&x[1], 0.5, (n - 1));
  } else {
    abip_float coef1 = -(POWF(a, 2) - ABIP(norm_sq)(b, n - 1)) / lambda;
    abip_float coef2 =
        -(4 * POWF(a, 2) + 4 * ABIP(norm_sq)(b, n - 1) + 16 * lambda) / lambda;

    abip_float r = 16 * a * a /
                   (8 * lambda - a * a + b_norm_sq +
                    SQRTF(POWF((8 * lambda - a * a + b_norm_sq), 2) +
                          32 * a * a * lambda));
    abip_float s1 = (r - SQRTF(r * (r + 8))) / 2;
    abip_float s2 = (r + SQRTF(r * (r + 8))) / 2;

    abip_float s = a > 0 ? s2 : s1;
    abip_float eta = (s + 2) * a / s;
    ABIP(scale_array)(b, (s + 2) / (s + 4), n - 1);

    x[0] = eta;
    memcpy(&x[1], b, (n - 1) * sizeof(abip_float));
  }
```

```c
  abip_free(b);
}

/*  K = {(t1,t2,x)| 2*t1*t2>||x||}
    n = len(t1,t2,x)
*/
void ABIP(rsoc_barrier_subproblem)(abip_float* x, abip_float* tmp,
                                   abip_float lambda, abip_int n) {
  abip_int nx = n - 2;
  abip_float tol = 1e-9;

  abip_float zeta_eta = tmp[0];
  abip_float zeta_nu = tmp[1];

  abip_float* zeta_x = (abip_float*)abip_malloc(nx * sizeof(abip_float));
  memcpy(zeta_x, &tmp[2], nx * sizeof(abip_float));
  abip_float zeta_x_norm_sq = ABIP(norm_sq)(zeta_x, nx);

  if (zeta_eta + zeta_nu == 0) {
    x[1] =
        (-zeta_eta + SQRTF(zeta_eta * zeta_eta + 4 * lambda + zeta_x_norm_sq)) /
        2;
    x[0] = x[0] + zeta_eta;
    memcpy(&x[2], zeta_x, nx * sizeof(abip_float));
    ABIP(scale_array)(&x[2], 0.5, nx);
  } else {
    abip_float w;
    abip_float s;
    if (2 * zeta_eta * zeta_nu - zeta_x_norm_sq < 0) {
      w = (2 * POWF(zeta_eta + zeta_nu, 2) / lambda) /
          (-(2 * zeta_eta * zeta_nu - zeta_x_norm_sq) / (2 * lambda)) /
          (1 + 4 / (-(2 * zeta_eta * zeta_nu - zeta_x_norm_sq) / (2 * lambda)) +
           SQRTF(
               1 +
               (4 * (zeta_eta * zeta_eta + zeta_nu * zeta_nu + zeta_x_norm_sq) /
                    lambda +
                16) /
                   (-(2 * zeta_eta * zeta_nu - zeta_x_norm_sq) / (2 * lambda)) /
                   (-(2 * zeta_eta * zeta_nu - zeta_x_norm_sq) /
                    (2 * lambda))));
    } else {
      w = (2 * zeta_eta * zeta_nu - zeta_x_norm_sq) / (2 * lambda) *
          (1 - 4 / ((2 * zeta_eta * zeta_nu - zeta_x_norm_sq) / (2 * lambda)) +
           SQRTF(
               1 +
               (4 * (zeta_eta * zeta_eta + zeta_nu * zeta_nu + zeta_x_norm_sq) /
                    lambda +
                16) /
                   ((2 * zeta_eta * zeta_nu - zeta_x_norm_sq) / (2 * lambda)) /
                   ((2 * zeta_eta * zeta_nu - zeta_x_norm_sq) /
                    (2 * lambda)))) /
          2;
    }
    if (zeta_eta + zeta_nu > 0) {
      s = (w + SQRTF(w * (w + 4))) / 2;
      x[0] = (zeta_eta * POWF(s + 1, 2) + zeta_nu * (s + 1)) / (s * (s + 2));
      x[1] = (zeta_nu * POWF(s + 1, 2) + zeta_eta * (s + 1)) / (s * (s + 2));
      memcpy(&x[2], zeta_x, nx * sizeof(abip_float));
      ABIP(scale_array)(&x[2], (s + 1) / (s + 2), nx);
    } else {
      if (w > 10) {
        s = 2 / (w + 2 + SQRTF(w * (w + 4)));
        x[0] = (zeta_eta * POWF(s, 2) + zeta_nu * s) / ((s - 1) * (s + 1));
        x[1] = (zeta_nu * POWF(s, 2) + zeta_eta * s) / ((s - 1) * (s + 1));
        memcpy(&x[2], zeta_x, nx * sizeof(abip_float));
        ABIP(scale_array)(&x[2], s / (s + 1), nx);
      } else {
        s = (w - SQRTF(w * (w + 4))) / 2;
        x[0] = (zeta_eta * POWF(s + 1, 2) + zeta_nu * (s + 1)) / (s * (s + 2));
        x[1] = (zeta_nu * POWF(s + 1, 2) + zeta_eta * (s + 1)) / (s * (s + 2));
        memcpy(&x[2], zeta_x, nx * sizeof(abip_float));
        ABIP(scale_array)(&x[2], (s + 1) / (s + 2), nx);
      }
    }
  }

  for (int i = nx + 2; i < n; i++) {
    if (tmp[i] >= 0) {
      x[i] = (tmp[i] + SQRTF(tmp[i] * tmp[i] + 4 * lambda)) / 2;
    } else {
      x[i] = 2 * lambda /
             (-tmp[i] * (1 + SQRTF(1 + 4 * lambda / POWF(tmp[i], 2))));
    }
  }
  abip_free(zeta_x);
}
```

```c
/*  K = {x | x free}
 */
void ABIP(free_barrier_subproblem)(abip_float* x, abip_float* tmp,
                                   abip_float lambda, abip_int n) {
  for (int i = 0; i < n; i++) {
    x[i] = tmp[i];
  }
}

/*  K = {x | x = 0}
 */
void ABIP(zero_barrier_subproblem)(abip_float* x, abip_float* tmp,
                                   abip_float lambda, abip_int n) {
  for (int i = 0; i < n; i++) {
    x[i] = 0;
  }
}

/*  K = {x | x >= 0}
 */
void ABIP(positive_orthant_barrier_subproblem)(abip_float* x, abip_float* tmp,
                                               abip_float lambda, abip_int n) {
  for (int i = 0; i < n; i++) {
    if (tmp[i] >= 0) {
      x[i] = (tmp[i] + SQRTF(tmp[i] * tmp[i] + 4 * lambda)) / 2;
    } else {
      x[i] = 2 * lambda /
             (-tmp[i] * (1 + SQRTF(1 + 4 * lambda / POWF(tmp[i], 2))));
    }
  }
}
```

4.181 source/ctrlc.c File Reference

```c
#include "ctrlc.h"
```

4.182 ctrlc.c

Go to the documentation of this file.

```c
/*
 * Implements signal handling (ctrl-c) for ABIP.
 *
 * Under Windows, we use SetConsoleCtrlHandler.
 * Under Unix systems, we use sigaction.
 * For Mex files, we use utSetInterruptEnabled/utIsInterruptPending.
 *
 */

#include "ctrlc.h"

#if CTRLC > 0

#ifdef MATLAB_MEX_FILE

static int istate;
void abip_start_interrupt_listener(void)
{
      istate = utSetInterruptEnabled(1);
}

void abip_end_interrupt_listener(void)
{
      utSetInterruptEnabled(istate);
}

int abip_is_interrupted(void)
{
      return utIsInterruptPending();
}

#elif(defined _WIN32 || _WIN64 || defined _WINDLL)

static int int_detected;
static BOOL WINAPI abip_handle_ctrlc(DWORD dwCtrlType)
```

```
{
      if (dwCtrlType != CTRL_C_EVENT)
      {
            return FALSE;
      }

      int_detected = 1;
      return TRUE;
}

void abip_start_interrupt_listener(void)
{
      int_detected = 0;
      SetConsoleCtrlHandler(abip_handle_ctrlc, TRUE);
}

void abip_end_interrupt_listener(void)
{
      SetConsoleCtrlHandler(abip_handle_ctrlc, FALSE);
}

int abip_is_interrupted(void)
{
      return int_detected;
}

#else /* Unix */

#include <signal.h>
static int int_detected;
struct sigaction oact;
static void abip_handle_ctrlc(int dummy)
{
      int_detected = dummy ? dummy : -1;
}

void abip_start_interrupt_listener(void)
{
      struct sigaction act;
      int_detected = 0;

      act.sa_flags = 0;
      sigemptyset(&act.sa_mask);

      act.sa_handler = abip_handle_ctrlc;
      sigaction(SIGINT, &act, &oact);
}

void abip_end_interrupt_listener(void)
{
      struct sigaction act;
      sigaction(SIGINT, &oact, &act);
}

int abip_is_interrupted(void)
{
      return int_detected;
}

#endif /* END IF MATLAB_MEX_FILE / WIN32 */

#endif /* END IF CTRLC > 0 */
```

4.183 source/lasso_config.c File Reference

```
#include "lasso_config.h"
```

Macros

- #define MIN_SCALE (1e-3)
- #define MAX_SCALE (1e3)

Functions

- abip_int init_lasso (lasso **self, ABIPData *d, ABIPSettings *stgs)

  Initialize the lasso problem structure.
- void lasso_A_times (lasso *self, const abip_float *x, abip_float *y)

  Customized matrix-vector multiplication for the lasso problem with A untransposed.
- void lasso_AT_times (lasso *self, const abip_float *x, abip_float *y)

  Customized matrix-vector multiplication for the lasso problem with A transposed.
- void scaling_lasso_data (lasso *self, ABIPCone *k)

  Customized scaling procedure for the lasso problem.
- void get_unscaled_x (lasso *self, abip_float *x, abip_float *us_x)

  Get the unscaled solution x of the lasso qcp problem.
- void get_unscaled_y (lasso *self, abip_float *y, abip_float *us_y)

  Get the unscaled solution y of the lasso qcp problem.
- void get_unscaled_s (lasso *self, abip_float *s, abip_float *us_s)

  Get the unscaled solution s of the lasso qcp problem.
- void un_scaling_lasso_sol (lasso *self, ABIPSolution *sol)

  Get the unscaled solution of the original lasso problem.
- abip_float lasso_inner_conv_check (lasso *self, ABIPWork *w)

  Check whether the inner loop of the lasso problem has converged.
- void calc_lasso_residuals (lasso *self, ABIPWork *w, ABIPResiduals *r, abip_int ipm_iter, abip_int admm←_iter)

  Calculate the residuals of the lasso qcp problem.
- cs * form_lasso_kkt (lasso *self)

  Formulate the qcp KKT matrix of the lasso problem.
- void init_lasso_precon (lasso *self)

  Initialize the preconditioner of conjugate gradient method for the lasso problem.
- abip_float get_lasso_pcg_tol (abip_int k, abip_float error_ratio, abip_float norm_p)

  Get the tolerance of the conjugate gradient method for the lasso problem.
- abip_int init_lasso_linsys_work (lasso *self)

  Initialize the linear system solver work space for the lasso problem.
- abip_int solve_lasso_linsys (lasso *self, abip_float *b, abip_float *pcg_warm_start, abip_int iter, abip_float error_ratio)

  Customized linear system solver for the lasso problem.
- void free_lasso_linsys_work (lasso *self)

  Free the linear system solver work space for the lasso problem.

4.183.1 Macro Definition Documentation

4.183.1.1 MAX_SCALE

```
#define MAX_SCALE (1e3)
```

Definition at line 3 of file lasso_config.c.

4.183.1.2 MIN_SCALE

```
#define MIN_SCALE (1e-3)
```

Definition at line 2 of file lasso_config.c.

4.183.2 Function Documentation

4.183.2.1 calc_lasso_residuals()

```
void calc_lasso_residuals (
            lasso * self,
            ABIPWork * w,
            ABIPResiduals * r,
            abip_int ipm_iter,
            abip_int admm_iter )
```

Calculate the residuals of the lasso qcp problem.

Definition at line 367 of file lasso_config.c.

4.183.2.2 form_lasso_kkt()

```
cs * form_lasso_kkt (
            lasso * self )
```

Formulate the qcp KKT matrix of the lasso problem.

Definition at line 507 of file lasso_config.c.

4.183.2.3 free_lasso_linsys_work()

```
void free_lasso_linsys_work (
            lasso * self )
```

Free the linear system solver work space for the lasso problem.

Definition at line 722 of file lasso_config.c.

4.183.2.4 get_lasso_pcg_tol()

```
abip_float get_lasso_pcg_tol (
             abip_int k,
             abip_float error_ratio,
             abip_float norm_p )
```

Get the tolerance of the conjugate gradient method for the lasso problem.

Definition at line 592 of file lasso_config.c.

4.183.2.5 get_unscaled_s()

```
void get_unscaled_s (
             lasso * self,
             abip_float * s,
             abip_float * us_s )
```

Get the unscaled solution s of the lasso qcp problem.

Definition at line 289 of file lasso_config.c.

4.183.2.6 get_unscaled_x()

```
void get_unscaled_x (
             lasso * self,
             abip_float * x,
             abip_float * us_x )
```

Get the unscaled solution x of the lasso qcp problem.

Definition at line 265 of file lasso_config.c.

4.183.2.7 get_unscaled_y()

```
void get_unscaled_y (
             lasso * self,
             abip_float * y,
             abip_float * us_y )
```

Get the unscaled solution y of the lasso qcp problem.

Definition at line 279 of file lasso_config.c.

4.183.2.8 init_lasso()

```
abip_int init_lasso (
            lasso ** self,
            ABIPData * d,
            ABIPSettings * stgs )
```

Initialize the lasso problem structure.

Definition at line 8 of file lasso_config.c.

4.183.2.9 init_lasso_linsys_work()

```
abip_int init_lasso_linsys_work (
            lasso * self )
```

Initialize the linear system solver work space for the lasso problem.

Definition at line 624 of file lasso_config.c.

4.183.2.10 init_lasso_precon()

```
void init_lasso_precon (
            lasso * self )
```

Initialize the preconditioner of conjugate gradient method for the lasso problem.

Definition at line 571 of file lasso_config.c.

4.183.2.11 lasso_A_times()

```
void lasso_A_times (
            lasso * self,
            const abip_float * x,
            abip_float * y )
```

Customized matrix-vector multiplication for the lasso problem with A untransposed.

Definition at line 99 of file lasso_config.c.

4.183.2.12 lasso_AT_times()

```
void lasso_AT_times (
            lasso * self,
            const abip_float * x,
            abip_float * y )
```

Customized matrix-vector multiplication for the lasso problem with A transposed.

Definition at line 116 of file lasso_config.c.

4.183.2.13 lasso_inner_conv_check()

```
abip_float lasso_inner_conv_check (
            lasso * self,
            ABIPWork * w )
```

Check whether the inner loop of the lasso problem has converged.

Definition at line 323 of file lasso_config.c.

4.183.2.14 scaling_lasso_data()

```
void scaling_lasso_data (
            lasso * self,
            ABIPCone * k )
```

Customized scaling procedure for the lasso problem.

Definition at line 131 of file lasso_config.c.

4.183.2.15 solve_lasso_linsys()

```
abip_int solve_lasso_linsys (
            lasso * self,
            abip_float * b,
            abip_float * pcg_warm_start,
            abip_int iter,
            abip_float error_ratio )
```

Customized linear system solver for the lasso problem.

Definition at line 648 of file lasso_config.c.

4.183.2.16 un_scaling_lasso_sol()

```
void un_scaling_lasso_sol (
            lasso * self,
            ABIPSolution * sol )
```

Get the unscaled solution of the original lasso problem.

Definition at line 303 of file lasso_config.c.

4.184 lasso_config.c

Go to the documentation of this file.

```
#include "lasso_config.h"
#define MIN_SCALE (1e-3)
#define MAX_SCALE (1e3)

abip_int init_lasso(lasso **self, ABIPData *d, ABIPSettings *stgs) {
  lasso *this_lasso = (lasso *)abip_malloc(sizeof(lasso));
  *self = this_lasso;

  this_lasso->m = d->m;
  this_lasso->n = d->n;
  this_lasso->p = d->m + 1;
  this_lasso->q = 2 + 2 * d->n + d->m;
  abip_int m = this_lasso->p;
  abip_int n = this_lasso->q;

  this_lasso->lambda = d->lambda;
  this_lasso->Q = ABIP_NULL;
  this_lasso->sparsity = (((abip_float)d->A->p[d->n] / (d->m * d->n)) < 0.1);

  // non-identity DR scaling
  this_lasso->rho_dr =
      (abip_float *)abip_malloc((m + n + 1) * sizeof(abip_float));
  for (int i = 0; i < m + n + 1; i++) {
    if (i < m) {
      this_lasso->rho_dr[i] = stgs->rho_y;
    } else if (i < m + n) {
      this_lasso->rho_dr[i] = stgs->rho_x;
    } else {
      this_lasso->rho_dr[i] = stgs->rho_tau;
    }
  }

  if (this_lasso->sparsity) {
    this_lasso->sc = 2;
    this_lasso->sc_c = 1 / d->lambda;
    this_lasso->sc_cone2 = d->lambda / d->m * 80;
    this_lasso->sc_cone1 = 0.8 / this_lasso->sc_c / this_lasso->sc_cone2;
    this_lasso->sc_b = this_lasso->sc_c * 300 * d->lambda / d->m;
  } else {
    if (d->m < d->n)
      this_lasso->sc = 4;
    else
      this_lasso->sc = 1;
    this_lasso->sc_c = 1 / d->lambda;
    this_lasso->sc_b = this_lasso->sc_c;
    this_lasso->sc_cone2 = 0.8;
    this_lasso->sc_cone1 = 1 / this_lasso->sc_c;
  }

  this_lasso->L = (ABIPLinSysWork *)abip_malloc(sizeof(ABIPLinSysWork));
  this_lasso->pro_type = LASSO;
  this_lasso->stgs = stgs;
  this_lasso->data = d;

  abip_float *data_b =
      (abip_float *)abip_malloc(this_lasso->p * sizeof(abip_float));
  data_b[0] = 1;
  memcpy(&data_b[1], d->b, d->m * sizeof(abip_float));
  this_lasso->data->b = data_b;

  abip_float *data_c =
      (abip_float *)abip_malloc(this_lasso->q * sizeof(abip_float));
  data_c[0] = 0;
```

```
  data_c[1] = 2;
  memset(&data_c[2], 0, d->m * sizeof(abip_float));
  for (int i = 0; i < 2 * d->n; i++) {
    data_c[i + 2 + d->m] = d->lambda;
  }
  this_lasso->data->c = data_c;
  /* for lasso problem, no need for inputing c*/
  d->c = data_c;

  this_lasso->A = (ABIPMatrix *)abip_malloc(sizeof(ABIPMatrix));
  this_lasso->b = (abip_float *)abip_malloc(this_lasso->p * sizeof(abip_float));
  this_lasso->c = (abip_float *)abip_malloc(this_lasso->q * sizeof(abip_float));
  this_lasso->D = (abip_float *)abip_malloc(this_lasso->m * sizeof(abip_float));
  this_lasso->E = (abip_float *)abip_malloc(this_lasso->n * sizeof(abip_float));

  this_lasso->scaling_data = &scaling_lasso_data;
  this_lasso->un_scaling_sol = &un_scaling_lasso_sol;
  this_lasso->calc_residuals = &calc_lasso_residuals;
  this_lasso->init_spe_linsys_work = &init_lasso_linsys_work;
  this_lasso->solve_spe_linsys = &solve_lasso_linsys;
  this_lasso->free_spe_linsys_work = &free_lasso_linsys_work;
  this_lasso->spe_A_times = &lasso_A_times;
  this_lasso->spe_AT_times = &lasso_AT_times;
  this_lasso->inner_conv_check = &lasso_inner_conv_check;

  return 0;
}

void lasso_A_times(lasso *self, const abip_float *x, abip_float *y) {
  y[0] += x[0];

  for (int i = 1; i < self->p; i++) {
    y[i] += self->D[i - 1] * SQRTF(self->sc_cone2) * x[i + 1];
  }

  ABIP(accum_by_A)(self->A, &x[self->m + 2], &y[1]);
  ABIP(scale_array)(&y[1], -1, self->m);
  ABIP(accum_by_A)(self->A, &x[self->m + self->n + 2], &y[1]);
  ABIP(scale_array)(&y[1], -1, self->m);
}

void lasso_AT_times(lasso *self, const abip_float *x, abip_float *y) {
  y[0] += x[0];
  for (int i = 2; i < self->m + 2; i++) {
    y[i] += x[i - 1] * self->D[i - 2] * SQRTF(self->sc_cone2);
  }

  ABIP(accum_by_Atrans)(self->A, &x[1], &y[self->m + 2]);
  ABIP(scale_array)(&y[self->m + 2 + self->n], -1, self->n);
  ABIP(accum_by_Atrans)(self->A, &x[1], &y[self->m + 2 + self->n]);
  ABIP(scale_array)(&y[self->m + 2 + self->n], -1, self->n);
}

void scaling_lasso_data(lasso *self, ABIPCone *k) {
  if (!ABIP(copy_A_matrix)(&(self->A), self->data->A)) {
    abip_printf("ERROR: copy A matrix failed\n");
    RETURN;
  }

  abip_int m = self->m;
  abip_int n = self->n;
  ABIPMatrix *A = self->A;

  abip_float min_row_scale = MIN_SCALE * SQRTF((abip_float)n);
  abip_float max_row_scale = MAX_SCALE * SQRTF((abip_float)n);
  abip_float min_col_scale = MIN_SCALE * SQRTF((abip_float)m);
  abip_float max_col_scale = MAX_SCALE * SQRTF((abip_float)m);

  abip_float *E = self->E;
  memset(E, 0, n * sizeof(abip_float));

  abip_float *D = self->D;
  memset(D, 0, m * sizeof(abip_float));

  abip_float avg = 0;
  abip_float avg1 = 0;

  if (self->stgs->scale_E) {
    if (self->sparsity) {
      for (int i = 0; i < n; i++) {
        for (int j = A->p[i]; j < A->p[i + 1]; j++) {
          E[i] += A->x[j] * A->x[j];
        }

        avg += SQRTF(E[i]);
      }

```

```
    avg /= n;

    for (int i = 0; i < n; i++) {
      E[i] = avg / SQRTF(E[i] + 1e-4) / self->sc;
      if (E[i] > 1000 * SQRTF(m)) {
        E[i] = 1000 * SQRTF(m);
      }
      if (E[i] < 0.001 * SQRTF(m)) {
        E[i] = 1;
      }
      if (E[i] > 50) {
        E[i] = 50;
      }

      avg1 += E[i];
    }

    avg1 /= n;

    for (int i = 0; i < n; i++) {
      E[i] = avg1 / E[i] / self->sc;
    }
  } else {
    for (int i = 0; i < n; i++) {
      for (int j = A->p[i]; j < A->p[i + 1]; j++) {
        E[i] += A->x[j] * A->x[j];
      }

      E[i] = SQRTF(E[i]);

      if (E[i] > 1000 * SQRTF(m)) {
        E[i] = 1000 * SQRTF(m);
      }
      if (E[i] < 0.001 * SQRTF(m)) {
        E[i] = 1;
      }
      if (E[i] > 7) {
        E[i] = 7;
      }

      E[i] = 1 / (E[i] * self->sc);
    }
  }

  for (int i = 0; i < n; i++) {
    for (int j = A->p[i]; j < A->p[i + 1]; j++) {
      A->x[j] *= E[i];
    }
  }
}

for (int i = 0; i < A->p[n]; i++) {
  D[A->i[i]] += A->x[i] * A->x[i];
}

avg = 0;
for (int i = 0; i < m; i++) {
  avg += SQRTF(2 * D[i] + self->sc_cone2);
}

avg /= m;

for (int i = 0; i < m; i++) {
  D[i] = avg / SQRTF(2 * D[i] + self->sc_cone2);
}

for (int i = 0; i < A->p[n]; i++) {
  A->x[i] *= D[A->i[i]];
}

memcpy(self->b, self->data->b, self->p * sizeof(abip_float));

self->b[0] = self->sc_cone1;

for (int i = 1; i < m + 1; i++) {
  self->b[i] *= D[i - 1];
}
ABIP(scale_array)(self->b, self->sc_b, self->p);

self->c[0] = 0;
self->c[1] = self->sc_cone1 * self->sc_cone2;
memset(&self->c[2], 0, self->m * sizeof(abip_float));

for (int i = 0; i < self->n; i++) {
  self->c[i + self->m + 2] = E[i] * self->lambda;
  self->c[i + self->m + 2 + self->n] = E[i] * self->lambda;
}
```

```
  ABIP(scale_array)(self->c, self->sc_c, self->q);

  self->D_hat = (abip_float *)abip_malloc(m * sizeof(abip_float));

  for (int i = 0; i < m; i++) {
    self->D_hat[i] =
        self->D[i] * self->D[i] * self->sc_cone2 + self->stgs->rho_y;
  }
}

void get_unscaled_x(lasso *self, abip_float *x, abip_float *us_x) {
  memcpy(us_x, x, self->q * sizeof(abip_float));
  for (int i = 0; i < self->m; i++) {
    us_x[i + 2] /= (self->sc_b / SQRTF(self->sc_cone2));
  }
  for (int i = 0; i < self->n; i++) {
    us_x[self->m + 2 + i] /= (self->E[i] * self->sc_b);
    us_x[self->m + self->n + 2 + i] /= (self->E[i] * self->sc_b);
  }
}

void get_unscaled_y(lasso *self, abip_float *y, abip_float *us_y) {
  memcpy(us_y, y, self->p * sizeof(abip_float));
  for (int i = 0; i < self->m; i++) {
    us_y[i + 1] *= (self->D[i] / self->sc_c);
  }
}

void get_unscaled_s(lasso *self, abip_float *s, abip_float *us_s) {
  memcpy(us_s, s, self->q * sizeof(abip_float));
  for (int i = 0; i < self->m; i++) {
    us_s[i + 2] /= (self->sc_c / SQRTF(self->sc_cone2));
  }
  for (int i = 0; i < self->n; i++) {
    us_s[self->m + 2 + i] /= (self->E[i] * self->sc_c);
    us_s[self->m + self->n + 2 + i] /= (self->E[i] * self->sc_c);
  }
}

void un_scaling_lasso_sol(lasso *self, ABIPSolution *sol) {
  abip_int m = self->m;
  abip_int n = self->n;

  abip_float *beta = (abip_float *)abip_malloc(n * sizeof(abip_float));
  memcpy(beta, &sol->x[m + 2], n * sizeof(abip_float));
  ABIP(add_scaled_array)(beta, &sol->x[m + n + 2], n, -1);
  ABIP(c_dot)(beta, self->E, n);
  ABIP(scale_array)(beta, 1 / self->sc_b, n);

  abip_free(sol->x);
  abip_free(sol->y);
  abip_free(sol->s);

  sol->x = beta;
}

abip_float lasso_inner_conv_check(lasso *self, ABIPWork *w) {
  abip_int m = self->p;
  abip_int n = self->q;

  abip_float *Qu = (abip_float *)abip_malloc((m + n + 1) * sizeof(abip_float));
  abip_float *Mu = (abip_float *)abip_malloc((m + n) * sizeof(abip_float));

  memset(Mu, 0, (m + n) * sizeof(abip_float));

  self->spe_A_times(self, &w->u[m], Mu);

  self->spe_AT_times(self, w->u, &Mu[m]);
  ABIP(scale_array)(&Mu[m], -1, n);

  if (self->Q != ABIP_NULL) {
    ABIP(accum_by_A)(self->Q, &w->u[m], &Mu[m]);
  }

  memcpy(Qu, Mu, (m + n) * sizeof(abip_float));

  ABIP(add_scaled_array)(Qu, self->b, m, -w->u[m + n]);
  ABIP(add_scaled_array)(&Qu[m], self->c, n, w->u[m + n]);

  Qu[m + n] = -ABIP(dot)(w->u, Mu, m + n) / w->u[m + n] +
              ABIP(dot)(w->u, self->b, m) - ABIP(dot)(&w->u[m], self->c, n);

  abip_float *tem = (abip_float *)abip_malloc((m + n + 1) * sizeof(abip_float));
  memcpy(tem, Qu, (m + n + 1) * sizeof(abip_float));
  ABIP(add_scaled_array)(tem, w->v_origin, m + n + 1, -1);

  abip_float error_inner =
```

```
      ABIP(norm)(tem, m + n + 1) /
      (1 + ABIP(norm)(w->u, m + n + 1) + ABIP(norm)(w->v_origin, m + n + 1));

  abip_free(Qu);
  abip_free(Mu);
  abip_free(tem);

  return error_inner;
}

void calc_lasso_residuals(lasso *self, ABIPWork *w, ABIPResiduals *r,
                          abip_int ipm_iter, abip_int admm_iter) {
  abip_int p = w->m;
  abip_int q = w->n;
  abip_int m = self->m;
  abip_int n = self->n;
  abip_float this_pr;
  abip_float this_dr;
  abip_float this_gap;

  r->tau = w->u[p + q];

  abip_float *x = (abip_float *)abip_malloc(m * sizeof(abip_float));
  memcpy(x, &(w->u[m + 3]), m * sizeof(abip_float));
  ABIP(scale_array)(x, SQRTF(self->sc_cone2) / (r->tau * self->sc_b), m);

  abip_float *beta_plus = (abip_float *)abip_malloc(n * sizeof(abip_float));
  memcpy(beta_plus, &(w->u[2 * m + 3]), n * sizeof(abip_float));
  for (int i = 0; i < n; i++) {
    beta_plus[i] *= self->E[i] / (r->tau * self->sc_b);
  }

  abip_float *beta_minus = (abip_float *)abip_malloc(n * sizeof(abip_float));
  memcpy(beta_minus, &(w->u[2 * m + 3 + n]), n * sizeof(abip_float));
  for (int i = 0; i < n; i++) {
    beta_minus[i] *= self->E[i] / (r->tau * self->sc_b);
  }

  abip_float *z = (abip_float *)abip_malloc(m * sizeof(abip_float));
  memcpy(z, &(w->u[1]), m * sizeof(abip_float));
  for (int i = 0; i < m; i++) {
    z[i] *= self->D[i] / (r->tau * self->sc_c);
  }

  abip_float *s1 = (abip_float *)abip_malloc(n * sizeof(abip_float));
  memcpy(s1, &(w->v[2 * m + 3]), n * sizeof(abip_float));
  for (int i = 0; i < n; i++) {
    s1[i] /= self->E[i] * r->tau * self->sc_c;
  }

  abip_float *s2 = (abip_float *)abip_malloc(n * sizeof(abip_float));
  memcpy(s2, &(w->v[2 * m + n + 3]), n * sizeof(abip_float));
  for (int i = 0; i < n; i++) {
    s2[i] /= self->E[i] * r->tau * self->sc_c;
  }

  abip_float *pr = (abip_float *)abip_malloc(m * sizeof(abip_float));
  memcpy(pr, x, m * sizeof(abip_float));
  ABIP(accum_by_A)(self->data->A, beta_plus, pr);
  ABIP(scale_array)(pr, -1, m);
  ABIP(accum_by_A)(self->data->A, beta_minus, pr);
  ABIP(scale_array)(pr, -1, m);
  ABIP(add_scaled_array)(pr, &self->data->b[1], m, -1);
  this_pr = ABIP(norm)(pr, m) / MAX(ABIP(norm)(&self->data->b[1], m), 1);

  abip_float *dr1 = (abip_float *)abip_malloc(n * sizeof(abip_float));
  memcpy(dr1, s1, n * sizeof(abip_float));
  ABIP(accum_by_Atrans)(self->data->A, z, dr1);
  for (int i = 0; i < n; i++) {
    dr1[i] -= self->lambda;
  }

  abip_float *dr2 = (abip_float *)abip_malloc(n * sizeof(abip_float));
  memcpy(dr2, s2, n * sizeof(abip_float));
  ABIP(scale_array)(dr2, -1, n);
  ABIP(accum_by_Atrans)(self->data->A, z, dr2);
  ABIP(scale_array)(dr2, -1, n);
  for (int i = 0; i < n; i++) {
    dr2[i] -= self->lambda;
  }
  abip_float *dr = (abip_float *)abip_malloc(2 * n * sizeof(abip_float));
  memcpy(dr, dr1, n * sizeof(abip_float));
  memcpy(&dr[n], dr2, n * sizeof(abip_float));
  this_dr = ABIP(norm)(dr, 2 * n) / (SQRTF(2 * n) * self->lambda);

  abip_float *lambda_ones = (abip_float *)abip_malloc(n * sizeof(abip_float));
  for (int i = 0; i < n; i++) {
```

```
      lambda_ones[i] = self->lambda;
    }
    this_gap =
        ABS(0.5 * ABIP(dot)(x, x, m) + ABIP(dot)(lambda_ones, beta_plus, n) +
            ABIP(dot)(lambda_ones, beta_minus, n) + 0.5 * ABIP(dot)(z, z, m) -
            ABIP(dot)(&self->data->b[1], z, m)) /
        (1 + ABS(0.5 * ABIP(dot)(x, x, m) + ABIP(dot)(lambda_ones, beta_plus, n) +
                 ABIP(dot)(lambda_ones, beta_minus, n)));

    r->last_ipm_iter = ipm_iter;
    r->last_admm_iter = admm_iter;

    r->dobj = (-0.5 * ABIP(dot)(z, z, m) + ABIP(dot)(&self->data->b[1], z, m));
    r->pobj = (0.5 * ABIP(dot)(x, x, m) + ABIP(dot)(lambda_ones, beta_plus, n) +
               ABIP(dot)(lambda_ones, beta_minus, n));

    r->res_dif = MAX(MAX(ABS(this_pr - r->res_pri), ABS(this_dr - r->res_dual)),
                     ABS(this_gap - r->rel_gap));
    r->res_pri = this_pr;
    r->res_dual = this_dr;
    r->rel_gap = this_gap;
    r->error_ratio =
        MAX(r->res_pri / self->stgs->eps_p,
            MAX(r->res_dual / self->stgs->eps_d, r->rel_gap / self->stgs->eps_g));

    if (ABIP(dot)(self->c, &w->u[p], q) < 0) {
      abip_float *Ax = (abip_float *)abip_malloc(p * sizeof(abip_float));
      memset(Ax, 0, p * sizeof(abip_float));
      self->spe_A_times(self, &w->u[p], Ax);
      r->res_unbdd = ABIP(norm)(Ax, p) / (-ABIP(dot)(self->c, &w->u[p], q));
      abip_free(Ax);
    } else {
      r->res_unbdd = INFINITY;
    }

    if (ABIP(dot)(self->b, w->u, p) > 0) {
      abip_float *ATy_s = (abip_float *)abip_malloc(q * sizeof(abip_float));
      memset(ATy_s, 0, q * sizeof(abip_float));
      self->spe_AT_times(self, w->u, ATy_s);
      ABIP(add_scaled_array)(ATy_s, &w->v_origin[p], q, 1);

      r->res_infeas = ABIP(norm)(ATy_s, q) / ABIP(dot)(self->b, w->u, p);
      abip_free(ATy_s);
    } else {
      r->res_infeas = INFINITY;
    }

    abip_free(x);
    abip_free(beta_plus);
    abip_free(beta_minus);
    abip_free(z);
    abip_free(s1);
    abip_free(s2);
    abip_free(pr);
    abip_free(dr);
    abip_free(dr1);
    abip_free(dr2);
    abip_free(lambda_ones);
  }

  cs *form_lasso_kkt(lasso *self) {
    abip_int n = self->n;
    abip_int m = self->m;
    cs *LTL;

    cs *Y1 = cs_spalloc(m, n, self->A->p[n], 1, 0);
    memcpy(Y1->i, self->A->i, self->A->p[n] * sizeof(abip_int));
    memcpy(Y1->p, self->A->p, (n + 1) * sizeof(abip_int));
    memcpy(Y1->x, self->A->x, self->A->p[n] * sizeof(abip_float));

    if (m > n) {
      cs *Y2 = cs_spalloc(m, n, self->A->p[n], 1, 0);
      memcpy(Y2->i, self->A->i, self->A->p[n] * sizeof(abip_int));
      memcpy(Y2->p, self->A->p, (n + 1) * sizeof(abip_int));
      memcpy(Y2->x, self->A->x, self->A->p[n] * sizeof(abip_float));

      cs *T1 = cs_spalloc(n, n, n, 1, 1); /* create triplet identity matrix */

      for (int i = 0; i < n; i++) {
        cs_entry(T1, i, i, 0.5);
      }

      cs *half_eye = cs_compress(T1);

      cs_spfree(T1);

      for (int i = 0; i < Y1->p[n]; i++) {
```

```
      Y1->x[i] /= self->D_hat[Y1->i[i]];
    }

    LTL = cs_add(half_eye, cs_multiply(cs_transpose(Y2, 1), Y1), 1, 1);

    cs_spfree(Y1);
    cs_spfree(Y2);
    cs_spfree(half_eye);
  } else {
    cs *YYT = cs_multiply(Y1, cs_transpose(Y1, 1));
    cs_spfree(Y1);

    cs *diag = cs_spalloc(m, m, m, 1, 1);

    for (int i = 0; i < m; i++) {
      cs_entry(diag, i, i, self->D_hat[i]);
    }
    cs *diag_D_hat = cs_compress(diag);
    cs_spfree(diag);

    LTL = cs_add(diag_D_hat, YYT, 1, 2);
    cs_spfree(YYT);
  }

  for (int i = 0; i < LTL->n; i++) {
    for (int j = LTL->p[i]; j < LTL->p[i + 1]; j++) {
      if (LTL->i[j] > i) LTL->x[j] = 0;
    }
  }
  cs_dropzeros(LTL);
  return LTL;
}

void init_lasso_precon(lasso *self) {
  self->L->M = (abip_float *)abip_malloc(self->p * sizeof(abip_float));
  memset(self->L->M, 0, self->p * sizeof(abip_float));

  abip_float *M = self->L->M;

  M[0] = 1 / (self->stgs->rho_y + 1);

  for (int i = 0; i < self->A->p[self->A->n]; i++) {
    M[self->A->i[i] + 1] += 2 * POWF(self->A->x[i], 2);
  }

  for (int i = 1; i < self->p; i++) {
    M[i] = 1 / (self->stgs->rho_y + M[i] +
                self->sc_cone2 * POWF(self->D[i - 1], 2));
  }
}

abip_float get_lasso_pcg_tol(abip_int k, abip_float error_ratio,
                             abip_float norm_p) {
  if (k == -1) {
    return 1e-9 * norm_p;
  } else {
    if (error_ratio > 100000) {
      return MAX(1e-9, 1.2e-2 * norm_p / POWF((k + 1), 2));
    } else if (error_ratio > 30000) {
      return MAX(1e-9, 8e-3 * norm_p / POWF((k + 1), 2));
    } else if (error_ratio > 10000) {
      return MAX(1e-9, 6e-3 * norm_p / POWF((k + 1), 2));
    } else if (error_ratio > 3000) {
      return MAX(1e-9, 5e-3 * norm_p / POWF((k + 1), 2));
    } else if (error_ratio > 1000) {
      return MAX(1e-9, 3e-3 * norm_p / POWF((k + 1), 2));
    } else if (error_ratio > 300) {
      return MAX(1e-9, 2e-3 * norm_p / POWF((k + 1), 2));
    } else if (error_ratio > 100) {
      return MAX(1e-9, 1.5e-3 * norm_p / POWF((k + 1), 2));
    } else if (error_ratio > 30) {
      return MAX(1e-9, 8e-4 * norm_p / POWF((k + 1), 2));
    } else if (error_ratio > 10) {
      return MAX(1e-9, 6e-4 * norm_p / POWF((k + 1), 2));
    } else {
      return MAX(1e-9, 5e-4 * norm_p / POWF((k + 1), 2));
    }
  }
}

abip_int init_lasso_linsys_work(lasso *self) {
  if (self->stgs->linsys_solver == 0) {
    self->L->K = cs_transpose(form_lasso_kkt(self), 1);
  } else if (self->stgs->linsys_solver == 1) {
    self->L->K = form_lasso_kkt(self);
  } else if (self->stgs->linsys_solver == 2) {
    self->L->K = form_lasso_kkt(self);
```

```
  } else if (self->stgs->linsys_solver == 3) {
    init_lasso_precon(self);
    self->L->K = ABIP_NULL;
  } else if (self->stgs->linsys_solver == 4) {
    self->L->K = cs_transpose(form_lasso_kkt(self), 1);
  } else if (self->stgs->linsys_solver == 5) {
    self->L->K = form_lasso_kkt(self);
  } else {
    printf("\nlinsys solver type error\n");
    return -1;
  }
  return ABIP(init_linsys_work)(self);
}

abip_int solve_lasso_linsys(lasso *self, abip_float *b,
                            abip_float *pcg_warm_start, abip_int iter,
                            abip_float error_ratio) {
  ABIP(timer) linsys_timer;
  ABIP(tic)(&linsys_timer);

  ABIP(scale_array)(&b[self->p], -1, self->q);

  if (self->stgs->linsys_solver == 3) {  // pcg

    abip_int p = self->p;
    abip_float norm_p = ABIP(norm)(b, p);

    self->spe_A_times(self, &b[p], b);
    abip_float pcg_tol = get_lasso_pcg_tol(iter, error_ratio, norm_p);
    abip_int cg_its = ABIP(solve_linsys)(self, b, p, pcg_warm_start, pcg_tol);

    if (iter >= 0) {
      self->L->total_cg_iters += cg_its;
    }
  }

  else {  // direct methods
    abip_float *b2 = (abip_float *)abip_malloc(self->p * sizeof(abip_float));
    memcpy(b2, b, self->p * sizeof(abip_float));
    self->spe_A_times(self, &b[self->p], b2);
    b[0] = b2[0] / (1 + self->stgs->rho_y);
    abip_int n = self->n;
    abip_int m = self->m;

    if (m > n) {
      for (int i = 0; i < m; i++) {
        b2[i + 1] /= self->D_hat[i];
        b[i + 1] = b2[i + 1];
      }

      abip_float *tmp = (abip_float *)abip_malloc(n * sizeof(abip_float));
      memset(tmp, 0, n * sizeof(abip_float));
      ABIP(accum_by_Atrans)(self->A, &b2[1], tmp);

      ABIP(solve_linsys)(self, tmp, n, ABIP_NULL, 0);

      abip_float *tmp2 = (abip_float *)abip_malloc(m * sizeof(abip_float));
      memset(tmp2, 0, m * sizeof(abip_float));

      ABIP(accum_by_A)(self->A, tmp, tmp2);
      for (int i = 0; i < m; i++) {
        tmp2[i] /= self->D_hat[i];
      }

      ABIP(add_scaled_array)(&b[1], tmp2, m, -1);

      abip_free(tmp);
      abip_free(tmp2);
    } else {
      memcpy(&b[1], &b2[1], m * sizeof(abip_float));

      ABIP(solve_linsys)(self, &b[1], m, ABIP_NULL, 0);
    }

    abip_free(b2);
  }

  ABIP(scale_array)(&b[self->p], -1, self->q);
  self->spe_AT_times(self, b, &b[self->p]);

  self->L->total_solve_time += ABIP(tocq)(&linsys_timer);

  return 0;
}

void free_lasso_linsys_work(lasso *self) { ABIP(free_linsys)(self); }
```

4.185 source/linalg.c File Reference

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

Functions

- void ABIP() c_dot (abip_float ∗x, const abip_float ∗y, const abip_int len)

  Elementwise multiplication of two vectors.
- abip_float ABIP() vec_mean (abip_float ∗x, abip_int len)

  Calculate the mean of a vector.
- void ABIP() set_as_scaled_array (abip_float ∗x, const abip_float ∗a, const abip_float b, abip_int len)

  Elementwise multiplication of a vector by a scalar.
- void ABIP() set_as_sqrt (abip_float ∗x, const abip_float ∗v, abip_int len)

  Elementwise square root of a vector.
- void ABIP() set_as_sq (abip_float ∗x, const abip_float ∗v, abip_int len)

  Elementwise square of a vector.
- void ABIP() scale_array (abip_float ∗a, const abip_float b, abip_int len)

  Elementwise multiplication of a vector by a scalar with replacement.
- abip_float ABIP() dot (const abip_float ∗x, const abip_float ∗y, abip_int len)

  Dot product of two vectors.
- abip_float ABIP() norm_sq (const abip_float ∗v, abip_int len)

  L2 norm of a vector.
- abip_float ABIP() norm (const abip_float ∗v, abip_int len)

  Square of L2 norm of a vector.
- abip_float ABIP() norm_1 (const abip_float ∗x, const abip_int len)

  L1 norm of a vector.
- abip_float ABIP() cone_norm_1 (const abip_float ∗x, const abip_int len)

  The absolute value of the largest component of x.
- abip_float ABIP() norm_inf (const abip_float ∗a, abip_int len)

  Calculate the maximum absolute value of a vector.
- void ABIP() add_array (abip_float ∗a, const abip_float b, abip_int len)

  Elementwise addition of two vectors with coefficients.
- void ABIP() add_scaled_array (abip_float ∗a, const abip_float ∗b, abip_int len, const abip_float sc)

  Elementwise addition of two vectors with coefficients 1.
- abip_float ABIP() norm_diff (const abip_float ∗a, const abip_float ∗b, abip_int len)

  L2 norm of the difference of two vectors.
- abip_float ABIP() norm_inf_diff (const abip_float ∗a, const abip_float ∗b, abip_int len)

  Maximum of the difference of two vectors.
- abip_int arr_ind (const abip_int i_col, const abip_int i_row, const abip_int nrows, const abip_int ncols, const abip_int format)
- abip_float ∗ABIP() csc_to_dense (const cs ∗in_csc, const abip_int out_format)

  Convert a CSC matrix to a dense matrix.

4.185.1 Function Documentation

4.185.1.1 add_array()

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

Elementwise addition of two vectors with coefficients.

Definition at line 181 of file linalg.c.

4.185.1.2 add_scaled_array()

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

Elementwise addition of two vectors with coefficients 1.

Definition at line 192 of file linalg.c.

4.185.1.3 arr_ind()

```
abip_int arr_ind (
            const abip_int i_col,
            const abip_int i_row,
            const abip_int nrows,
            const abip_int ncols,
            const abip_int format )
```

Definition at line 237 of file linalg.c.

4.185.1.4 c_dot()

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

Elementwise multiplication of two vectors.

Definition at line 9 of file linalg.c.

4.185.1.5 cone_norm_1()

```
abip_float ABIP() cone_norm_1 (
            const abip_float * x,
            const abip_int len )
```

The absolute value of the largest component of x.

Definition at line 144 of file linalg.c.

4.185.1.6 csc_to_dense()

```
abip_float *ABIP() csc_to_dense (
            const cs * in_csc,
            const abip_int out_format )
```

Convert a CSC matrix to a dense matrix.

Definition at line 247 of file linalg.c.

4.185.1.7 dot()

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

Dot product of two vectors.

Definition at line 85 of file linalg.c.

4.185.1.8 norm()

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

Square of L2 norm of a vector.

Definition at line 119 of file linalg.c.

4.185.1.9 norm_1()

```
abip_float ABIP() norm_1 (
            const abip_float * x,
            const abip_int len )
```

L1 norm of a vector.

Definition at line 130 of file linalg.c.

4.185.1.10 norm_diff()

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

L2 norm of the difference of two vectors.

Definition at line 207 of file linalg.c.

4.185.1.11 norm_inf()

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

Calculate the maximum absolute value of a vector.

Definition at line 161 of file linalg.c.

4.185.1.12 norm_inf_diff()

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

Maximum of the difference of two vectors.

Definition at line 223 of file linalg.c.

4.185.1.13 norm_sq()

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

L2 norm of a vector.

Definition at line 102 of file linalg.c.

4.185.1.14 scale_array()

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

Elementwise multiplication of a vector by a scalar with replacement.

Definition at line 71 of file linalg.c.

4.185.1.15 set_as_scaled_array()

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

Elementwise multiplication of a vector by a scalar.

Definition at line 37 of file linalg.c.

4.185.1.16 set_as_sq()

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

Elementwise square of a vector.

Definition at line 60 of file linalg.c.

4.185.1.17 set_as_sqrt()

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

Elementwise square root of a vector.

Definition at line 49 of file linalg.c.

4.185.1.18 vec_mean()

```
abip_float ABIP() vec_mean (
            abip_float * x,
            abip_int len )
```

Calculate the mean of a vector.

Definition at line 19 of file linalg.c.

4.186 linalg.c

Go to the documentation of this file.

```
00001 #include "linalg.h"
00002
00003 #include <math.h>
00004
00008 /* x = x .* y*/
00009 void ABIP(c_dot)(abip_float *x, const abip_float *y, const abip_int len) {
00010   for (int i = 0; i < len; i++) {
00011     x[i] *= y[i];
00012   }
00013 }
00014
00018 /* y = mean(x) */
00019 abip_float ABIP(vec_mean)(abip_float *x, abip_int len) {
00020   if (len <= 0 || x == ABIP_NULL) {
00021     printf("invalid ABIP(vec_mean) parameter");
00022     return -1;
00023   }
00024   abip_float y = 0;
00025
00026   for (int i = 0; i < len; i++) {
00027     y += x[i];
00028   }
00029
00030   return y / len;
00031 }
00032
00036 /* x = b*a */
00037 void ABIP(set_as_scaled_array)(abip_float *x, const abip_float *a,
00038                                const abip_float b, abip_int len) {
00039   abip_int i;
00040   for (i = 0; i < len; ++i) {
00041     x[i] = b * a[i];
00042   }
00043 }
00044
00048 /* x = sqrt(v) */
00049 void ABIP(set_as_sqrt)(abip_float *x, const abip_float *v, abip_int len) {
00050   abip_int i;
00051   for (i = 0; i < len; ++i) {
00052     x[i] = SQRTF(v[i]);
00053   }
00054 }
```

```c

/* x = v.^2 */
void ABIP(set_as_sq)(abip_float *x, const abip_float *v, abip_int len) {
  abip_int i;
  for (i = 0; i < len; ++i) {
    x[i] = v[i] * v[i];
  }
}

/* a *= b */
void ABIP(scale_array)(abip_float *a, const abip_float b, abip_int len) {
  if (a == ABIP_NULL) {
    return;
  }
  abip_int i;
  for (i = 0; i < len; ++i) {
    a[i] *= b;
  }
}

/* x'*y */
abip_float ABIP(dot)(const abip_float *x, const abip_float *y, abip_int len) {
  if (x == ABIP_NULL || y == ABIP_NULL) {
    return 0;
  }

  abip_int i;
  abip_float ip = 0.0;
  for (i = 0; i < len; ++i) {
    ip += x[i] * y[i];
  }
  return ip;
}

/* ||v||_2^2 */
abip_float ABIP(norm_sq)(const abip_float *v, abip_int len) {
  if (v == ABIP_NULL) {
    return 0;
  }

  abip_int i;
  abip_float nmsq = 0.0;
  for (i = 0; i < len; ++i) {
    nmsq += v[i] * v[i];
  }
  return nmsq;
}

/* ||v||_2 */
abip_float ABIP(norm)(const abip_float *v, abip_int len) {
  if (v == ABIP_NULL) {
    return 0;
  }
  return SQRTF(ABIP(norm_sq)(v, len));
}

/* ||x||_1 */
abip_float ABIP(norm_1)(const abip_float *x, const abip_int len) {
  if (x == ABIP_NULL) {
    return 0;
  }
  abip_float result = 0;
  for (int i = 0; i < len; i++) {
    result += ABS(x[i]);
  }
  return result;
}

abip_float ABIP(cone_norm_1)(const abip_float *x, const abip_int len) {
  abip_int i;
  abip_float tmp;
  abip_float max = 0.0;
  for (i = 0; i < len; ++i) {
    tmp = x[i];
    if (tmp > max) {
      max = tmp;
    }
  }
  return ABS(max);
}

/* max(|v|) */
abip_float ABIP(norm_inf)(const abip_float *a, abip_int len) {
  if (a == ABIP_NULL || len == 0) {
    return 0;
  }
  abip_int i;
```

```
  abip_float tmp;
  abip_float max = 0.0;
  for (i = 0; i < len; ++i) {
    tmp = ABS(a[i]);
    if (tmp > max) {
      max = tmp;
    }
  }
  return max;
}

/* a .+= b */
void ABIP(add_array)(abip_float *a, const abip_float b, abip_int len) {
  abip_int i;
  for (i = 0; i < len; ++i) {
    a[i] += b;
  }
}

/* saxpy a += sc*b */
void ABIP(add_scaled_array)(abip_float *a, const abip_float *b, abip_int len,
                            const abip_float sc) {
  if (b == ABIP_NULL) {
    return;
  }
  abip_int i;
  for (i = 0; i < len; ++i) {
    a[i] += sc * b[i];
  }
}

/* ||a-b||_2^2 */
abip_float ABIP(norm_diff)(const abip_float *a, const abip_float *b,
                           abip_int len) {
  abip_int i;
  abip_float tmp;
  abip_float nm_diff = 0.0;
  for (i = 0; i < len; ++i) {
    tmp = (a[i] - b[i]);
    nm_diff += tmp * tmp;
  }
  return SQRTF(nm_diff);
}

/* max(|a-b|) */
abip_float ABIP(norm_inf_diff)(const abip_float *a, const abip_float *b,
                               abip_int len) {
  abip_int i;
  abip_float tmp;
  abip_float max = 0.0;
  for (i = 0; i < len; ++i) {
    tmp = ABS(a[i] - b[i]);
    if (tmp > max) {
      max = tmp;
    }
  }
  return max;
}

abip_int arr_ind(const abip_int i_col, const abip_int i_row,
                 const abip_int nrows, const abip_int ncols,
                 const abip_int format) {
  return (format == RowMajor) ? (i_col + i_row * ncols)
                              : (i_row + i_col * nrows);
}

abip_float *ABIP(csc_to_dense)(const cs *in_csc, const abip_int out_format) {
  abip_int i_row, i_col, nnz_in_col, i_val = 0, i_nnz;
  const abip_int nrows = in_csc->m;
  const abip_int ncols = in_csc->n;
  abip_float *out_matrix =
      (abip_float *)abip_malloc(nrows * ncols * sizeof(abip_float));
  memset(out_matrix, 0, nrows * ncols * sizeof(abip_float));
  abip_int *col_nnz = in_csc->p;
  abip_int *rows = in_csc->i;
  abip_float *values = in_csc->x;

  for (i_col = 0; i_col < ncols; i_col++) {
    nnz_in_col = col_nnz[i_col + 1] - col_nnz[i_col];
    if (nnz_in_col > 0) {
      for (i_nnz = 0; i_nnz < nnz_in_col; i_nnz++) {
        i_row = rows[i_val];
        out_matrix[arr_ind(i_col, i_row, nrows, ncols, out_format)] =
            values[i_val];
        i_val++;
      }
    }
```

```
00268     }
00269
00270     return out_matrix;
00271 }
```

4.187 source/linsys.c File Reference

```
#include "linsys.h"
#include "amd.h"
```

Macros

- #define _CRT_SECURE_NO_WARNINGS
- #define MIN_SCALE (1e-3)
- #define MAX_SCALE (1e3)

Functions

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

  Copy a matrix.
- char *ABIP() get_lin_sys_method (spe_problem *spe)

  Get the method used to solve the linear system.
- char *ABIP() get_lin_sys_summary (spe_problem *self, ABIPInfo *info)

  Get the summary infomation of the linear system.
- abip_int ABIP() validate_lin_sys (const ABIPMatrix *A)

  Check the validity of the linear system.
- void ABIP() free_A_matrix (ABIPMatrix *A)

  Free the memory of a matrix.
- void ABIP() accum_by_Atrans (const ABIPMatrix *A, const abip_float *x, abip_float *y)

  Add the transposed matrix-vector product to a vector.
- void ABIP() accum_by_A (const ABIPMatrix *A, const abip_float *x, abip_float *y)

  Add the matrix-vector product to a vector.
- cs * permute_kkt (spe_problem *spe)
- _MKL_DSS_HANDLE_t init_mkl_work (cs *K)
- abip_int mkl_solve_linsys (_MKL_DSS_HANDLE_t handle, abip_float *b, abip_int n)
- abip_int init_pardiso (spe_problem *self)
- abip_int pardiso_solve (spe_problem *self, abip_float *b, abip_int n)
- abip_int pardiso_free (spe_problem *self)
- abip_int LDL_factor (cs *A, cs **L, abip_float *Dinv)
- abip_int abip_cholsol (spe_problem *self, abip_float *b, abip_int n)
- abip_int pcg (spe_problem *self, abip_float *b, abip_float *x, abip_float rho_x, abip_int max_iter, abip_float tol)

  Preconditioned conjugate gradient method for general linear system.
- abip_int qcp_pcg (spe_problem *self, abip_float *b, abip_float *x, abip_int max_iter, abip_float tol)

  Preconditioned conjugate gradient method for qcp.
- abip_int svmqp_pcg (spe_problem *self, abip_float *b, abip_float *x, abip_int max_iter, abip_float tol)

  Customized preconditioned conjugate gradient solver for QP formulation of SVM.
- abip_int init_dense_chol (spe_problem *spe)

  MKL-LAPACK dense cholesky linear system solver.

- abip_int dense_chol_sol (spe_problem *spe, abip_float *b, abip_int n)
- abip_int dense_chol_free (spe_problem *spe)
- abip_int ABIP() init_linsys_work (spe_problem *spe)

    Initialize linear system solver work space.
- abip_int ABIP() solve_linsys (spe_problem *spe, abip_float *b, abip_int n, abip_float *pcg_warm_start, abip_float pcg_tol)

    solve linear system according to the specific linsys solver
- abip_int ABIP() free_linsys (spe_problem *spe)

    free memory for linear system solver

4.187.1 Macro Definition Documentation

4.187.1.1 _CRT_SECURE_NO_WARNINGS

```
#define _CRT_SECURE_NO_WARNINGS
```

Definition at line 1 of file linsys.c.

4.187.1.2 MAX_SCALE

```
#define MAX_SCALE (1e3)
```

Definition at line 7 of file linsys.c.

4.187.1.3 MIN_SCALE

```
#define MIN_SCALE (1e-3)
```

Definition at line 6 of file linsys.c.

4.187.2 Function Documentation

4.187.2.1 abip_cholsol()

```
abip_int abip_cholsol (
            spe_problem * self,
            abip_float * b,
            abip_int n )
```

Definition at line 613 of file linsys.c.

4.187.2.2 accum_by_A()

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

Add the matrix-vector product to a vector.

Definition at line 229 of file linsys.c.

4.187.2.3 accum_by_Atrans()

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

Add the transposed matrix-vector product to a vector.

Definition at line 191 of file linsys.c.

4.187.2.4 copy_A_matrix()

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

Copy a matrix.

Definition at line 12 of file linsys.c.

4.187.2.5 dense_chol_free()

```
abip_int dense_chol_free (
            spe_problem * spe )
```

Definition at line 1012 of file linsys.c.

4.187.2.6 dense_chol_sol()

```
abip_int dense_chol_sol (
             spe_problem * spe,
             abip_float * b,
             abip_int n )
```

Definition at line 1005 of file linsys.c.

4.187.2.7 free_A_matrix()

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

Free the memory of a matrix.

Definition at line 152 of file linsys.c.

4.187.2.8 free_linsys()

```
abip_int ABIP() free_linsys (
             spe_problem * spe )
```

free memory for linear system solver

Definition at line 1181 of file linsys.c.

4.187.2.9 get_lin_sys_method()

```
char *ABIP() get_lin_sys_method (
             spe_problem * spe )
```

Get the method used to solve the linear system.

Definition at line 39 of file linsys.c.

4.187.2.10 get_lin_sys_summary()

```
char *ABIP() get_lin_sys_summary (
             spe_problem * self,
             ABIPInfo * info )
```

Get the summary infomation of the linear system.

Definition at line 71 of file linsys.c.

4.187.2.11 init_dense_chol()

```
abip_int init_dense_chol (
            spe_problem * spe )
```

MKL-LAPACK dense cholesky linear system solver.

Definition at line 995 of file linsys.c.

4.187.2.12 init_linsys_work()

```
abip_int ABIP() init_linsys_work (
            spe_problem * spe )
```

Initialize linear system solver work space.

Definition at line 1027 of file linsys.c.

4.187.2.13 init_mkl_work()

```
_MKL_DSS_HANDLE_t init_mkl_work (
            cs * K )
```

Definition at line 318 of file linsys.c.

4.187.2.14 init_pardiso()

```
abip_int init_pardiso (
            spe_problem * self )
```

Definition at line 361 of file linsys.c.

4.187.2.15 LDL_factor()

```
abip_int LDL_factor (
            cs * A,
            cs ** L,
            abip_float * Dinv )
```

Definition at line 542 of file linsys.c.

4.187.2.16 mkl_solve_linsys()

```
abip_int mkl_solve_linsys (
            _MKL_DSS_HANDLE_t handle,
            abip_float * b,
            abip_int n )
```

Definition at line 345 of file linsys.c.

4.187.2.17 pardiso_free()

```
abip_int pardiso_free (
            spe_problem * self )
```

Definition at line 527 of file linsys.c.

4.187.2.18 pardiso_solve()

```
abip_int pardiso_solve (
            spe_problem * self,
            abip_float * b,
            abip_int n )
```

Definition at line 503 of file linsys.c.

4.187.2.19 pcg()

```
abip_int pcg (
            spe_problem * self,
            abip_float * b,
            abip_float * x,
            abip_float rho_x,
            abip_int max_iter,
            abip_float tol )
```

Preconditioned conjugate gradient method for general linear system.

Definition at line 630 of file linsys.c.

4.187.2.20 permute_kkt()

```
cs * permute_kkt (
            spe_problem * spe )
```

Definition at line 275 of file linsys.c.

4.187.2.21 qcp_pcg()

```
abip_int qcp_pcg (
            spe_problem * self,
            abip_float * b,
            abip_float * x,
            abip_int max_iter,
            abip_float tol )
```

Preconditioned conjugate gradient method for qcp.

Definition at line 755 of file linsys.c.

4.187.2.22 solve_linsys()

```
abip_int ABIP() solve_linsys (
            spe_problem * spe,
            abip_float * b,
            abip_int n,
            abip_float * pcg_warm_start,
            abip_float pcg_tol )
```

solve linear system according to the specific linsys solver

Definition at line 1141 of file linsys.c.

4.187.2.23 svmqp_pcg()

```
abip_int svmqp_pcg (
            spe_problem * self,
            abip_float * b,
            abip_float * x,
            abip_int max_iter,
            abip_float tol )
```

Customized preconditioned conjugate gradient solver for QP formulation of SVM.

Definition at line 894 of file linsys.c.

4.187.2.24 validate_lin_sys()

```
abip_int ABIP() validate_lin_sys (
            const ABIPMatrix * A )
```

Check the validity of the linear system.

Definition at line 102 of file linsys.c.

4.188 linsys.c

Go to the documentation of this file.

```c
#define _CRT_SECURE_NO_WARNINGS
#include "linsys.h"

#include "amd.h"

#define MIN_SCALE (1e-3)
#define MAX_SCALE (1e3)

abip_int ABIP(copy_A_matrix)(ABIPMatrix **dstp, const ABIPMatrix *src) {
  abip_int Annz = src->p[src->n];
  ABIPMatrix *A = (ABIPMatrix *)abip_calloc(1, sizeof(ABIPMatrix));
  if (!A) {
    return 0;
  }
  A->n = src->n;
  A->m = src->m;
  A->x = (abip_float *)abip_malloc(sizeof(abip_float) * Annz);
  A->i = (abip_int *)abip_malloc(sizeof(abip_int) * Annz);
  A->p = (abip_int *)abip_malloc(sizeof(abip_int) * (src->n + 1));

  if (!A->x || !A->i || !A->p) {
    return 0;
  }

  memcpy(A->x, src->x, sizeof(abip_float) * Annz);
  memcpy(A->i, src->i, sizeof(abip_int) * Annz);
  memcpy(A->p, src->p, sizeof(abip_int) * (src->n + 1));

  *dstp = A;
  return 1;
}

char *ABIP(get_lin_sys_method)(spe_problem *spe) {
  char *tmp = (char *)abip_malloc(sizeof(char) * 128);

  if (spe->data->A == ABIP_NULL) {
    sprintf(tmp, "This problem has no linear constraints");
    return tmp;
  }

  abip_int n = spe->data->A->p[spe->data->A->n];

  if (spe->stgs->linsys_solver == 0) {
    sprintf(tmp, "sparse-direct using MKL-DSS, nnz in A = %li", (long)n);
  } else if (spe->stgs->linsys_solver == 1) {
    sprintf(tmp, "sparse-direct using QDLDL, nnz in A = %li", (long)n);
  } else if (spe->stgs->linsys_solver == 2) {
    sprintf(tmp, "sparse-direct using sparse cholesky, nnz in A = %li",
            (long)n);
  } else if (spe->stgs->linsys_solver == 3) {
    sprintf(tmp, "sparse-indirect using pcg, nnz in A = %li", (long)n);
  } else if (spe->stgs->linsys_solver == 4) {
    sprintf(tmp, "sparse-direct using MKL-PARDISO, nnz in A = %li", (long)n);
  } else if (spe->stgs->linsys_solver == 5) {
    sprintf(tmp, "dense-direct using dense cholesky, nnz in A = %li", (long)n);
  } else {
    sprintf(tmp, "\nlinsys solver type error\n");
  }
  return tmp;
}

char *ABIP(get_lin_sys_summary)(spe_problem *self, ABIPInfo *info) {
  char *str = (char *)abip_malloc(sizeof(char) * 128);

  abip_int n = self->L->nnz_LDL;
  info->avg_linsys_time =
      self->L->total_solve_time / (info->admm_iter + 1) / 1e3;

  if (self->stgs->linsys_solver == 3) {
    sprintf(str,
            "\tLin-sys: avg # CG iterations: %2.2f, avg solve time per admm "
            "iter: %1.2es\n",
            (abip_float)self->L->total_cg_iters / (info->admm_iter + 1),
            info->avg_linsys_time);
  } else {
    sprintf(str,
            "\tLin-sys: nnz in L factor: %li, avg solve time per admm iter: "
            "%1.2es\n",
            (long)n, info->avg_linsys_time);
  }

  info->avg_cg_iters =
```

```
      (abip_float)self->L->total_cg_iters / (info->admm_iter + 1);
  self->L->total_solve_time = 0;
  self->L->total_cg_iters = 0;

  return str;
}

abip_int ABIP(validate_lin_sys)(const ABIPMatrix *A) {
  abip_int i;
  abip_int r_max;
  abip_int Annz;

  if (A == ABIP_NULL) {
    return 0;
  }

  if (!A->x || !A->i || !A->p) {
    abip_printf("ERROR: incomplete data!\n");
    return -1;
  }

  for (i = 0; i < A->n; ++i) {
    if (A->p[i] == A->p[i + 1]) {
      abip_printf("WARN: the %li-th column empty!\n", (long)i);
    } else if (A->p[i] > A->p[i + 1]) {
      abip_printf("ERROR: the column pointers decreases!\n");
      return -1;
    }
  }

  Annz = A->p[A->n];
  if (((abip_float)Annz / A->m > A->n) || (Annz <= 0)) {
    abip_printf(
        "ERROR: the number of nonzeros in A = %li, outside of valid range!\n",
        (long)Annz);
    return -1;
  }

  r_max = 0;
  for (i = 0; i < Annz; ++i) {
    if (A->i[i] > r_max) {
      r_max = A->i[i];
    }
  }
  if (r_max > A->m - 1) {
    abip_printf(
        "ERROR: the number of rows in A is inconsistent with input "
        "dimension!\n");
    return -1;
  }

  return 0;
}

void ABIP(free_A_matrix)(ABIPMatrix *A) {
  if (A->x) {
    abip_free(A->x);
  }
  if (A->i) {
    abip_free(A->i);
  }
  if (A->p) {
    abip_free(A->p);
  }

  abip_free(A);
}

#if EXTRA_VERBOSE > 0

static void print_A_matrix(const ABIPMatrix *A) {
  abip_int i;
  abip_int j;

  if (A->p[A->n] < 2500) {
    abip_printf("\n");
    for (i = 0; i < A->n; ++i) {
      abip_printf("Col %li: ", (long)i);
      for (j = A->p[i]; j < A->p[i + 1]; j++) {
        abip_printf("A[%li,%li] = %4f, ", (long)A->i[j], (long)i, A->x[j]);
      }
      abip_printf("norm col = %4f\n",
                  ABIP(norm)(&(A->x[A->p[i]]), A->p[i + 1] - A->p[i]));
    }
    abip_printf("norm A = %4f\n", ABIP(norm)(A->x, A->p[A->n]));
  }
}
```

```
#endif

// y += A'*x
void ABIP(accum_by_Atrans)(const ABIPMatrix *A, const abip_float *x,
                           abip_float *y) {
  abip_int p;
  abip_int j;

  abip_int c1;
  abip_int c2;
  abip_float yj;

#if EXTRA_VERBOSE > 0
  ABIP(timer) mult_by_Atrans_timer;
  ABIP(tic)(&mult_by_Atrans_timer);
#endif

#ifdef _OPENMP
#pragma omp parallel for private(p, c1, c2, yj)
#endif

  for (j = 0; j < A->n; j++) {
    yj = y[j];
    c1 = A->p[j];
    c2 = A->p[j + 1];
    for (p = c1; p < c2; p++) {
      yj += A->x[p] * x[A->i[p]];
    }
    y[j] = yj;
  }

#if EXTRA_VERBOSE > 0
  abip_printf("mult By A trans time: %1.2e seconds. \n",
              ABIP(tocq)(&mult_by_Atrans_timer) / 1e3);
#endif
}

// y += A*x
void ABIP(accum_by_A)(const ABIPMatrix *A, const abip_float *x, abip_float *y) {
  abip_int p;
  abip_int j;

  abip_int c1;
  abip_int c2;
  abip_float xj;

#if EXTRA_VERBOSE > 0
  ABIP(timer) mult_by_A_timer;
  ABIP(tic)(&mult_by_A_timer);
#endif

#ifdef _OPENMP
#pragma omp parallel for private(p, c1, c2, xj)
  for (j = 0; j < n; j++) {
    xj = x[j];
    c1 = A->p[j];
    c2 = A->p[j + 1];
    for (p = c1; p < c2; p++) {
#pragma omp atomic
      y[A->i[p]] += A->x[p] * xj;
    }
  }
#endif

  for (j = 0; j < A->n; j++) {
    xj = x[j];
    c1 = A->p[j];
    c2 = A->p[j + 1];
    for (p = c1; p < c2; p++) {
      y[A->i[p]] += A->x[p] * xj;
    }
  }

#if EXTRA_VERBOSE > 0
  abip_printf("mult By A time: %1.2e seconds \n",
              ABIP(tocq)(&mult_by_A_timer) / 1e3);
#endif
}

static abip_int _ldl_init(cs *A, abip_int *P, abip_float **info) {
  *info = (abip_float *)abip_calloc(AMD_INFO, sizeof(abip_float));
  return amd_order(A->n, A->p, A->i, P, (abip_float *)ABIP_NULL, *info);
}

cs *permute_kkt(spe_problem *spe) {
  abip_float *info;
  abip_int *Pinv, amd_status, *idx_mapping, i;
```

```
  cs *kkt = spe->L->K;
  cs *kkt_perm;
  if (!kkt) {
    return ABIP_NULL;
  }
  amd_status = _ldl_init(kkt, spe->L->P, &info);
  if (amd_status < 0) {
    abip_printf("AMD permutatation error.\n");
    return ABIP_NULL;
  }
#if VERBOSITY > 0
  abip_printf("Matrix factorization info:\n");
  amd_info(info);
#endif
  Pinv = cs_pinv(spe->L->P, spe->L->K->n);
  kkt_perm = cs_symperm(kkt, Pinv, 1);
  abip_free(Pinv);
  abip_free(info);
  return kkt_perm;
}

static void _ldl_perm(abip_int n, abip_float *x, abip_float *b, abip_int *P) {
  abip_int j;
  for (j = 0; j < n; j++) x[j] = b[P[j]];
}

static void _ldl_permt(abip_int n, abip_float *x, abip_float *b, abip_int *P) {
  abip_int j;
  for (j = 0; j < n; j++) x[P[j]] = b[j];
}

static void _ldl_solve(abip_float *b, cs *L, abip_float *Dinv, abip_int *P,
                       abip_float *bp) {
  /* solves PLDL'P' x = b for x */
  abip_int n = L->n;
  _ldl_perm(n, bp, b, P);
  QDLDL_solve(n, L->p, L->i, L->x, Dinv, bp);
  _ldl_permt(n, b, bp, P);
}

_MKL_DSS_HANDLE_t init_mkl_work(cs *K) {
  _INTEGER_t error;
  MKL_INT create_opt = MKL_DSS_ZERO_BASED_INDEXING;
  MKL_INT order_opt = MKL_DSS_DEFAULTS;
  MKL_INT sym = MKL_DSS_SYMMETRIC;
  MKL_INT type = MKL_DSS_INDEFINITE;

  _MKL_DSS_HANDLE_t handle;

  error = dss_create(handle, create_opt);
  if (error != MKL_DSS_SUCCESS) goto printError;

  error = dss_define_structure(handle, sym, K->p, K->m, K->n, K->i, K->p[K->n]);
  if (error != MKL_DSS_SUCCESS) goto printError;

  error = dss_reorder(handle, order_opt, 0);
  if (error != MKL_DSS_SUCCESS) goto printError;

  error = dss_factor_real(handle, type, K->x);
  if (error != MKL_DSS_SUCCESS) goto printError;

  return handle;
printError:
  printf("MKL-DSS returned error code %i\n", error);
  return -1;
}

abip_int mkl_solve_linsys(_MKL_DSS_HANDLE_t handle, abip_float *b, abip_int n) {
  _INTEGER_t error;
  abip_int nrhs = 1;
  abip_float *solValues = (abip_float *)abip_malloc(n * sizeof(abip_float));
  MKL_INT opt = MKL_DSS_DEFAULTS;
  error = dss_solve_real(handle, opt, b, nrhs, solValues);
  if (error != MKL_DSS_SUCCESS) {
    printf("solve err");
    exit(1);
  }

  memcpy(b, solValues, n * sizeof(abip_float));
  abip_free(solValues);
  return 0;
}

abip_int init_pardiso(spe_problem *self) {
  MKL_INT PARDISO_PARAMS_LDL[64] = {

      1,
```

```
    /* Non-default value */ 3,
    /* P Nested dissection */ 0, /* Reserved          */
    0,
    /* No CG            */ 0,
    /* No user permutation */ 0, /* Overwriting    */
    0,
    /* Refinement report */ 0,
    /* Auto ItRef step     */ 0, /* Reserved          */
    8,
    /* Perturb          */ 0,
    /* Disable scaling     */ 0, /* No transpose      */
    0,
    /* Disable matching  */ 0,
    /* Report on pivots    */ 0, /* Output            */
    0,
    /* Output           */ 0,
    /* Output              */ -1, /* No report         */
    0,
    /* No report        */ 0,
    /* Output              */ 2, /* Pivoting          */
    0,
    /* nPosEigVals      */ 0,
    /* nNegEigVals         */ 0, /* Classic factorize */
    0,
    0,
    0, /* Matrix checker    */
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    1,
    /* 0-based solve       */ 0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    /* No diagonal         */ 0,
    0,
    0,
    0,
    0,
    0,
    0,
    0};

cs *K = self->L->K;
MKL_INT n = K->n;
MKL_INT *ia = K->i;
MKL_INT *ja = K->p;
abip_float *a = K->x;

self->L->mtype = -2; /* Real symmetric matrix */
/* RHS and solution vectors. */
MKL_INT nrhs = 1; /* Number of right hand sides. */
/* Internal solver memory pointer pt, */
/* 32-bit: int pt[64]; 64-bit: long int pt[64] */
/* or void *pt[64] should be OK on both architectures */
// void *pt[64];
/* Pardiso control parameters. */
// MKL_INT iparm[64];
MKL_INT phase;
/* Auxiliary variables. */
MKL_INT i;

/* ------------------------------------*/
/* .. Setup Pardiso control parameters. */
/* ------------------------------------*/
for (i = 0; i < 64; i++) {
```

```
    self->L->iparm[i] = 0;
  }

  for (i = 0; i < 64; i++) {
    self->L->iparm[i] = PARDISO_PARAMS_LDL[i];
  }

  self->L->maxfct = 1; /* Maximum number of numerical factorizations. */
  self->L->mnum = 1;   /* Which factorization to use. */
  self->L->msglvl = 1; /* Print statistical information in file */
  self->L->error = 0;  /* Initialize error flag */
  /* -----------------------------------------------------------------*/
  /* .. Initialize the internal solver memory pointer. This is only  */
  /*   necessary for the FIRST call of the PARDISO solver.           */
  /* -----------------------------------------------------------------*/
  for (i = 0; i < 64; i++) {
    self->L->pt[i] = 0;
  }
  /* --------------------------------------------------------------------*/
  /* .. Reordering and Symbolic Factorization. This step also allocates  */
  /*    all memory that is necessary for the factorization.              */
  /* --------------------------------------------------------------------*/
  phase = 11;
  PARDISO(self->L->pt, &(self->L->maxfct), &(self->L->mnum), &(self->L->mtype),
          &phase, &n, a, ja, ia, &(self->L->idum), &nrhs, self->L->iparm,
          &(self->L->msglvl), &(self->L->ddum), &(self->L->ddum),
          &(self->L->error));
  if (self->L->error != 0) {
    printf("\nERROR during symbolic factorization: %i", self->L->error);
    exit(1);
  }
  printf("\nReordering completed ... ");
  printf("\nNumber of nonzeros in factors = %i", self->L->iparm[17]);
  printf("\nNumber of factorization MFLOPS = %i", self->L->iparm[18]);
  /* ----------------------------*/
  /* .. Numerical factorization. */
  /* ----------------------------*/
  phase = 22;
  PARDISO(self->L->pt, &(self->L->maxfct), &(self->L->mnum), &(self->L->mtype),
          &phase, &n, a, ja, ia, &(self->L->idum), &nrhs, self->L->iparm,
          &(self->L->msglvl), &(self->L->ddum), &(self->L->ddum),
          &(self->L->error));
  if (self->L->error != 0) {
    printf("\nERROR during numerical factorization: %i", self->L->error);
    exit(2);
  }
  printf("\nFactorization completed ... ");

  return 0;
}

abip_int pardiso_solve(spe_problem *self, abip_float *b, abip_int n) {
  /* ----------------------------------------------*/
  /* .. Back substitution and iterative refinement. */
  /* ----------------------------------------------*/
  cs *K = self->L->K;
  MKL_INT phase = 33;
  self->L->iparm[7] = 2; /* Max numbers of iterative refinement steps. */
  /* Set right hand side to one. */
  MKL_INT nrhs = 1; /* Number of right hand sides. */
  abip_float *x = (abip_float *)abip_malloc(n * sizeof(abip_float));

  PARDISO(self->L->pt, &self->L->maxfct, &self->L->mnum, &self->L->mtype,
          &phase, &n, K->x, K->p, K->i, &self->L->idum, &nrhs, self->L->iparm,
          &self->L->msglvl, b, x, &self->L->error);
  if (self->L->error != 0) {
    printf("\nERROR during solution: %i", self->L->error);
    exit(3);
  }
  memcpy(b, x, n * sizeof(abip_float));
  abip_free(x);

  return 0;
}

abip_int pardiso_free(spe_problem *self) {
  cs *K = self->L->K;
  MKL_INT phase = -1;
  MKL_INT nrhs = 1;
  MKL_INT n = K->n;
  PARDISO(self->L->pt, &self->L->maxfct, &self->L->mnum, &self->L->mtype,
          &phase, &n, &self->L->ddum, K->p, K->i, &self->L->idum, &nrhs,
          self->L->iparm, &self->L->msglvl, &self->L->ddum, &self->L->ddum,
          &self->L->error);

  cs_spfree(K);

```

```
  return 0;
}

abip_int LDL_factor(cs *A, cs **L, abip_float *Dinv) {
  // data for elim tree calculation
  QDLDL_int *etree;
  QDLDL_int *Lnz;
  QDLDL_int sumLnz;

  // working data for factorisation
  QDLDL_int *iwork;
  QDLDL_bool *bwork;
  QDLDL_float *fwork;

  /*--------------------------------
   * pre-factorisation memory allocations
   *---------------------------------*/

  // These can happen *before* the etree is calculated
  // since the sizes are not sparsity pattern specific

  // For the elimination tree
  etree = (QDLDL_int *)malloc(sizeof(QDLDL_int) * A->n);
  Lnz = (QDLDL_int *)malloc(sizeof(QDLDL_int) * A->n);

  // For the L factors.   Li and Lx are sparsity dependent
  // so must be done after the etree is constructed

  QDLDL_float *D = (QDLDL_float *)malloc(sizeof(QDLDL_float) * A->n);
  //    (*Dinv) = (QDLDL_float*)malloc(sizeof(QDLDL_float) * A->n);

  // Working memory.  Note that both the etree and factor
  // calls requires a working vector of QDLDL_int, with
  // the factor function requiring 3*An elements and the
  // etree only An elements.   Just allocate the larger
  // amount here and use it in both places
  iwork = (QDLDL_int *)malloc(sizeof(QDLDL_int) * (3 * A->n));
  bwork = (QDLDL_bool *)malloc(sizeof(QDLDL_bool) * A->n);
  fwork = (QDLDL_float *)malloc(sizeof(QDLDL_float) * A->n);

  /*--------------------------------
   * elimination tree calculation
   *---------------------------------*/
  sumLnz = QDLDL_etree(A->n, A->p, A->i, iwork, Lnz, etree);

  if (sumLnz < 0) {
    return sumLnz;  // error
  }
  /*--------------------------------
   * LDL factorisation
   *---------------------------------*/

  // First allocate memory for Li and Lx
  (*L) = cs_spalloc(A->n, A->n, sumLnz, 1, 0);
  //    L->p    = (QDLDL_int*)malloc(sizeof(QDLDL_int)*(A->n+1));
  //    L->i    = (QDLDL_int*)malloc(sizeof(QDLDL_int)*sumLnz);
  //    L->x    = (QDLDL_float*)malloc(sizeof(QDLDL_float)*sumLnz);
  //    Dinv  = (QDLDL_float*)malloc(sizeof(QDLDL_float)*A->n);

  // now factor
  abip_int status =
      QDLDL_factor(A->n, A->p, A->i, A->x, (*L)->p, (*L)->i, (*L)->x, D, Dinv,
                   Lnz, etree, bwork, iwork, fwork);

  free(D);
  free(etree);
  free(Lnz);
  free(iwork);
  free(bwork);
  free(fwork);

  return status;
}

abip_int abip_cholsol(spe_problem *self, abip_float *b, abip_int n) {
  abip_float *x;
  x = cs_malloc(n, sizeof(abip_float)); /* get workspace */
  abip_int ok = (self->L->S && self->L->N && x);
  if (ok) {
    cs_ipvec(self->L->S->pinv, b, x, n); /* x = P*b */
    cs_lsolve(self->L->N->L, x);         /* x = L\x */
    cs_ltsolve(self->L->N->L, x);        /* x = L'\x */
    cs_pvec(self->L->S->pinv, x, b, n);  /* b = P'*x */
  }
  cs_free(x);
  return (ok);
}
```

```
abip_int pcg(spe_problem *self, abip_float *b, abip_float *x, abip_float rho_x,
             abip_int max_iter, abip_float tol) {
  /*
      x is used for warm start
      result overwrite b
  */

  abip_int m = self->p;
  abip_int n = self->q;

  abip_float *ATx = (abip_float *)abip_malloc(n * sizeof(abip_float));
  memset(ATx, 0, n * sizeof(abip_float));
  self->spe_AT_times(self, x, ATx);
  ABIP(scale_array)(ATx, -1, n);

  abip_float *r = (abip_float *)abip_malloc(m * sizeof(abip_float));
  memcpy(r, b, m * sizeof(abip_float));
  self->spe_A_times(self, ATx, r);
  ABIP(add_scaled_array)(r, x, m, -rho_x);

  abip_float *z = (abip_float *)abip_malloc(m * sizeof(abip_float));
  for (int k = 0; k < m; k++) {
    z[k] = self->L->M[k] * r[k];
  }

  abip_float *p = (abip_float *)abip_malloc(m * sizeof(abip_float));
  memcpy(p, z, m * sizeof(abip_float));

  abip_float ip = ABIP(dot)(r, z, m);

  abip_int i;
  abip_float *Ap = (abip_float *)abip_malloc(m * sizeof(abip_float));
  abip_float alpha, ipold, beta;

  memcpy(b, x, m * sizeof(abip_float));

  for (i = 0; i < max_iter; i++) {
    memset(ATx, 0, n * sizeof(abip_float));
    self->spe_AT_times(self, p, ATx);
    memcpy(Ap, p, m * sizeof(abip_float));
    ABIP(scale_array)(Ap, rho_x, m);
    self->spe_A_times(self, ATx, Ap);

    alpha = ip / (ABIP(dot)(Ap, p, m));

    // ABIP(add_scaled_array)(x, p, m, alpha);
    ABIP(add_scaled_array)(b, p, m, alpha);

    ABIP(add_scaled_array)(r, Ap, m, -alpha);

    if (ABIP(norm)(r, m) < tol) {
#if EXTRA_VERBOSE > 0
      abip_printf(
          "CG took %d iterations to converge, resisdual %4f <= tolerance %4f\n",
          i, ABIP(norm)(r, m), tol);
#endif

      abip_free(ATx);
      abip_free(r);
      abip_free(z);
      abip_free(p);
      abip_free(Ap);

      return i + 1;
    }

    for (int j = 0; j < m; j++) {
      z[j] = self->L->M[j] * r[j];
    }

    ipold = ip;
    ip = ABIP(dot)(z, r, m);
    beta = ip / ipold;

    ABIP(scale_array)(p, beta, m);
    ABIP(add_scaled_array)(p, z, m, 1);
  }

  printf(
      "CG did not converge within %d iterations, resisdual %4f > tolerance "
      "%4f\n",
      max_iter, ABIP(norm)(r, m), tol);

  abip_free(ATx);
  abip_free(r);
  abip_free(z);
```

```
  abip_free(p);
  abip_free(Ap);

  return i + 1;
}

static void mat_vec(spe_problem *self, const abip_float *x, abip_float *y) {
  abip_int m = self->m;
  abip_int n = self->n;
  abip_int i;

  memcpy(y, x, n * sizeof(abip_float));

  for (i = 0; i < n; i++) {
    y[i] *= self->rho_dr[i + m];
  }

  if (self->Q != ABIP_NULL) {
    ABIP(accum_by_A)(self->Q, x, y);
  }

  abip_float *tem = (abip_float *)abip_malloc(sizeof(abip_float) * m);
  memset(tem, 0, m * sizeof(abip_float));
  ABIP(accum_by_A)(self->A, x, tem);
  for (i = 0; i < m; i++) {
    tem[i] /= self->rho_dr[i];
  }

  ABIP(accum_by_Atrans)(self->A, tem, y);

  abip_free(tem);
}

abip_int qcp_pcg(spe_problem *self, abip_float *b, abip_float *x,
                 abip_int max_iter, abip_float tol) {
  /*
      x is used for warm start
      result overwrite b
  */

  abip_int m = self->m;
  abip_int n = self->n;
  abip_int i, j;

  abip_float ztr, ztr_prev, alpha;
  abip_float *p =
      (abip_float *)abip_calloc(n, sizeof(abip_float)); /* cg direction */
  abip_float *Gp = (abip_float *)abip_calloc(
      n, sizeof(abip_float)); /* updated CG direction */
  abip_float *r =
      (abip_float *)abip_calloc(n, sizeof(abip_float)); /* cg residual */
  abip_float *z = (abip_float *)abip_calloc(n, sizeof(abip_float));
  ; /* for preconditioning */

  if (x == ABIP_NULL) {
    /* no warm_start, take x = 0 */
    /* r = b */
    memcpy(r, b, n * sizeof(abip_float));
    /* b = 0 */
    memset(b, 0, n * sizeof(abip_float));
  } else {
    /* r = Mat * s */
    mat_vec(self, x, r);
    /* r = Mat * s - b */
    ABIP(add_scaled_array)(r, b, n, -1.);
    /* r = b - Mat * s */
    ABIP(scale_array)(r, -1., n);
    /* b = s */
    memcpy(b, x, n * sizeof(abip_float));
  }
  /* check to see if we need to run CG at all */
  if (ABIP(norm_inf)(r, n) < MAX(tol, 1e-12)) {
    abip_free(p);
    abip_free(Gp);
    abip_free(r);
    abip_free(z);
    return 0;
  }

  /* z = M r (M is inverse preconditioner) */
  memcpy(z, r, n * sizeof(abip_float));
  ABIP(c_dot)(z, self->L->M, n);

  /* ztr = z'r */
  ztr = ABIP(dot)(z, r, n);
  /* p = z */
  memcpy(p, z, n * sizeof(abip_float));
```

```
  for (i = 0; i < max_iter; ++i) {
    /* Gp = Mat * p */
    mat_vec(self, p, Gp);
    /* alpha = z'r / p'G p */
    alpha = ztr / ABIP(dot)(p, Gp, n);
    /* b += alpha * p */
    ABIP(add_scaled_array)(b, p, n, alpha);
    /* r -= alpha * G p */
    ABIP(add_scaled_array)(r, Gp, n, -alpha);

    if (ABIP(norm_inf)(r, n) < tol) {
      break;
    }
    /* z = M r (M is inverse preconditioner) */
    memcpy(z, r, n * sizeof(abip_float));
    ABIP(c_dot)(z, self->L->M, n);

    ztr_prev = ztr;
    /* ztr = z'r */
    ztr = ABIP(dot)(z, r, n);
    /* p = beta * p, where beta = ztr / ztr_prev */
    ABIP(scale_array)(p, ztr / ztr_prev, n);
    /* p = z + beta * p */
    ABIP(add_scaled_array)(p, z, n, 1.);
  }

  printf("tol: %.4e, resid: %.4e, iters: %li\n", tol, ABIP(norm_inf)(r, n),
         (long)i + 1);
  if (i == max_iter - 1) {
    printf(
        "CG did not converge within %d iterations, resisdual %4f > tolerance "
        "%4f\n",
        max_iter, ABIP(norm)(r, m), tol);
  }

  abip_free(p);
  abip_free(Gp);
  abip_free(r);
  abip_free(z);

  return i + 1;
}

static void svm_mat_vec(spe_problem *self, const abip_float *x, abip_float *y) {
  abip_int m = self->p;
  abip_int n = self->q;
  abip_int i;

  memcpy(y, x, m * sizeof(abip_float));

  for (i = 0; i < m; i++) {
    y[i] *= self->rho_dr[i];
  }

  abip_float *tem = (abip_float *)abip_malloc(sizeof(abip_float) * n);
  memset(tem, 0, n * sizeof(abip_float));
  self->spe_AT_times(self, x, tem);

  abip_float *H = (abip_float *)abip_malloc(sizeof(abip_float) * n);
  for (i = 0; i < self->q; i++) {
    if (i < self->n)
      H[i] = self->stgs->rho_x + self->Q->x[i];
    else
      H[i] = self->stgs->rho_x;
  }

  for (i = 0; i < n; i++) {
    // tem[i] /= self->H[i];
    tem[i] /= H[i];
  }

  self->spe_A_times(self, tem, y);

  abip_free(tem);
  abip_free(H);
}

abip_int svmqp_pcg(spe_problem *self, abip_float *b, abip_float *x,
                   abip_int max_iter, abip_float tol) {
  /*
      x is used for warm start
      result overwrite b
  */

  // abip_int m = self->p;
  abip_int n = self->p;
```

```
abip_int i, j;

abip_float ztr, ztr_prev, alpha;
abip_float *p =
    (abip_float *)abip_calloc(n, sizeof(abip_float)); /* cg direction */
abip_float *Gp = (abip_float *)abip_calloc(
    n, sizeof(abip_float)); /* updated CG direction */
abip_float *r =
    (abip_float *)abip_calloc(n, sizeof(abip_float)); /* cg residual */
abip_float *z = (abip_float *)abip_calloc(n, sizeof(abip_float));
; /* for preconditioning */

if (x == ABIP_NULL) {
  /* no warm_start, take x = 0 */
  /* r = b */
  memcpy(r, b, n * sizeof(abip_float));
  /* b = 0 */
  memset(b, 0, n * sizeof(abip_float));
} else {
  /* r = Mat * s */
  svm_mat_vec(self, x, r);
  /* r = Mat * s - b */
  ABIP(add_scaled_array)(r, b, n, -1.);
  /* r = b - Mat * s */
  ABIP(scale_array)(r, -1., n);
  /* b = s */
  memcpy(b, x, n * sizeof(abip_float));
}
/* check to see if we need to run CG at all */
if (ABIP(norm_inf)(r, n) < MAX(tol, 1e-12)) {
  abip_free(p);
  abip_free(Gp);
  abip_free(r);
  abip_free(z);
  return 0;
}

/* z = M r (M is inverse preconditioner) */
memcpy(z, r, n * sizeof(abip_float));
ABIP(c_dot)(z, self->L->M, n);

/* ztr = z'r */
ztr = ABIP(dot)(z, r, n);
/* p = z */
memcpy(p, z, n * sizeof(abip_float));

for (i = 0; i < max_iter; ++i) {
  /* Gp = Mat * p */
  svm_mat_vec(self, p, Gp);
  /* alpha = z'r / p'G p */
  alpha = ztr / ABIP(dot)(p, Gp, n);
  /* b += alpha * p */
  ABIP(add_scaled_array)(b, p, n, alpha);
  /* r -= alpha * G p */
  ABIP(add_scaled_array)(r, Gp, n, -alpha);

  if (ABIP(norm_inf)(r, n) < tol) {
    break;
  }
  /* z = M r (M is inverse preconditioner) */
  memcpy(z, r, n * sizeof(abip_float));
  ABIP(c_dot)(z, self->L->M, n);

  ztr_prev = ztr;
  /* ztr = z'r */
  ztr = ABIP(dot)(z, r, n);
  /* p = beta * p, where beta = ztr / ztr_prev */
  ABIP(scale_array)(p, ztr / ztr_prev, n);
  /* p = z + beta * p */
  ABIP(add_scaled_array)(p, z, n, 1.);
}

printf("tol: %.4e, resid: %.4e, iters: %li\n", tol, ABIP(norm_inf)(r, n),
       (long)i + 1);
if (i == max_iter - 1) {
  printf(
      "CG did not converge within %d iterations, resisdual %4f > tolerance "
      "%4f\n",
      max_iter, ABIP(norm)(r, n), tol);
}

abip_free(p);
abip_free(Gp);
abip_free(r);
abip_free(z);

return i + 1;
```

```
}

abip_int init_dense_chol(spe_problem *spe) {
  // K is CSC format and only stores upper triangle
  spe->L->U = ABIP(csc_to_dense)(spe->L->K, ColMajor);
  abip_int info;
  abip_int n = spe->L->K->n;

  info = LAPACKE_dpotrf(LAPACK_COL_MAJOR, 'U', n, spe->L->U, n);
  return info;  // 0 if successful
}

abip_int dense_chol_sol(spe_problem *spe, abip_float *b, abip_int n) {
  abip_int info;

  info = LAPACKE_dpotrs(LAPACK_COL_MAJOR, 'U', n, 1, spe->L->U, n, b, n);
  return info;  // 0 if successful
}

abip_int dense_chol_free(spe_problem *spe) {
  if (spe->L->U) {
    abip_free(spe->L->U);
  }
  if (spe->L) {
    abip_free(spe->L);
  }

  return 0;
}
/*------------------------------------------------------------------*/

abip_int ABIP(init_linsys_work)(spe_problem *spe) {
  if (spe->stgs->linsys_solver == 3) {  // pcg
    spe->L->S = ABIP_NULL;
    spe->L->N = ABIP_NULL;
    spe->L->handle = ABIP_NULL;
    spe->L->Dinv = ABIP_NULL;
    spe->L->L = ABIP_NULL;
    spe->L->P = ABIP_NULL;
    spe->L->U = ABIP_NULL;

    spe->L->total_solve_time = 0.0;
    spe->L->total_cg_iters = 0;

    return 0;
  }

  cs *K = spe->L->K;
  spe->L->nnz_LDL = K->nzmax;  // K is NULL ptr if using pcg

  printf("\nStarting decomposition, with\nn = %d, m = %d , nnz = %d\n", K->m,
         K->n, K->nzmax);

  if (spe->stgs->linsys_solver == 0) {  // mkl_dss
    spe->L->handle = init_mkl_work(K);
    if (spe->L->handle == -1) {
      printf("\nerror in LDL factorization using MKL-DSS\n");
      return -1;
    }
    cs_spfree(K);
    spe->L->Dinv = ABIP_NULL;
    spe->L->L = ABIP_NULL;
    spe->L->P = ABIP_NULL;
    spe->L->M = ABIP_NULL;
    spe->L->S = ABIP_NULL;
    spe->L->N = ABIP_NULL;
    spe->L->U = ABIP_NULL;

    spe->L->total_solve_time = 0.0;
    return 0;
  }

  else if (spe->stgs->linsys_solver == 1) {  // qdldl

    spe->L->Dinv = (abip_float *)abip_malloc(K->n * sizeof(abip_float));
    spe->L->bp = (abip_float *)abip_malloc(K->n * sizeof(abip_float));
    spe->L->P = (abip_int *)abip_malloc(K->n * sizeof(abip_int));
    cs *kkt_perm = permute_kkt(spe);
    cs_spfree(K);
    if (LDL_factor(kkt_perm, &(spe->L->L), spe->L->Dinv) < 0) {
      spe->L->L = ABIP_NULL;
      printf("\nerror in LDL factorization using QDLDL\n");
      return -1;
    }
    cs_spfree(kkt_perm);

    spe->L->handle = ABIP_NULL;
```

```
    spe->L->M = ABIP_NULL;
    spe->L->S = ABIP_NULL;
    spe->L->N = ABIP_NULL;
    spe->L->U = ABIP_NULL;

    spe->L->total_solve_time = 0.0;
    return 0;

  } else if (spe->stgs->linsys_solver == 2) {  // cholesky
    spe->L->S = cs_schol(1, K);
    spe->L->N = cs_chol(K, spe->L->S);
    cs_spfree(K);
    spe->L->handle = ABIP_NULL;
    spe->L->Dinv = ABIP_NULL;
    spe->L->L = ABIP_NULL;
    spe->L->P = ABIP_NULL;
    spe->L->M = ABIP_NULL;
    spe->L->U = ABIP_NULL;

    spe->L->total_solve_time = 0.0;
    return 0;
  } else if (spe->stgs->linsys_solver == 4) {  // mkl_parsido
    init_pardiso(spe);
    spe->L->Dinv = ABIP_NULL;
    spe->L->handle = ABIP_NULL;
    spe->L->L = ABIP_NULL;
    spe->L->P = ABIP_NULL;
    spe->L->M = ABIP_NULL;
    spe->L->S = ABIP_NULL;
    spe->L->N = ABIP_NULL;
    spe->L->U = ABIP_NULL;

    spe->L->total_solve_time = 0.0;
    return 0;
  } else if (spe->stgs->linsys_solver == 5) {  // lapack dense chol

    init_dense_chol(spe);
    spe->L->handle = ABIP_NULL;
    spe->L->Dinv = ABIP_NULL;
    spe->L->L = ABIP_NULL;
    spe->L->P = ABIP_NULL;
    spe->L->M = ABIP_NULL;
    spe->L->S = ABIP_NULL;
    spe->L->N = ABIP_NULL;

    spe->L->total_solve_time = 0.0;
    return 0;
  }

  else {
    printf("\nlinsys solver type error\n");
    return -1;
  }
}

abip_int ABIP(solve_linsys)(spe_problem *spe, abip_float *b, abip_int n,
                            abip_float *pcg_warm_start, abip_float pcg_tol) {
  if (spe->stgs->linsys_solver == 0) {  // mkl_dss
    mkl_solve_linsys(spe->L->handle, b, n);
    return 0;
  } else if (spe->stgs->linsys_solver == 1) {  // qdldl
    // QDLDL_solve(spe->L->L->n,spe->L->L->p,spe->L->L->i,spe->L->L->x,spe->L->Dinv,b);
    // return 0;

    // for new ldl
    _ldl_solve(b, spe->L->L, spe->L->Dinv, spe->L->P, spe->L->bp);
    return 0;
  } else if (spe->stgs->linsys_solver == 2) {  // cholesky
    abip_cholsol(spe, b, n);
    return 0;
  } else if (spe->stgs->linsys_solver == 3) {  // PCG

    if (spe->stgs->prob_type == 3) {
      // return qcp_pcg(spe, b, pcg_warm_start, n, pcg_tol);
      return qcp_pcg(spe, b, pcg_warm_start, n, pcg_tol);
    } else if (spe->stgs->prob_type == 4) {
      return svmqp_pcg(spe, b, pcg_warm_start, n, pcg_tol);
    } else {
      return pcg(spe, b, pcg_warm_start, spe->stgs->rho_y, n, pcg_tol);
    }
  } else if (spe->stgs->linsys_solver == 4) {  // mkl_pardiso
    pardiso_solve(spe, b, n);
    return 0;
  } else if (spe->stgs->linsys_solver == 5) {  // lapack dense chol
    dense_chol_sol(spe, b, n);
    return 0;
  } else {
```

```
    printf("\nlinsys solver type error\n");
    return -1;
  }
}

abip_int ABIP(free_linsys)(spe_problem *spe) {
  if (spe->L) {
    if (spe->stgs->linsys_solver == 0) {  // mkl_dss
      MKL_INT opt = MKL_DSS_DEFAULTS;
      dss_delete(spe->L->handle, opt);
      return 0;
    } else if (spe->stgs->linsys_solver == 1) {  // qdldl
      if (spe->L->Dinv) abip_free(spe->L->Dinv);
      if (spe->L->L) cs_spfree(spe->L->L);
      if (spe->L->P) abip_free(spe->L->P);
      if (spe->L->bp) abip_free(spe->L->bp);
      return 0;
    } else if (spe->stgs->linsys_solver == 2) {  // cholesky
      if (spe->L->S) cs_sfree(spe->L->S);
      if (spe->L->N) cs_nfree(spe->L->N);
      return 0;
    } else if (spe->stgs->linsys_solver == 3) {  // pcg
      if (spe->L->M) abip_free(spe->L->M);
      return 0;
    } else if (spe->stgs->linsys_solver == 4) {  // mkl_pardiso
      pardiso_free(spe);
      return 0;
    } else if (spe->stgs->linsys_solver == 5) {  // lapack dense chol
      dense_chol_free(spe);
      return 0;
    }

    else {
      printf("\nlinsys solver type error\n");
      return -1;
    }
  }
}
```

4.189 source/qcp_config.c File Reference

`#include "qcp_config.h"`

Macros

- #define MIN_SCALE (1e-3)
- #define MAX_SCALE (1e3)

Functions

- abip_int init_qcp (qcp **self, ABIPData *d, ABIPSettings *stgs)

  Initialize the qcp problem structure.
- void qcp_A_times (qcp *self, const abip_float *x, abip_float *y)

  Matrix-vector multiplication for the general qcp problem with A untransposed.
- void qcp_AT_times (qcp *self, const abip_float *x, abip_float *y)

  Matrix-vector multiplication for the general qcp problem with A transposed.
- void scaling_qcp_data (qcp *self, ABIPCone *k)

  Scale the data for the qcp problem.
- void un_scaling_qcp_sol (qcp *self, ABIPSolution *sol)

  Get the unscaled solution of the general qcp problem.
- abip_float qcp_inner_conv_check (qcp *self, ABIPWork *w)

  Check whether the inner loop of the genral qcp problem has converged.

- void calc_qcp_residuals (qcp *self, ABIPWork *w, ABIPResiduals *r, abip_int ipm_iter, abip_int admm_iter)

    Calculate the residuals of the general qcp problem.
- cs * form_qcp_kkt (qcp *self)

    Formulate the qcp KKT matrix of the general qcp problem.
- void init_qcp_precon (qcp *self)

    Initialize the preconditioner of conjugate gradient method for the general qcp problem.
- abip_float get_qcp_pcg_tol (abip_int k, abip_float error_ratio, abip_float norm_p)

    Get the tolerance of the conjugate gradient method for the general qcp problem.
- abip_int init_qcp_linsys_work (qcp *self)

    Initialize the linear system solver work space for the general qcp problem.
- abip_int solve_qcp_linsys (qcp *self, abip_float *b, abip_float *pcg_warm_start, abip_int iter, abip_float error_ratio)

    Linear system solver for the general qcp problem.
- void free_qcp_linsys_work (qcp *self)

    Free the linear system solver work space for the general qcp problem.

4.189.1 Macro Definition Documentation

4.189.1.1 MAX_SCALE

```
#define MAX_SCALE (1e3)
```

Definition at line 3 of file qcp_config.c.

4.189.1.2 MIN_SCALE

```
#define MIN_SCALE (1e-3)
```

Definition at line 2 of file qcp_config.c.

4.189.2 Function Documentation

4.189.2.1 calc_qcp_residuals()

```
void calc_qcp_residuals (
            qcp * self,
            ABIPWork * w,
            ABIPResiduals * r,
            abip_int ipm_iter,
            abip_int admm_iter )
```

Calculate the residuals of the general qcp problem.

Definition at line 562 of file qcp_config.c.

4.189.2.2 form_qcp_kkt()

```
cs * form_qcp_kkt (
            qcp * self )
```

Formulate the qcp KKT matrix of the general qcp problem.

Definition at line 699 of file qcp_config.c.

4.189.2.3 free_qcp_linsys_work()

```
void free_qcp_linsys_work (
            qcp * self )
```

Free the linear system solver work space for the general qcp problem.

Definition at line 886 of file qcp_config.c.

4.189.2.4 get_qcp_pcg_tol()

```
abip_float get_qcp_pcg_tol (
            abip_int k,
            abip_float error_ratio,
            abip_float norm_p )
```

Get the tolerance of the conjugate gradient method for the general qcp problem.

Definition at line 786 of file qcp_config.c.

4.189.2.5 init_qcp()

```
abip_int init_qcp (
            qcp ** self,
            ABIPData * d,
            ABIPSettings * stgs )
```

Initialize the qcp problem structure.

Definition at line 8 of file qcp_config.c.

4.189.2.6 init_qcp_linsys_work()

```
abip_int init_qcp_linsys_work (
            qcp * self )
```

Initialize the linear system solver work space for the general qcp problem.

Definition at line 799 of file qcp_config.c.

4.189.2.7 init_qcp_precon()

```
void init_qcp_precon (
            qcp * self )
```

Initialize the preconditioner of conjugate gradient method for the general qcp problem.

Definition at line 754 of file qcp_config.c.

4.189.2.8 qcp_A_times()

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

Matrix-vector multiplication for the general qcp problem with A untransposed.

Definition at line 72 of file qcp_config.c.

4.189.2.9 qcp_AT_times()

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

Matrix-vector multiplication for the general qcp problem with A transposed.

Definition at line 82 of file qcp_config.c.

4.189.2.10 qcp_inner_conv_check()

```
abip_float qcp_inner_conv_check (
            qcp * self,
            ABIPWork * w )
```

Check whether the inner loop of the genral qcp problem has converged.

Definition at line 518 of file qcp_config.c.

4.189.2.11 scaling_qcp_data()

```
void scaling_qcp_data (
            qcp * self,
            ABIPCone * k )
```

Scale the data for the qcp problem.

Definition at line 91 of file qcp_config.c.

4.189.2.12 solve_qcp_linsys()

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

Linear system solver for the general qcp problem.

Definition at line 826 of file qcp_config.c.

4.189.2.13 un_scaling_qcp_sol()

```
void un_scaling_qcp_sol (
            qcp * self,
            ABIPSolution * sol )
```

Get the unscaled solution of the general qcp problem.

Definition at line 496 of file qcp_config.c.

4.190 qcp_config.c

Go to the documentation of this file.

```
#include "qcp_config.h"
#define MIN_SCALE (1e-3)
#define MAX_SCALE (1e3)

abip_int init_qcp(qcp **self, ABIPData *d, ABIPSettings *stgs) {
  qcp *this_qcp = (qcp *)abip_malloc(sizeof(qcp));
  *self = this_qcp;

  this_qcp->m = d->m;
  this_qcp->n = d->n;
  this_qcp->p = d->m;
  this_qcp->q = d->n;
  abip_int m = this_qcp->p;
  abip_int n = this_qcp->q;

  if (d->A == ABIP_NULL) {
    this_qcp->sparsity = 0;
  } else {
    this_qcp->sparsity = ((d->A->p[d->n] / (d->m * d->n)) < 0.05);
  }

  // non-identity DR scaling
  this_qcp->rho_dr =
      (abip_float *)abip_malloc((m + n + 1) * sizeof(abip_float));
  for (int i = 0; i < m + n + 1; i++) {
    if (i < m) {
      this_qcp->rho_dr[i] = stgs->rho_y;
    } else if (i < m + n) {
      this_qcp->rho_dr[i] = stgs->rho_x;
    } else {
      this_qcp->rho_dr[i] = stgs->rho_tau;
    }
  }

  this_qcp->L = (ABIPLinSysWork *)abip_malloc(sizeof(ABIPLinSysWork));
  this_qcp->pro_type = QCP;
  this_qcp->stgs = stgs;
  this_qcp->data = d;
  if (d->A != ABIP_NULL) {
    this_qcp->A = (ABIPMatrix *)abip_malloc(sizeof(ABIPMatrix));
  }
  if (d->Q != ABIP_NULL) {
    this_qcp->Q = (ABIPMatrix *)abip_malloc(sizeof(ABIPMatrix));
  }
  if (d->b != ABIP_NULL) {
    this_qcp->b = (abip_float *)abip_malloc(this_qcp->p * sizeof(abip_float));
  }
  this_qcp->c = (abip_float *)abip_malloc(this_qcp->q * sizeof(abip_float));
  this_qcp->D = (abip_float *)abip_malloc(this_qcp->m * sizeof(abip_float));
  this_qcp->E = (abip_float *)abip_malloc(this_qcp->n * sizeof(abip_float));

  this_qcp->scaling_data = &scaling_qcp_data;
  this_qcp->un_scaling_sol = &un_scaling_qcp_sol;
  this_qcp->calc_residuals = &calc_qcp_residuals;
  this_qcp->init_spe_linsys_work = &init_qcp_linsys_work;
  this_qcp->solve_spe_linsys = &solve_qcp_linsys;
  this_qcp->free_spe_linsys_work = &free_qcp_linsys_work;
  this_qcp->spe_A_times = &qcp_A_times;
  this_qcp->spe_AT_times = &qcp_AT_times;
  this_qcp->inner_conv_check = &qcp_inner_conv_check;

  return 0;
}

void qcp_A_times(qcp *self, const abip_float *x, abip_float *y) {
  if (self->A != ABIP_NULL) {
    ABIP(accum_by_A)(self->A, x, y);
  }
}

void qcp_AT_times(qcp *self, const abip_float *x, abip_float *y) {
  if (self->A != ABIP_NULL) {
    ABIP(accum_by_Atrans)(self->A, x, y);
  }
}

void scaling_qcp_data(qcp *self, ABIPCone *k) {
  if (self->data->b == ABIP_NULL) {
    self->b = ABIP_NULL;
  } else {
    memcpy(self->b, self->data->b, self->m * sizeof(abip_float));
  }
```

```
memcpy(self->c, self->data->c, self->n * sizeof(abip_float));

if (self->data->A == ABIP_NULL) {
  self->A = ABIP_NULL;
} else if (!ABIP(copy_A_matrix)(&(self->A), self->data->A)) {
  abip_printf("ERROR: copy A matrix failed\n");
  RETURN;
}

if (self->data->Q == ABIP_NULL) {
  self->Q = ABIP_NULL;
} else if (!ABIP(copy_A_matrix)(&(self->Q), self->data->Q)) {
  abip_printf("ERROR: copy Q matrix failed\n");
  RETURN;
}

abip_int m = self->m;
abip_int n = self->n;
ABIPMatrix *A = self->A;
ABIPMatrix *Q = self->Q;

abip_float min_row_scale = MIN_SCALE * SQRTF((abip_float)n);
abip_float max_row_scale = MAX_SCALE * SQRTF((abip_float)n);
abip_float min_col_scale = MIN_SCALE * SQRTF((abip_float)m);
abip_float max_col_scale = MAX_SCALE * SQRTF((abip_float)m);

abip_float *E_hat = self->E;
abip_float *D_hat = self->D;

for (int i = 0; i < n; i++) {
  E_hat[i] = 1;
}
for (int i = 0; i < m; i++) {
  D_hat[i] = 1;
}

abip_float *E = (abip_float *)abip_malloc(n * sizeof(abip_float));
memset(E, 0, n * sizeof(abip_float));

abip_float *E1 = (abip_float *)abip_malloc(n * sizeof(abip_float));
memset(E1, 0, n * sizeof(abip_float));

abip_float *E2 = (abip_float *)abip_malloc(n * sizeof(abip_float));
memset(E2, 0, n * sizeof(abip_float));

abip_float *D = (abip_float *)abip_malloc(m * sizeof(abip_float));
memset(D, 0, m * sizeof(abip_float));

abip_int origin_scaling = self->stgs->origin_scaling;
abip_int ruiz_scaling = self->stgs->ruiz_scaling;
abip_int pc_scaling = self->stgs->pc_scaling;
abip_int count;
abip_float mean_E;

if (A == ABIP_NULL && Q == ABIP_NULL) {
  origin_scaling = 0;
  ruiz_scaling = 0;
  pc_scaling = 0;
}

if (ruiz_scaling) {
  abip_int n_ruiz = 10;

  for (int ruiz_iter = 0; ruiz_iter < n_ruiz; ruiz_iter++) {
    count = 0;
    memset(E, 0, n * sizeof(abip_float));
    memset(E1, 0, n * sizeof(abip_float));
    memset(E2, 0, n * sizeof(abip_float));
    memset(D, 0, m * sizeof(abip_float));

    if (A != ABIP_NULL) {
      for (int j = 0; j < n; j++) {
        if (A->p[j] == A->p[j + 1]) {
          E1[j] = 0;
        } else {
          E1[j] =
              SQRTF(ABIP(norm_inf)(&A->x[A->p[j]], A->p[j + 1] - A->p[j]));
        }
      }
    }

    if (Q != ABIP_NULL) {
      for (int j = 0; j < n; j++) {
        if (Q->p[j] == Q->p[j + 1]) {
          E2[j] = 0;
        } else {
```

```
            E2[j] =
                SQRTF(ABIP(norm_inf)(&Q->x[Q->p[j]], Q->p[j + 1] - Q->p[j]));
          }
        }
      }

      for (int i = 0; i < n; i++) {
        E[i] = E1[i] < E2[i] ? E2[i] : E1[i];
      }

      if (k->q) {
        for (int i = 0; i < k->qsize; i++) {
          mean_E = ABIP(vec_mean)(&E[count], k->q[i]);
          for (int j = 0; j < k->q[i]; j++) {
            E[j + count] = mean_E;
          }
          count += k->q[i];
        }
      }

      if (k->rq) {
        for (int i = 0; i < k->rqsize; i++) {
          mean_E = ABIP(vec_mean)(&E[count], k->rq[i]);
          for (int j = 0; j < k->rq[i]; j++) {
            E[j + count] = mean_E;
          }
          count += k->rq[i];
        }
      }

      if (A != ABIP_NULL) {
        for (int i = 0; i < A->p[n]; i++) {
          if (D[A->i[i]] < ABS(A->x[i])) {
            D[A->i[i]] = ABS(A->x[i]);
          }
        }
        for (int i = 0; i < m; i++) {
          D[i] = SQRTF(D[i]);
          if (D[i] < min_row_scale)
            D[i] = 1;
          else if (D[i] > max_row_scale)
            D[i] = max_row_scale;
        }

        for (int i = 0; i < n; i++) {
          if (E[i] < min_col_scale)
            E[i] = 1;
          else if (E[i] > max_col_scale)
            E[i] = max_col_scale;
          for (int j = A->p[i]; j < A->p[i + 1]; j++) {
            A->x[j] /= E[i];
          }
        }
      }

      if (Q != ABIP_NULL) {
        for (int i = 0; i < n; i++) {
          for (int j = Q->p[i]; j < Q->p[i + 1]; j++) {
            Q->x[j] /= E[i];
          }
        }
        for (int i = 0; i < Q->p[n]; i++) {
          Q->x[i] /= E[Q->i[i]];
        }
      }

      if (A != ABIP_NULL) {
        for (int i = 0; i < A->p[n]; i++) {
          A->x[i] /= D[A->i[i]];
        }
      }

      for (int i = 0; i < n; i++) {
        E_hat[i] *= E[i];
      }

      for (int i = 0; i < m; i++) {
        D_hat[i] *= D[i];
      }
    }
  }

  if (origin_scaling) {
    memset(E, 0, n * sizeof(abip_float));
    memset(E1, 0, n * sizeof(abip_float));
    memset(E2, 0, n * sizeof(abip_float));
    memset(D, 0, m * sizeof(abip_float));
```

```
  count = 0;

  if (A != ABIP_NULL) {
    for (int i = 0; i < n; i++) {
      for (int j = A->p[i]; j < A->p[i + 1]; j++) {
        E1[i] += A->x[j] * A->x[j];
      }
      E1[i] = SQRTF(E1[i]);
    }
  }

  if (Q != ABIP_NULL) {
    for (int i = 0; i < n; i++) {
      for (int j = Q->p[i]; j < Q->p[i + 1]; j++) {
        E2[i] += Q->x[j] * Q->x[j];
      }
      E2[i] = SQRTF(E2[i]);
    }
  }

  for (int i = 0; i < n; i++) {
    E[i] = SQRTF(E1[i] < E2[i] ? E2[i] : E1[i]);
  }

  if (k->q) {
    for (int i = 0; i < k->qsize; i++) {
      mean_E = ABIP(vec_mean)(&E[count], k->q[i]);
      for (int j = 0; j < k->q[i]; j++) {
        E[j + count] = mean_E;
      }
      count += k->q[i];
    }
  }

  if (k->rq) {
    for (int i = 0; i < k->rqsize; i++) {
      mean_E = ABIP(vec_mean)(&E[count], k->rq[i]);
      for (int j = 0; j < k->rq[i]; j++) {
        E[j + count] = mean_E;
      }
      count += k->rq[i];
    }
  }

  if (A != ABIP_NULL) {
    for (int i = 0; i < A->p[n]; i++) {
      D[A->i[i]] += A->x[i] * A->x[i];
    }
    for (int i = 0; i < m; i++) {
      D[i] = SQRTF(SQRTF(D[i]));
      if (D[i] < min_row_scale)
        D[i] = 1;
      else if (D[i] > max_row_scale)
        D[i] = max_row_scale;
    }

    for (int i = 0; i < n; i++) {
      if (E[i] < min_col_scale)
        E[i] = 1;
      else if (E[i] > max_col_scale)
        E[i] = max_col_scale;
      for (int j = A->p[i]; j < A->p[i + 1]; j++) {
        A->x[j] /= E[i];
      }
    }
  }

  if (Q != ABIP_NULL) {
    for (int i = 0; i < n; i++) {
      for (int j = Q->p[i]; j < Q->p[i + 1]; j++) {
        Q->x[j] /= E[i];
      }
    }
    for (int i = 0; i < Q->p[n]; i++) {
      Q->x[i] /= E[Q->i[i]];
    }
  }

  if (A != ABIP_NULL) {
    for (int i = 0; i < A->p[n]; i++) {
      A->x[i] /= D[A->i[i]];
    }
  }

  for (int i = 0; i < n; i++) {
    E_hat[i] *= E[i];
```

```
    }

    for (int i = 0; i < m; i++) {
      D_hat[i] *= D[i];
    }
  }

  if (pc_scaling) {
    memset(E, 0, n * sizeof(abip_float));
    memset(E1, 0, n * sizeof(abip_float));
    memset(E2, 0, n * sizeof(abip_float));
    memset(D, 0, m * sizeof(abip_float));
    count = 0;
    abip_float alpha_pc = 1;
    if (A != ABIP_NULL) {
      for (int i = 0; i < n; i++) {
        for (int j = A->p[i]; j < A->p[i + 1]; j++) {
          E1[i] += POWF(ABS(A->x[j]), alpha_pc);
        }
        E1[i] = SQRTF(POWF(E1[i], 1 / alpha_pc));
      }
    }
    if (Q != ABIP_NULL) {
      for (int i = 0; i < n; i++) {
        for (int j = Q->p[i]; j < Q->p[i + 1]; j++) {
          E2[i] += POWF(ABS(Q->x[j]), alpha_pc);
        }
        E2[i] = SQRTF(POWF(E2[i], 1 / alpha_pc));
      }
    }

    for (int i = 0; i < n; i++) {
      E[i] = E1[i] < E2[i] ? E2[i] : E1[i];
    }

    if (k->q) {
      for (int i = 0; i < k->qsize; i++) {
        mean_E = ABIP(vec_mean)(&E[count], k->q[i]);
        for (int j = 0; j < k->q[i]; j++) {
          E[j + count] = mean_E;
        }
        count += k->q[i];
      }
    }

    if (k->rq) {
      for (int i = 0; i < k->rqsize; i++) {
        mean_E = ABIP(vec_mean)(&E[count], k->rq[i]);
        for (int j = 0; j < k->rq[i]; j++) {
          E[j + count] = mean_E;
        }
        count += k->rq[i];
      }
    }

    if (A != ABIP_NULL) {
      for (int i = 0; i < A->p[n]; i++) {
        D[A->i[i]] += POWF(ABS(A->x[i]), 2 - alpha_pc);
      }
      for (int i = 0; i < m; i++) {
        D[i] = SQRTF(POWF(D[i], 1 / (2 - alpha_pc)));
        if (D[i] < min_row_scale)
          D[i] = 1;
        else if (D[i] > max_row_scale)
          D[i] = max_row_scale;
      }

      for (int i = 0; i < n; i++) {
        if (E[i] < min_col_scale)
          E[i] = 1;
        else if (E[i] > max_col_scale)
          E[i] = max_col_scale;
        for (int j = A->p[i]; j < A->p[i + 1]; j++) {
          A->x[j] /= E[i];
        }
      }
    }

    if (Q != ABIP_NULL) {
      for (int i = 0; i < n; i++) {
        for (int j = Q->p[i]; j < Q->p[i + 1]; j++) {
          Q->x[j] /= E[i];
        }
      }
      for (int i = 0; i < Q->p[n]; i++) {
        Q->x[i] /= E[Q->i[i]];
      }
```

```
    }

    if (A != ABIP_NULL) {
      for (int i = 0; i < A->p[n]; i++) {
        A->x[i] /= D[A->i[i]];
      }
    }

    for (int i = 0; i < n; i++) {
      E_hat[i] *= E[i];
    }

    for (int i = 0; i < m; i++) {
      D_hat[i] *= D[i];
    }
  }

  abip_float sc =
      SQRTF(SQRTF(ABIP(norm_sq)(self->c, n) + ABIP(norm_sq)(self->b, m)));

  if (self->b != ABIP_NULL) {
    for (int i = 0; i < m; i++) {
      self->b[i] /= D_hat[i];
    }
  }

  for (int i = 0; i < n; i++) {
    self->c[i] /= E_hat[i];
  }

  if (sc < MIN_SCALE)
    sc = 1;
  else if (sc > MAX_SCALE)
    sc = MAX_SCALE;
  self->sc_b = 1 / sc;
  self->sc_c = 1 / sc;

  if (self->b != ABIP_NULL) {
    ABIP(scale_array)(self->b, self->sc_b * self->stgs->scale, m);
  }
  ABIP(scale_array)(self->c, self->sc_c * self->stgs->scale, n);

  abip_free(E);
  abip_free(E1);
  abip_free(E2);
  abip_free(D);
}

void un_scaling_qcp_sol(qcp *self, ABIPSolution *sol) {
  abip_int i;

  abip_float *D = self->D;
  abip_float *E = self->E;

  for (i = 0; i < self->n; ++i) {
    sol->x[i] /= (E[i] * self->sc_b);
  }

  for (i = 0; i < self->m; ++i) {
    sol->y[i] /= (D[i] * self->sc_c);
  }

  for (i = 0; i < self->n; ++i) {
    sol->s[i] *= E[i] / (self->sc_c * self->stgs->scale);
  }
}

abip_float qcp_inner_conv_check(qcp *self, ABIPWork *w) {
  abip_int m = self->p;
  abip_int n = self->q;

  abip_float *Qu = (abip_float *)abip_malloc((m + n + 1) * sizeof(abip_float));
  abip_float *Mu = (abip_float *)abip_malloc((m + n) * sizeof(abip_float));

  memset(Mu, 0, (m + n) * sizeof(abip_float));

  self->spe_A_times(self, &w->u[m], Mu);

  self->spe_AT_times(self, w->u, &Mu[m]);
  ABIP(scale_array)(&Mu[m], -1, n);

  if (self->Q != ABIP_NULL) {
    ABIP(accum_by_A)(self->Q, &w->u[m], &Mu[m]);
  }

  memcpy(Qu, Mu, (m + n) * sizeof(abip_float));

```

```c
  ABIP(add_scaled_array)(Qu, self->b, m, -w->u[m + n]);
  ABIP(add_scaled_array)(&Qu[m], self->c, n, w->u[m + n]);

  Qu[m + n] = -ABIP(dot)(w->u, Mu, m + n) / w->u[m + n] +
              ABIP(dot)(w->u, self->b, m) - ABIP(dot)(&w->u[m], self->c, n);

  abip_float *tem = (abip_float *)abip_malloc((m + n + 1) * sizeof(abip_float));
  memcpy(tem, Qu, (m + n + 1) * sizeof(abip_float));
  ABIP(add_scaled_array)(tem, w->v_origin, m + n + 1, -1);

  abip_float error_inner =
      ABIP(norm)(tem, m + n + 1) /
      (1 + ABIP(norm)(Qu, m + n + 1) + ABIP(norm)(w->v_origin, m + n + 1));

  abip_free(Qu);
  abip_free(Mu);
  abip_free(tem);

  return error_inner;
}

void calc_qcp_residuals(qcp *self, ABIPWork *w, ABIPResiduals *r,
                        abip_int ipm_iter, abip_int admm_iter) {
  DEBUG_FUNC

  abip_int n = w->n;
  abip_int m = w->m;

  abip_float *y = (abip_float *)abip_malloc(m * sizeof(abip_float));
  abip_float *x = (abip_float *)abip_malloc(n * sizeof(abip_float));
  abip_float *s = (abip_float *)abip_malloc(n * sizeof(abip_float));

  abip_float this_pr;
  abip_float this_dr;
  abip_float this_gap;

  if (admm_iter && r->last_admm_iter == admm_iter) {
    RETURN;
  }

  r->last_ipm_iter = ipm_iter;
  r->last_admm_iter = admm_iter;

  r->tau = ABS(w->u[n + m]);
  r->kap =
      ABS(w->v_origin[n + m]) /
      (self->stgs->normalize ? (self->stgs->scale * self->sc_c * self->sc_b)
                             : 1);

  memcpy(y, w->u, m * sizeof(abip_float));
  memcpy(x, &w->u[m], n * sizeof(abip_float));
  memcpy(s, &w->v_origin[m], n * sizeof(abip_float));

  ABIP(scale_array)(y, 1 / r->tau, m);
  ABIP(scale_array)(x, 1 / r->tau, n);
  ABIP(scale_array)(s, 1 / r->tau, n);

  abip_float *Ax = (abip_float *)abip_malloc(m * sizeof(abip_float));
  abip_float *Ax_b = (abip_float *)abip_malloc(m * sizeof(abip_float));

  memset(Ax, 0, m * sizeof(abip_float));
  self->spe_A_times(self, x, Ax);

  memcpy(Ax_b, Ax, m * sizeof(abip_float));
  ABIP(add_scaled_array)(Ax_b, self->b, m, -1);

  // for scs cg tol
  r->Ax_b_norm = ABIP(norm_inf)(Ax_b, m);

  ABIP(c_dot)(Ax, self->D, m);
  ABIP(c_dot)(Ax_b, self->D, m);

  this_pr = ABIP(norm_inf)(Ax_b, m) /
            (self->sc_b + MAX(ABIP(norm_inf)(Ax, m), self->sc_b * w->nm_inf_b));

  abip_float *Qx = (abip_float *)abip_malloc(n * sizeof(abip_float));
  abip_float *ATy = (abip_float *)abip_malloc(n * sizeof(abip_float));
  abip_float *Qx_ATy_c_s = (abip_float *)abip_malloc(n * sizeof(abip_float));

  memset(Qx, 0, n * sizeof(abip_float));
  abip_float xQx_2 = 0;

  if (self->Q != ABIP_NULL) {
    ABIP(accum_by_A)(self->Q, x, Qx);
    xQx_2 = ABIP(dot)(x, Qx, n) / (2 * self->sc_b * self->sc_c);
  }

```

```
  memset(ATy, 0, n * sizeof(abip_float));
  self->spe_AT_times(self, y, ATy);

  memcpy(Qx_ATy_c_s, Qx, n * sizeof(abip_float));
  ABIP(add_scaled_array)(Qx_ATy_c_s, ATy, n, -1);
  ABIP(add_scaled_array)(Qx_ATy_c_s, self->c, n, 1);
  ABIP(add_scaled_array)(Qx_ATy_c_s, s, n, -1);

  r->Qx_ATy_c_s_norm = ABIP(norm_inf)(Qx_ATy_c_s, n);

  ABIP(c_dot)(Qx, self->E, n);
  ABIP(c_dot)(ATy, self->E, n);
  ABIP(c_dot)(Qx_ATy_c_s, self->E, n);
  ABIP(c_dot)(s, self->E, n);

  this_dr = ABIP(norm_inf)(Qx_ATy_c_s, n) /
            (self->sc_c + MAX(self->sc_c * w->nm_inf_c, ABIP(norm_inf)(Qx, n)));

  abip_float cTx = ABIP(dot)(self->c, x, n) / (self->sc_b * self->sc_c);
  abip_float bTy = ABIP(dot)(self->b, y, m) / (self->sc_b * self->sc_c);

  this_gap = ABS(2 * xQx_2 + cTx - bTy) /
             (1 + MAX(2 * xQx_2, MAX(ABS(cTx), ABS(bTy))));

  r->pobj = xQx_2 + cTx;
  r->dobj = -xQx_2 + bTy;

  r->res_dif = MAX(MAX(ABS(this_pr - r->res_pri), ABS(this_dr - r->res_dual)),
                   ABS(this_gap - r->rel_gap));
  r->res_pri = this_pr;
  r->res_dual = this_dr;
  r->rel_gap = this_gap;
  r->error_ratio =
      MAX(r->res_pri / self->stgs->eps_p,
          MAX(r->res_dual / self->stgs->eps_d, r->rel_gap / self->stgs->eps_g));

  if (ABIP(dot)(self->c, &w->u[m], n) < 0) {
    ABIP(scale_array)(Qx, r->tau, n);
    ABIP(scale_array)(Ax, r->tau, m);
    r->res_unbdd = MAX(ABIP(norm)(Qx, n), ABIP(norm)(Ax, m)) /
                   (-ABIP(dot)(self->c, &w->u[m], n));
  } else {
    r->res_unbdd = INFINITY;
  }

  if (ABIP(dot)(self->b, w->u, m) > 0) {
    ABIP(scale_array)(ATy, r->tau, n);
    ABIP(scale_array)(s, r->tau, n);
    ABIP(add_scaled_array)(ATy, s, n, 1);

    r->res_infeas = ABIP(norm)(ATy, n) / ABIP(dot)(self->b, w->u, m);
  } else {
    r->res_infeas = INFINITY;
  }

  abip_free(x);
  abip_free(y);
  abip_free(s);
  abip_free(Ax);
  abip_free(Ax_b);
  abip_free(Qx);
  abip_free(ATy);
  abip_free(Qx_ATy_c_s);
}

/*K =  -rho_dr(1:m)*I            -A
                         Q + rho_dr(m+1:m+n)*I
*/
cs *form_qcp_kkt(qcp *self) {
  abip_int m = self->m;
  abip_int n = self->n;
  ABIPMatrix *A = self->A;
  ABIPMatrix *Q = self->Q;

  abip_int nnzA = A == ABIP_NULL ? 0 : A->p[A->n];
  ;
  abip_int nnzQ = Q == ABIP_NULL ? 0 : Q->p[Q->n];
  abip_int nnzK = m + nnzA + nnzQ + n;
  abip_int i;
  abip_int j;
  abip_float tem;

  cs *K = cs_spalloc(m + n, m + n, nnzK, 1, 1);

  for (i = 0; i < m; i++) {
    cs_entry(K, i, i, -self->rho_dr[i]);
  }
```

```

  if (A != ABIP_NULL) {
    for (i = 0; i < n; i++) {
      for (j = A->p[i]; j < A->p[i + 1]; j++) {
        cs_entry(K, A->i[j], m + i, -A->x[j]);
      }
    }
  }

  for (i = 0; i < n; i++) {
    if (Q == ABIP_NULL || Q->p[i] == Q->p[i + 1]) {
      cs_entry(K, m + i, m + i, self->rho_dr[m + i]);
    } else {
      for (j = Q->p[i]; j < Q->p[i + 1]; j++) {
        if (Q->i[j] > i)
          tem = 0;
        else if (Q->i[j] == i)
          tem = Q->x[j] + self->rho_dr[m + i];
        else
          tem = Q->x[j];
        cs_entry(K, m + Q->i[j], m + i, tem);
      }
    }
  }

  cs *K_csc = cs_compress(K);
  cs_spfree(K);
  cs_dropzeros(K_csc);

  return K_csc;
}

void init_qcp_precon(qcp *self) {
  self->L->M = (abip_float *)abip_malloc(self->q * sizeof(abip_float));
  memset(self->L->M, 0, self->q * sizeof(abip_float));

  for (int i = 0; i < self->q; i++) {
    for (int j = self->A->p[i]; j < self->A->p[i + 1]; j++) {
      self->L->M[i] +=
          self->A->x[j] * self->A->x[j] / self->rho_dr[self->A->i[j]];
    }
  }

  if (self->Q != ABIP_NULL) {
    for (int i = 0; i < self->q; i++) {
      for (int j = self->Q->p[i]; j < self->Q->p[i + 1]; j++) {
        if (i == self->Q->i[j]) {
          self->L->M[i] += self->Q->x[j];
          break;
        }
      }
    }
  }

  ABIP(add_scaled_array)(self->L->M, &self->rho_dr[self->p], self->q, 1);
  for (int i = 0; i < self->q; i++) {
    self->L->M[i] = 1.0 / self->L->M[i];
  }
}

abip_float get_qcp_pcg_tol(abip_int k, abip_float error_ratio,
                           abip_float norm_p) {
  if (k == -1) {
    return 1e-9 * norm_p;
  } else {
    return MAX(1e-9, 1e-5 * norm_p / POWF((k + 1), 2));
  }
}

abip_int init_qcp_linsys_work(qcp *self) {
  if (self->stgs->linsys_solver == 0) {  // mkl-dss need lower triangle
    self->L->K = cs_transpose(form_qcp_kkt(self), 1);
  } else if (self->stgs->linsys_solver == 1) {  // qdldl need upper triangle
    self->L->K = form_qcp_kkt(self);
  } else if (self->stgs->linsys_solver == 2) {  // cholesky need upper triangle
    self->L->K = form_qcp_kkt(self);
  } else if (self->stgs->linsys_solver == 3) {  // pcg doesn't need kkt matrix
    init_qcp_precon(self);
    self->L->K = ABIP_NULL;
  } else if (self->stgs->linsys_solver ==
             4) {  // mkl-pardiso need lower triangle
    self->L->K = cs_transpose(form_qcp_kkt(self), 1);
  } else if (self->stgs->linsys_solver ==
             5) {  //  dense cholesky need upper triangle
    self->L->K = form_qcp_kkt(self);
  } else {
    printf("\nlinsys solver type error\n");
```

```
    return -1;
  }

  return ABIP(init_linsys_work)(self);
}

abip_int solve_qcp_linsys(qcp *self, abip_float *b, abip_float *pcg_warm_start,
                          abip_int iter, abip_float error_ratio) {
  ABIP(timer) linsys_timer;
  ABIP(tic)(&linsys_timer);

  if (self->stgs->linsys_solver == 3) {  // pcg

    abip_int n = self->n;
    abip_int m = self->m;
    abip_int i;

    abip_float norm_p = ABIP(norm)(&b[m], n);

    abip_float *tem = (abip_float *)abip_malloc(sizeof(abip_float) * m);
    memcpy(tem, b, m * sizeof(abip_float));
    for (i = 0; i < m; i++) {
      tem[i] /= self->rho_dr[i];
    }
    self->spe_AT_times(self, tem, &b[m]);

    abip_free(tem);

    abip_float pcg_tol = get_qcp_pcg_tol(iter, error_ratio, norm_p);

    abip_int cg_its;
    if (iter == -1) {
      cg_its = ABIP(solve_linsys)(self, &b[m], n, pcg_warm_start, error_ratio);
    } else {
      cg_its =
          ABIP(solve_linsys)(self, &b[m], n, &pcg_warm_start[m], error_ratio);
    }

    if (iter >= 0) {
      self->L->total_cg_iters += cg_its;
    }

    ABIP(scale_array)(b, -1, m);
    self->spe_A_times(self, &b[m], b);
    for (i = 0; i < m; i++) {
      b[i] /= -self->rho_dr[i];
    }

  } else {  // direct methods

    abip_int n = self->n + self->m;

    // difference here
    ABIP(scale_array)(b, -1, self->m);

    ABIP(solve_linsys)(self, b, n, ABIP_NULL, 0);
  }

  self->L->total_solve_time += ABIP(tocq)(&linsys_timer);

  return 0;
}

void free_qcp_linsys_work(qcp *self) { ABIP(free_linsys)(self); }
```

4.191 source/svm_config.c File Reference

```
#include "svm_config.h"
```

Macros

- #define MIN_SCALE (1e-3)
- #define MAX_SCALE (1e3)

Functions

- abip_int init_svm (svm **self, ABIPData *d, ABIPSettings *stgs)

    Initialize the svm socp formulation structure.
- void svm_A_times (svm *self, const abip_float *x, abip_float *y)

    Customized matrix-vector multiplication for the svm socp formulation with A untransposed.
- void svm_AT_times (svm *self, const abip_float *x, abip_float *y)

    Customized matrix-vector multiplication for the svm socp formulation with A transposed.
- abip_float svm_inner_conv_check (svm *self, ABIPWork *w)

    Check whether the inner loop of the svm socp formulation has converged.
- void scaling_svm_data (svm *self, ABIPCone *k)

    Customized scaling procedure for the svm socp formulation.
- void un_scaling_svm_sol (svm *self, ABIPSolution *sol)

    Get the unscaled solution of the original svm problem.
- void calc_svm_residuals (svm *self, ABIPWork *w, ABIPResiduals *r, abip_int ipm_iter, abip_int admm_iter)

    Calculate the residuals of the svm socp formulation.
- cs * form_svm_kkt (svm *self)

    Formulate the qcp KKT matrix of the svm socp formulation.
- void init_svm_precon (svm *self)

    Initialize the preconditioner of conjugate gradient method for the svm socp formulation.
- abip_float get_svm_pcg_tol (abip_int k, abip_float error_ratio, abip_float norm_p)

    Get the tolerance of the conjugate gradient method for the svm socp formulation.
- abip_int init_svm_linsys_work (svm *self)

    Initialize the linear system solver work space for the svm socp formulation.
- abip_int solve_svm_linsys (svm *self, abip_float *b, abip_float *pcg_warm_start, abip_int iter, abip_float error_ratio)

    Customized linear system solver for the svm socp formulation.
- void free_svm_linsys_work (svm *self)

    Free the linear system solver work space for the svm socp formulation.

4.191.1 Macro Definition Documentation

4.191.1.1 MAX_SCALE

```
#define MAX_SCALE (1e3)
```

Definition at line 3 of file svm_config.c.

4.191.1.2 MIN_SCALE

```
#define MIN_SCALE (1e-3)
```

Definition at line 2 of file svm_config.c.

4.191.2 Function Documentation

4.191.2.1 calc_svm_residuals()

```
void calc_svm_residuals (
            svm * self,
            ABIPWork * w,
            ABIPResiduals * r,
            abip_int ipm_iter,
            abip_int admm_iter )
```

Calculate the residuals of the svm socp formulation.

Definition at line 445 of file svm_config.c.

4.191.2.2 form_svm_kkt()

```
cs * form_svm_kkt (
            svm * self )
```

Formulate the qcp KKT matrix of the svm socp formulation.

Definition at line 577 of file svm_config.c.

4.191.2.3 free_svm_linsys_work()

```
void free_svm_linsys_work (
            svm * self )
```

Free the linear system solver work space for the svm socp formulation.

Definition at line 812 of file svm_config.c.

4.191.2.4 get_svm_pcg_tol()

```
abip_float get_svm_pcg_tol (
            abip_int k,
            abip_float error_ratio,
            abip_float norm_p )
```

Get the tolerance of the conjugate gradient method for the svm socp formulation.

Definition at line 669 of file svm_config.c.

4.191.2.5 init_svm()

```
abip_int init_svm (
            svm ** self,
            ABIPData * d,
            ABIPSettings * stgs )
```

Initialize the svm socp formulation structure.

Definition at line 8 of file svm_config.c.

4.191.2.6 init_svm_linsys_work()

```
abip_int init_svm_linsys_work (
            svm * self )
```

Initialize the linear system solver work space for the svm socp formulation.

Definition at line 702 of file svm_config.c.

4.191.2.7 init_svm_precon()

```
void init_svm_precon (
            svm * self )
```

Initialize the preconditioner of conjugate gradient method for the svm socp formulation.

Definition at line 642 of file svm_config.c.

4.191.2.8 scaling_svm_data()

```
void scaling_svm_data (
            svm * self,
            ABIPCone * k )
```

Customized scaling procedure for the svm socp formulation.

Definition at line 283 of file svm_config.c.

4.191.2.9 solve_svm_linsys()

```
abip_int solve_svm_linsys (
            svm * self,
            abip_float * b,
            abip_float * pcg_warm_start,
            abip_int iter,
            abip_float error_ratio )
```

Customized linear system solver for the svm socp formulation.

Definition at line 728 of file svm_config.c.

4.191.2.10 svm_A_times()

```
void svm_A_times (
            svm * self,
            const abip_float * x,
            abip_float * y )
```

Customized matrix-vector multiplication for the svm socp formulation with A untransposed.

Definition at line 175 of file svm_config.c.

4.191.2.11 svm_AT_times()

```
void svm_AT_times (
            svm * self,
            const abip_float * x,
            abip_float * y )
```

Customized matrix-vector multiplication for the svm socp formulation with A transposed.

Definition at line 202 of file svm_config.c.

4.191.2.12 svm_inner_conv_check()

```
abip_float svm_inner_conv_check (
            svm * self,
            ABIPWork * w )
```

Check whether the inner loop of the svm socp formulation has converged.

Definition at line 234 of file svm_config.c.

4.191.2.13 un_scaling_svm_sol()

```
void un_scaling_svm_sol (
            svm * self,
            ABIPSolution * sol )
```

Get the unscaled solution of the original svm problem.

Definition at line 413 of file svm_config.c.

4.192 svm_config.c

Go to the documentation of this file.

```
#include "svm_config.h"
#define MIN_SCALE (1e-3)
#define MAX_SCALE (1e3)

abip_int init_svm(svm **self, ABIPData *d, ABIPSettings *stgs) {
  svm *this_svm = (svm *)abip_malloc(sizeof(svm));
  *self = this_svm;

  this_svm->m = d->m;
  this_svm->n = d->n;
  this_svm->p = d->m + d->n + 1;
  this_svm->q = 4 + 3 * d->n + 2 * d->m;
  this_svm->lambda = d->lambda;
  abip_int m = this_svm->p;
  abip_int n = this_svm->q;
  this_svm->Q = ABIP_NULL;
  this_svm->sparsity = (((abip_float)d->A->p[d->n] / (d->m * d->n)) < 0.05);

  // non-identity DR scaling
  this_svm->rho_dr =
      (abip_float *)abip_malloc((m + n + 1) * sizeof(abip_float));
  for (int i = 0; i < m + n + 1; i++) {
    if (i < m) {
      this_svm->rho_dr[i] = stgs->rho_y;
    } else if (i < m + n) {
      this_svm->rho_dr[i] = stgs->rho_x;
    } else {
      this_svm->rho_dr[i] = stgs->rho_tau;
    }
  }

  this_svm->L = (ABIPLinSysWork *)abip_malloc(sizeof(ABIPLinSysWork));
  this_svm->pro_type = SVM;
  this_svm->stgs = stgs;

  this_svm->data = (ABIPData *)abip_malloc(sizeof(ABIPData));

  abip_float *data_b =
      (abip_float *)abip_malloc(this_svm->p * sizeof(abip_float));
  memset(data_b, 0, this_svm->p * sizeof(abip_float));
  for (int i = 0; i < d->m + 1; i++) {
    data_b[i] = 1;
  }
  this_svm->data->b = data_b;

  abip_float *data_c =
      (abip_float *)abip_malloc(this_svm->q * sizeof(abip_float));
  memset(data_c, 0, this_svm->q * sizeof(abip_float));

  data_c[1] = 1;

  for (int i = 0; i < d->m; i++) {
    data_c[i + 4 + 3 * d->n] = 1;
  }

  this_svm->data->c = data_c;

  /* for svm problem, no need for inputing c*/
  d->c = (abip_float *)abip_malloc(this_svm->q * sizeof(abip_float));
  memcpy(d->c, data_c, this_svm->q * sizeof(abip_float));

  if ((d->m < 10 * d->n) && (10 * d->m > d->n)) {
    this_svm->sc = 1;
```

```
    this_svm->sc_c = MAX(0.45, POWF(7.5, (-log(2 * d->lambda) / log(10))) * 2);
    this_svm->sc_b = 1;
    this_svm->sc_cone1 = MAX(3, log(2 * d->lambda) / log(10) * 4 + 4);
    this_svm->sc_cone2 = this_svm->sc_cone1;
  } else if (10 * d->m < d->n) {
    this_svm->sc = 1;
    this_svm->sc_b = 1;
    this_svm->sc_cone2 = MAX(3, log(2 * d->lambda) / log(10) * 2 + 2);
    if (d->lambda >= 1) {
      this_svm->sc_c =
          MAX(0.2, POWF(0.2, (log(2 * d->lambda) / log(10))) * 7.5);
      this_svm->sc_cone1 = this_svm->sc_cone2;
    } else {
      this_svm->sc_c = POWF(0.3, log(2 * d->lambda) / log(10)) * 3;
      this_svm->sc_cone1 = MAX(0.4, log(2 * d->lambda) / log(10) * 0.2 + 0.8);
    }
  } else if (d->m > 10 * d->n) {
    if (d->n < 10) {
      this_svm->sc = 1;
      this_svm->sc_c = 1 / this_svm->lambda;
      // this_svm->sc_c = 1.0/d->n;
      this_svm->sc_b = 1;
      this_svm->sc_cone1 = 6;
      if (d->lambda < 0.002) {
        this_svm->sc_cone2 =
            this_svm->sc_cone2 - 3 * log(this_svm->lambda * 500) / log(10);
      }
    } else if (d->lambda >= 1) {
      this_svm->sc = 1;
      this_svm->sc_c = 1 / this_svm->lambda;
      this_svm->sc_b = 1;
      this_svm->sc_cone1 = 6;
      this_svm->sc_cone2 = this_svm->lambda;
    } else {
      this_svm->sc = 1;
      this_svm->sc_c = MIN(POWF(5, (-log(5 * d->lambda) / log(10))) * 4, 300);
      this_svm->sc_b = MAX(0.1, log(5 * d->lambda) / log(10) * 0.2 + 0.9);
      this_svm->sc_cone1 = MAX(0.05, log(5 * d->lambda) / log(10) * 0.3 + 0.7);
      this_svm->sc_cone2 = -log(5 * d->lambda) / log(10) * 2 + 6;
      if (d->lambda < 0.002) {
        this_svm->sc_cone2 =
            this_svm->sc_cone2 - 3 * log(this_svm->lambda * 500) / log(10);
      }
    }
  }

  ABIPMatrix *data_A = (ABIPMatrix *)abip_malloc(sizeof(ABIPMatrix));
  data_A->m = d->m;
  data_A->n = d->n + 1;
  data_A->p = (abip_int *)abip_malloc((data_A->n + 1) * sizeof(abip_int));
  data_A->i =
      (abip_int *)abip_malloc((d->A->p[d->n] + d->m) * sizeof(abip_int));
  data_A->x =
      (abip_float *)abip_malloc((d->A->p[d->n] + d->m) * sizeof(abip_float));
  for (int i = 0; i < d->A->p[d->n]; i++) {
    d->A->x[i] *= d->b[d->A->i[i]];
  }
  memcpy(data_A->p, d->A->p, (d->n + 1) * sizeof(abip_int));
  data_A->p[data_A->n] = d->A->p[d->n] + d->m;

  memcpy(data_A->i, d->A->i, d->A->p[d->n] * sizeof(abip_int));
  for (int i = d->A->p[d->n]; i < d->A->p[d->n] + d->m; i++) {
    data_A->i[i] = i - d->A->p[d->n];
  }

  memcpy(data_A->x, d->A->x, d->A->p[d->n] * sizeof(abip_float));
  for (int i = d->A->p[d->n]; i < d->A->p[d->n] + d->m; i++) {
    data_A->x[i] = d->b[i - d->A->p[d->n]];
  }
  this_svm->data->A = data_A;
  this_svm->A = (ABIPMatrix *)abip_malloc(sizeof(ABIPMatrix));
  this_svm->b = (abip_float *)abip_malloc(this_svm->p * sizeof(abip_float));
  this_svm->c = (abip_float *)abip_malloc(this_svm->q * sizeof(abip_float));
  this_svm->sc_D = (abip_float *)abip_malloc(this_svm->m * sizeof(abip_float));
  this_svm->sc_E =
      (abip_float *)abip_malloc((this_svm->n + 1) * sizeof(abip_float));
  this_svm->sc_F =
      (abip_float *)abip_malloc((this_svm->n) * sizeof(abip_float));

  this_svm->wA = (ABIPMatrix *)abip_malloc(sizeof(ABIPMatrix));
  this_svm->wy = (abip_float *)abip_malloc(this_svm->m * sizeof(abip_float));
  this_svm->wB = (abip_float *)abip_malloc(this_svm->m * sizeof(abip_float));
  this_svm->wC = (abip_float *)abip_malloc(this_svm->m * sizeof(abip_float));
  this_svm->wD = (abip_float *)abip_malloc((this_svm->n) * sizeof(abip_float));
  this_svm->wE = (abip_float *)abip_malloc((this_svm->n) * sizeof(abip_float));
  this_svm->wF = (abip_float *)abip_malloc(this_svm->m * sizeof(abip_float));
  this_svm->wG = (abip_float *)abip_malloc((this_svm->n) * sizeof(abip_float));
```

```
  this_svm->wH =
      (abip_float *)abip_malloc((this_svm->n + 1) * sizeof(abip_float));
  this_svm->wX = (ABIPMatrix *)abip_malloc(sizeof(ABIPMatrix));

  this_svm->scaling_data = &scaling_svm_data;
  this_svm->un_scaling_sol = &un_scaling_svm_sol;
  this_svm->calc_residuals = &calc_svm_residuals;
  this_svm->init_spe_linsys_work = &init_svm_linsys_work;
  this_svm->solve_spe_linsys = &solve_svm_linsys;
  this_svm->free_spe_linsys_work = &free_svm_linsys_work;
  this_svm->spe_A_times = &svm_A_times;
  this_svm->spe_AT_times = &svm_AT_times;
  this_svm->inner_conv_check = &svm_inner_conv_check;

  return 0;
}

void svm_A_times(svm *self, const abip_float *x, abip_float *y) {
  y[0] += x[0];
  abip_int m = self->m;
  abip_int n = self->n;

  abip_float *tmp = (abip_float *)abip_malloc(n * sizeof(abip_float));
  memcpy(tmp, &x[n + 2], n * sizeof(abip_float));
  ABIP(add_scaled_array)(tmp, &x[2 * n + 3], n, -1);
  ABIP(accum_by_A)(self->wA, tmp, &y[1]);

  ABIP(add_scaled_array)(&y[1], self->wy, m, (x[2 * n + 2] - x[3 * n + 3]));
  for (int i = 0; i < m; i++) {
    y[i + 1] +=
        (self->wB[i] * x[i + 3 * n + 4] - self->wC[i] * x[i + 3 * n + 4 + m]);
  }

  for (int i = 0; i < n; i++) {
    y[i + 1 + m] += (self->wD[i] * x[i + 2] - self->wE[i] * tmp[i]);
  }

  abip_free(tmp);
}

void svm_AT_times(svm *self, const abip_float *x, abip_float *y) {
  y[0] += x[0];
  abip_int m = self->m;
  abip_int n = self->n;

  for (int i = 0; i < n; i++) {
    y[i + 2] += self->wD[i] * x[i + m + 1];
  }

  abip_float *tmp = (abip_float *)abip_malloc(n * sizeof(abip_float));
  for (int i = 0; i < n; i++) {
    tmp[i] = -self->wE[i] * x[i + m + 1];
  }
  ABIP(accum_by_Atrans)(self->wA, &x[1], tmp);

  ABIP(add_scaled_array)(&y[n + 2], tmp, n, 1);
  ABIP(add_scaled_array)(&y[2 * n + 3], tmp, n, -1);

  y[2 * n + 2] += ABIP(dot)(self->wy, &x[1], m);
  y[3 * n + 3] -= ABIP(dot)(self->wy, &x[1], m);

  for (int i = 0; i < m; i++) {
    y[i + 3 * n + 4] += self->wB[i] * x[i + 1];
    y[i + 3 * n + 4 + m] -= self->wC[i] * x[i + 1];
  }

  abip_free(tmp);
}

abip_float svm_inner_conv_check(svm *self, ABIPWork *w) {
  DEBUG_FUNC

  abip_int m = w->m;
  abip_int n = w->n;
  abip_int l = m + n + 1;

  abip_float *y = (abip_float *)abip_malloc(m * sizeof(abip_float));
  memcpy(y, w->u, m * sizeof(abip_float));
  abip_float *x = (abip_float *)abip_malloc(n * sizeof(abip_float));
  memcpy(x, &w->u[m], n * sizeof(abip_float));
  abip_float *s = (abip_float *)abip_malloc(n * sizeof(abip_float));
  memcpy(s, &w->v_origin[m], n * sizeof(abip_float));

  abip_float tau = w->u[l - 1];
  abip_float kap = w->v_origin[l - 1];

  abip_float *row1 = (abip_float *)abip_malloc(m * sizeof(abip_float));
```

```
    memcpy(row1, self->b, m * sizeof(abip_float));
    ABIP(scale_array)(row1, -tau, m);
    self->spe_A_times(self, x, row1);

    abip_float *row2 = (abip_float *)abip_malloc(n * sizeof(abip_float));
    memcpy(row2, s, n * sizeof(abip_float));
    ABIP(add_scaled_array)(row2, self->c, n, -tau);
    self->spe_AT_times(self, y, row2);
    ABIP(scale_array)(row2, -1, n);

    abip_float *Pu_v = (abip_float *)abip_malloc(l * sizeof(abip_float));
    memcpy(Pu_v, row1, m * sizeof(abip_float));
    memcpy(&Pu_v[m], row2, n * sizeof(abip_float));
    Pu_v[l - 1] = ABIP(dot)(self->b, y, m) - ABIP(dot)(self->c, x, n) - kap;

    abip_float err_inner =
        ABIP(norm)(Pu_v, l) /
        (1 + SQRTF(ABIP(norm_sq)(w->u, l) + ABIP(norm_sq)(w->v_origin, l)));

    abip_free(x);
    abip_free(y);
    abip_free(s);
    abip_free(row1);
    abip_free(row2);
    abip_free(Pu_v);
    return err_inner;
  }

  void scaling_svm_data(svm *self, ABIPCone *k) {
    if (!ABIP(copy_A_matrix)(&(self->A), self->data->A)) {
      abip_printf("ERROR: copy A matrix failed\n");
      RETURN;
    }

    abip_int m = self->m;
    abip_int n = self->n + 1;
    ABIPMatrix *A = self->A;

    abip_float min_row_scale = MIN_SCALE * SQRTF((abip_float)n);
    abip_float max_row_scale = MAX_SCALE * SQRTF((abip_float)n);
    abip_float min_col_scale = MIN_SCALE * SQRTF((abip_float)m);
    abip_float max_col_scale = MAX_SCALE * SQRTF((abip_float)m);

    abip_float *E = self->sc_E;
    memset(E, 0, n * sizeof(abip_float));

    abip_float *D = self->sc_D;
    memset(D, 0, m * sizeof(abip_float));

    abip_float avg = 0;

    if (self->stgs->scale_E) {
      for (int i = 0; i < n; i++) {
        for (int j = A->p[i]; j < A->p[i + 1]; j++) {
          E[i] += A->x[j] * A->x[j];
        }
        E[i] = SQRTF(E[i]);
        avg += E[i];
      }

      avg /= n;

      for (int i = 0; i < n; i++) {
        E[i] = avg / E[i];
      }

      for (int i = 0; i < n; i++) {
        for (int j = A->p[i]; j < A->p[i + 1]; j++) {
          A->x[j] *= E[i];
        }
      }
    }

    for (int i = 0; i < A->p[n]; i++) {
      D[A->i[i]] += A->x[i] * A->x[i];
    }

    avg = 0;
    for (int i = 0; i < m; i++) {
      avg += SQRTF(D[i]);
    }

    avg /= m;

    for (int i = 0; i < m; i++) {
      D[i] = avg / SQRTF(D[i]);
    }
```

```

  for (int i = 0; i < A->p[n]; i++) {
    A->x[i] *= D[A->i[i]];
  }

  for (int i = 0; i < n - 1; i++) {
    self->sc_F[i] = 1 / SQRTF(1 + 2 * E[i] * E[i]);
  }

  memcpy(self->b, self->data->b, self->p * sizeof(abip_float));

  self->b[0] = self->sc_cone2;

  for (int i = 1; i < m + 1; i++) {
    self->b[i] = D[i - 1];
  }
  ABIP(scale_array)(self->b, self->sc_b, self->p);

  memcpy(self->c, self->data->c, self->q * sizeof(abip_float));
  self->c[0] = 0;
  self->c[1] = self->sc_c * self->sc_cone1 * self->sc_cone2;

  for (int i = 0; i < m; i++) {
    self->c[i + 4 + 3 * self->n] = self->lambda * self->sc_c / self->sc;
  }

  n -= 1;

  ABIP(copy_A_matrix)(&(self->wA), self->A);
  self->wA->n -= 1;

  memcpy(self->wy, &self->A->x[self->A->p[n]], m * sizeof(abip_float));

  memcpy(self->wB, self->sc_D, m * sizeof(abip_float));
  ABIP(scale_array)(self->wB, 1 / self->sc, m);

  memcpy(self->wC, self->sc_D, m * sizeof(abip_float));

  memcpy(self->wD, self->sc_F, n * sizeof(abip_float));
  ABIP(scale_array)(self->wD, -SQRTF(self->sc_cone1), n);

  for (int i = 0; i < n; i++) {
    self->wE[i] = self->sc_E[i] * self->sc_F[i];
  }

  for (int i = 0; i < m; i++) {
    self->wF[i] = self->wB[i] * self->wB[i] + self->wC[i] * self->wC[i] +
                  self->stgs->rho_y;
  }

  for (int i = 0; i < n; i++) {
    self->wG[i] = self->wD[i] * self->wD[i] + 2 * self->wE[i] * self->wE[i] +
                  self->stgs->rho_y;
  }

  for (int i = 0; i < n; i++) {
    self->wH[i] = 2 - 4 / self->wG[i] * self->wE[i] * self->wE[i];
  }
  self->wH[n] = 2;

  ABIP(copy_A_matrix)(&(self->wX), self->wA);
  for (int i = 0; i < n; i++) {
    for (int j = self->wX->p[i]; j < self->wX->p[i + 1]; j++) {
      self->wX->x[j] *= (-2 * self->wE[i]);
    }
  }
}

void un_scaling_svm_sol(svm *self, ABIPSolution *sol) {
  abip_int m = self->m;
  abip_int n = self->n;
  abip_float *x = sol->x;
  abip_float *y = sol->y;
  abip_float *s = sol->s;

  abip_float *w = (abip_float *)abip_malloc(n * sizeof(abip_float));
  memcpy(w, &x[n + 2], n * sizeof(abip_float));
  ABIP(add_scaled_array)(w, &x[2 * n + 3], n, -1);
  ABIP(c_dot)(w, self->sc_E, n);
  ABIP(scale_array)(w, 1 / self->sc_b, n);

  abip_float *b = (abip_float *)abip_malloc(sizeof(abip_float));
  b[0] = (x[2 * n + 2] - x[3 * n + 3]) * self->sc_E[n] / self->sc_b;

  abip_float *xi = (abip_float *)abip_malloc(m * sizeof(abip_float));
  memcpy(xi, &x[3 * n + 4], m * sizeof(abip_float));
  ABIP(scale_array)(xi, 1 / (self->sc_b * self->sc_c), m);
```

```
  abip_free(x);
  abip_free(y);
  abip_free(s);

  sol->x = w;
  sol->y = b;
  sol->s = xi;
}

void calc_svm_residuals(svm *self, ABIPWork *w, ABIPResiduals *r,
                        abip_int ipm_iter, abip_int admm_iter) {
  abip_int p = w->m;
  abip_int q = w->n;
  abip_int m = self->m;
  abip_int n = self->n;
  abip_float C = self->lambda;
  abip_float this_pr;
  abip_float this_dr;
  abip_float this_gap;

  r->tau = w->u[p + q];

  abip_float *w1 = (abip_float *)abip_malloc((n + 1) * sizeof(abip_float));
  memcpy(w1, &w->u[m + 2 * n + 3], n * sizeof(abip_float));
  ABIP(add_scaled_array)(w1, &w->u[m + 3 * n + 4], n, -1);
  for (int i = 0; i < n; i++) {
    w1[i] *= (self->sc_E[i] / (r->tau * self->sc_b));
  }

  abip_float b = (w->u[m + 3 * n + 3] - w->u[m + 4 * n + 4]) / r->tau *
                 self->sc_E[n] / self->sc_b;
  w1[n] = b;

  abip_float *xi = (abip_float *)abip_malloc(m * sizeof(abip_float));
  memcpy(xi, &w->u[m + 4 * n + 5], m * sizeof(abip_float));
  ABIP(scale_array)(xi, 1 / (r->tau * self->sc * self->sc_b), m);

  abip_float *t = (abip_float *)abip_malloc(m * sizeof(abip_float));
  memcpy(t, &w->u[2 * m + 4 * n + 5], m * sizeof(abip_float));
  ABIP(scale_array)(t, 1 / (r->tau * self->sc_b), m);

  abip_float *y = (abip_float *)abip_malloc(m * sizeof(abip_float));
  memcpy(y, &w->u[1], m * sizeof(abip_float));
  for (int i = 0; i < m; i++) {
    y[i] = y[i] / r->tau * self->sc_D[i] / self->sc_c;
  }

  abip_float *s2 = (abip_float *)abip_malloc(m * sizeof(abip_float));
  memcpy(s2, &w->v[2 * m + 4 * n + 5], m * sizeof(abip_float));
  ABIP(scale_array)(s2, 1 / (r->tau * self->sc_c), m);

  abip_float *s1 = (abip_float *)abip_malloc(m * sizeof(abip_float));
  memcpy(s1, &w->v[m + 4 * n + 5], m * sizeof(abip_float));
  ABIP(scale_array)(s1, self->sc / (r->tau * self->sc_c), m);

  abip_float *pr = (abip_float *)abip_malloc(m * sizeof(abip_float));
  memcpy(pr, xi, m * sizeof(abip_float));
  ABIP(accum_by_A)(self->data->A, w1, pr);
  for (int i = 0; i < m; i++) {
    pr[i] -= (t[i] + 1);
  }
  this_pr = ABIP(norm)(pr, m) / SQRTF(m);

  abip_float *dr = (abip_float *)abip_malloc(2 * m * sizeof(abip_float));
  for (int i = 0; i < m; i++) {
    dr[i] = y[i] - s2[i];
    dr[i + m] = y[i] + s1[i] - C;
  }
  this_dr = ABIP(norm)(dr, 2 * m) / (SQRTF(m) * C);

  ABIPMatrix *B = (ABIPMatrix *)abip_malloc(sizeof(ABIPMatrix));
  ABIP(copy_A_matrix)(&B, self->data->A);
  B->n -= 1;
  abip_float *BTy = (abip_float *)abip_malloc(n * sizeof(abip_float));
  memset(BTy, 0, n * sizeof(abip_float));
  ABIP(accum_by_Atrans)(B, y, BTy);

  r->dobj = 0;
  r->pobj = 0;
  for (int i = 0; i < m; i++) {
    r->dobj += y[i];
    r->pobj += C * xi[i];
  }

  r->pobj += 0.5 * ABIP(dot)(w1, w1, n);
  r->dobj -= 0.5 * ABIP(dot)(BTy, BTy, n);
```

```

  this_gap = r->dobj - r->pobj;
  this_gap = ABS(this_gap) / (1 + ABS(r->pobj));

  r->last_ipm_iter = ipm_iter;
  r->last_admm_iter = admm_iter;

  r->res_infeas = NAN;
  r->res_unbdd = NAN;

  r->res_dif = MAX(MAX(ABS(this_pr - r->res_pri), ABS(this_dr - r->res_dual)),
                   ABS(this_gap - r->rel_gap));
  r->res_pri = this_pr;
  r->res_dual = this_dr;
  r->rel_gap = this_gap;
  r->error_ratio =
      MAX(r->res_pri / self->stgs->eps_p,
          MAX(r->res_dual / self->stgs->eps_d, r->rel_gap / self->stgs->eps_g));

  if (ABIP(dot)(self->c, &w->u[p], q) < 0) {
    abip_float *Ax = (abip_float *)abip_malloc(p * sizeof(abip_float));
    memset(Ax, 0, p * sizeof(abip_float));
    self->spe_A_times(self, &w->u[p], Ax);
    r->res_unbdd = ABIP(norm)(Ax, p) / (-ABIP(dot)(self->c, &w->u[p], q));
    abip_free(Ax);
  } else {
    r->res_unbdd = INFINITY;
  }

  if (ABIP(dot)(self->b, w->u, p) > 0) {
    abip_float *ATy_s = (abip_float *)abip_malloc(q * sizeof(abip_float));
    memset(ATy_s, 0, q * sizeof(abip_float));
    self->spe_AT_times(self, w->u, ATy_s);
    ABIP(add_scaled_array)(ATy_s, &w->v_origin[p], q, 1);

    r->res_infeas = ABIP(norm)(ATy_s, q) / ABIP(dot)(self->b, w->u, p);
    abip_free(ATy_s);
  } else {
    r->res_infeas = INFINITY;
  }

  ABIP(free_A_matrix)(B);
  abip_free(w1);
  abip_free(xi);
  abip_free(t);
  abip_free(y);
  abip_free(s1);
  abip_free(s2);
  abip_free(pr);
  abip_free(dr);
}

cs *form_svm_kkt(svm *self) {
  abip_int n = self->n;
  abip_int m = self->m;
  cs *LTL;

  cs *N1 = cs_spalloc(m, n + 1, self->A->p[n + 1], 1, 0);
  memcpy(N1->i, self->A->i, self->A->p[n + 1] * sizeof(abip_int));
  memcpy(N1->p, self->A->p, (n + 2) * sizeof(abip_int));
  memcpy(N1->x, self->A->x, self->A->p[n + 1] * sizeof(abip_float));

  cs *N2 = cs_spalloc(m, n + 1, self->A->p[n + 1], 1, 0);
  memcpy(N2->i, self->A->i, self->A->p[n + 1] * sizeof(abip_int));
  memcpy(N2->p, self->A->p, (n + 2) * sizeof(abip_int));
  memcpy(N2->x, self->A->x, self->A->p[n + 1] * sizeof(abip_float));

  if (m > n + 1) {
    for (int i = 0; i < N1->p[n + 1]; i++) {
      N1->x[i] /= self->wF[N1->i[i]];
    }

    cs *diag = cs_spalloc(n + 1, n + 1, n + 1, 1, 1);

    for (int i = 0; i < n + 1; i++) {
      cs_entry(diag, i, i, 1 / self->wH[i]);
    }
    cs *diag_H = cs_compress(diag);
    cs_spfree(diag);

    LTL = cs_add(diag_H, cs_multiply(cs_transpose(N2, 1), N1), 1, 1);
    cs_spfree(diag_H);
  } else {
    for (int i = 0; i < N1->n; i++) {
      for (int j = N1->p[i]; j < N1->p[i + 1]; j++) {
        N1->x[j] *= self->wH[i];
      }
```

```
    }
    cs *diag = cs_spalloc(m, m, m, 1, 1);

    for (int i = 0; i < m; i++) {
      cs_entry(diag, i, i, self->wF[i]);
    }
    cs *diag_F = cs_compress(diag);
    cs_spfree(diag);

    LTL = cs_add(diag_F, cs_multiply(N1, cs_transpose(N2, 1)), 1, 1);

    cs_spfree(diag_F);
  }

  cs_spfree(N1);
  cs_spfree(N2);

  for (int i = 0; i < LTL->n; i++) {
    for (int j = LTL->p[i]; j < LTL->p[i + 1]; j++) {
      if (LTL->i[j] > i) LTL->x[j] = 0;
    }
  }
  cs_dropzeros(LTL);
  return LTL;
}

void init_svm_precon(svm *self) {
  self->L->M = (abip_float *)abip_malloc(self->p * sizeof(abip_float));
  memset(self->L->M, 0, self->p * sizeof(abip_float));

  abip_float *M = self->L->M;

  M[0] = 1 / (self->stgs->rho_y + 1);

  for (int i = 0; i < self->A->p[self->A->n]; i++) {
    M[self->A->i[i] + 1] += 2 * POWF(self->A->x[i], 2);
  }

  for (int i = 1; i < self->m + 1; i++) {
    M[i] = 1 / (self->stgs->rho_y + M[i] + POWF(self->wB[i - 1], 2) +
                POWF(self->sc_D[i - 1], 2));
  }

  for (int i = self->m + 1; i < self->m + self->n + 1; i++) {
    M[i] = 1 / (self->stgs->rho_y + 2 * POWF(self->wE[i - self->m + 1], 2) +
                POWF(self->wD[i - self->m + 1], 2));
  }
}

abip_float get_svm_pcg_tol(abip_int k, abip_float error_ratio,
                           abip_float norm_p) {
  if (k == -1) {
    return 1e-9 * norm_p;
  } else {
    if (error_ratio > 100000) {
      return MAX(1e-9, 3e-2 * norm_p / POWF((k + 1), 2));
    } else if (error_ratio > 30000) {
      return MAX(1e-9, 3e-2 * norm_p / POWF((k + 1), 2));
    } else if (error_ratio > 10000) {
      return MAX(1e-9, 3e-2 * norm_p / POWF((k + 1), 2));
    } else if (error_ratio > 3000) {
      return MAX(1e-9, 2.5e-2 * norm_p / POWF((k + 1), 2));
    } else if (error_ratio > 1000) {
      return MAX(1e-9, 2e-2 * norm_p / POWF((k + 1), 2));
    } else if (error_ratio > 300) {
      return MAX(1e-9, 1.6e-2 * norm_p / POWF((k + 1), 2));
    } else if (error_ratio > 100) {
      return MAX(1e-9, 1.3e-2 * norm_p / POWF((k + 1), 2));
    } else if (error_ratio > 30) {
      return MAX(1e-9, 1e-2 * norm_p / POWF((k + 1), 2));
    } else if (error_ratio > 10) {
      return MAX(1e-9, 7e-3 * norm_p / POWF((k + 1), 2));
    } else {
      return MAX(1e-9, 4e-3 * norm_p / POWF((k + 1), 2));
    }
  }
}

abip_int init_svm_linsys_work(svm *self) {
  if (self->stgs->linsys_solver == 0) {  // mkl-dss need lower triangle
    self->L->K = cs_transpose(form_svm_kkt(self), 1);
  } else if (self->stgs->linsys_solver == 1) {  // qdldl need upper triangle
    self->L->K = form_svm_kkt(self);
  } else if (self->stgs->linsys_solver == 2) {  // cholesky need upper triangle
    self->L->K = form_svm_kkt(self);
  } else if (self->stgs->linsys_solver == 3) {  // pcg doesn't need kkt matrix
    init_svm_precon(self);
```

```
    self->L->K = ABIP_NULL;
  } else if (self->stgs->linsys_solver ==
             4) {  // mkl-pardiso need lower triangle
    self->L->K = cs_transpose(form_svm_kkt(self), 1);
  } else if (self->stgs->linsys_solver ==
             5) {  //  dense cholesky need upper triangle
    self->L->K = form_svm_kkt(self);
  } else {
    printf("\nlinsys solver type error\n");
    return -1;
  }
  return ABIP(init_linsys_work)(self);
}

abip_int solve_svm_linsys(svm *self, abip_float *b, abip_float *pcg_warm_start,
                          abip_int iter, abip_float error_ratio) {
  ABIP(timer) linsys_timer;
  ABIP(tic)(&linsys_timer);

  ABIP(scale_array)(&b[self->p], -1, self->q);

  if (self->stgs->linsys_solver == 3) {  // pcg

    abip_int p = self->p;
    abip_float norm_p = ABIP(norm)(b, p);

    self->spe_A_times(self, &b[p], b);
    abip_float pcg_tol = get_svm_pcg_tol(iter, error_ratio, norm_p);
    abip_int cg_its = ABIP(solve_linsys)(self, b, p, pcg_warm_start, pcg_tol);

    if (iter >= 0) {
      self->L->total_cg_iters += cg_its;
    }
  } else {  // direct methods
    abip_int n = self->n;
    abip_int m = self->m;
    abip_float *b2 = (abip_float *)abip_malloc(self->p * sizeof(abip_float));
    memcpy(b2, b, self->p * sizeof(abip_float));
    self->spe_A_times(self, &b[self->p], b2);
    b[0] = b2[0] / (1 + self->stgs->rho_y);

    for (int i = 0; i < n; i++) {
      b[i + m + 1] = b2[i + m + 1] / self->wG[i];
    }

    abip_float *b3 = (abip_float *)abip_malloc(m * sizeof(abip_float));
    memcpy(b3, &b2[1], m * sizeof(abip_float));
    ABIP(scale_array)(b3, -1, m);
    ABIP(accum_by_A)(self->wX, &b[m + 1], b3);
    ABIP(scale_array)(b3, -1, m);

    abip_float *tmp = (abip_float *)abip_malloc((n + 1) * sizeof(abip_float));
    if (m > n + 1) {
      for (int i = 0; i < m; i++) {
        b3[i] /= self->wF[i];
      }
      memset(tmp, 0, (n + 1) * sizeof(abip_float));
      ABIP(accum_by_Atrans)(self->A, b3, tmp);
      ABIP(solve_linsys)(self, tmp, n + 1, ABIP_NULL, 0);

      abip_float *tmp2 = (abip_float *)abip_malloc(m * sizeof(abip_float));
      memset(tmp2, 0, m * sizeof(abip_float));

      ABIP(accum_by_A)(self->A, tmp, tmp2);
      for (int i = 0; i < m; i++) {
        tmp2[i] /= self->wF[i];
      }
      memcpy(&b[1], b3, m * sizeof(abip_float));
      ABIP(add_scaled_array)(&b[1], tmp2, m, -1);
      abip_free(tmp2);
    } else {
      ABIP(solve_linsys)(self, b3, m, ABIP_NULL, 0);
      memcpy(&b[1], b3, m * sizeof(abip_float));
    }

    memset(tmp, 0, (n + 1) * sizeof(abip_float));
    ABIP(accum_by_Atrans)(self->wX, &b[1], tmp);
    for (int i = 0; i < n; i++) {
      tmp[i] /= self->wG[i];
    }
    ABIP(add_scaled_array)(&b[m + 1], tmp, n, -1);

    abip_free(b2);
    abip_free(b3);
    abip_free(tmp);
  }

```

```
  ABIP(scale_array)(&b[self->p], -1, self->q);
  self->spe_AT_times(self, b, &b[self->p]);

  self->L->total_solve_time += ABIP(tocq)(&linsys_timer);

  return 0;
}

void free_svm_linsys_work(svm *self) { ABIP(free_linsys)(self); }
```

4.193 source/svm_qp_config.c File Reference

```
#include "svm_qp_config.h"
```

Macros

- #define MIN_SCALE (1e-3)
- #define MAX_SCALE (1e3)

Functions

- abip_int init_svmqp (svmqp **self, ABIPData *d, ABIPSettings *stgs)

  Initialize the svm qp formulation structure.
- void svmqp_A_times (svmqp *self, const abip_float *x, abip_float *y)

  Customized matrix-vector multiplication for the svm qp formulation with A untransposed.
- void svmqp_AT_times (svmqp *self, const abip_float *x, abip_float *y)

  Customized matrix-vector multiplication for the svm qp formulation with A transposed.
- abip_float svmqp_inner_conv_check (svmqp *self, ABIPWork *w)

  Check whether the inner loop of the svm qp formulation has converged.
- void scaling_svmqp_data (svmqp *self, ABIPCone *k)

  Customized scaling procedure for the svm qp formulation.
- void un_scaling_svmqp_sol (svmqp *self, ABIPSolution *sol)

  Get the unscaled solution of the original svm problem.
- void calc_svmqp_residuals (svmqp *self, ABIPWork *w, ABIPResiduals *r, abip_int ipm_iter, abip_int admm_iter)

  Calculate the residuals of the svm qp formulation.
- cs * form_svmqp_kkt (svmqp *self)

  Formulate the qcp KKT matrix of the svm qp formulation.
- void init_svmqp_precon (svmqp *self)

  Initialize the preconditioner of conjugate gradient method for the svm qp formulation.
- abip_float get_svmqp_pcg_tol (abip_int k, abip_float error_ratio, abip_float norm_p)

  Get the tolerance of the conjugate gradient method for the svm qp formulation.
- abip_int init_svmqp_linsys_work (svmqp *self)

  Initialize the linear system solver work space for the svm qp formulation.
- abip_int solve_svmqp_linsys (svmqp *self, abip_float *b, abip_float *pcg_warm_start, abip_int iter, abip_float error_ratio)

  Customized linear system solver for the svm qp formulation.
- void free_svmqp_linsys_work (svmqp *self)

  Free the linear system solver work space for the svm qp formulation.

4.193.1 Macro Definition Documentation

4.193.1.1 MAX_SCALE

```
#define MAX_SCALE (1e3)
```

Definition at line 3 of file svm_qp_config.c.

4.193.1.2 MIN_SCALE

```
#define MIN_SCALE (1e-3)
```

Definition at line 2 of file svm_qp_config.c.

4.193.2 Function Documentation

4.193.2.1 calc_svmqp_residuals()

```
void calc_svmqp_residuals (
            svmqp * self,
            ABIPWork * w,
            ABIPResiduals * r,
            abip_int ipm_iter,
            abip_int admm_iter )
```

Calculate the residuals of the svm qp formulation.

Definition at line 628 of file svm_qp_config.c.

4.193.2.2 form_svmqp_kkt()

```
cs * form_svmqp_kkt (
            svmqp * self )
```

Formulate the qcp KKT matrix of the svm qp formulation.

Definition at line 761 of file svm_qp_config.c.

4.193.2.3 free_svmqp_linsys_work()

```
void free_svmqp_linsys_work (
            svmqp * self )
```

Free the linear system solver work space for the svm qp formulation.

Definition at line 1002 of file svm_qp_config.c.

4.193.2.4 get_svmqp_pcg_tol()

```
abip_float get_svmqp_pcg_tol (
            abip_int k,
            abip_float error_ratio,
            abip_float norm_p )
```

Get the tolerance of the conjugate gradient method for the svm qp formulation.

Definition at line 855 of file svm_qp_config.c.

4.193.2.5 init_svmqp()

```
abip_int init_svmqp (
            svmqp ** self,
            ABIPData * d,
            ABIPSettings * stgs )
```

Initialize the svm qp formulation structure.

Definition at line 8 of file svm_qp_config.c.

4.193.2.6 init_svmqp_linsys_work()

```
abip_int init_svmqp_linsys_work (
            svmqp * self )
```

Initialize the linear system solver work space for the svm qp formulation.

Definition at line 868 of file svm_qp_config.c.

4.193.2.7 init_svmqp_precon()

```
void init_svmqp_precon (
            svmqp * self )
```

Initialize the preconditioner of conjugate gradient method for the svm qp formulation.

Definition at line 826 of file svm_qp_config.c.

4.193.2.8 scaling_svmqp_data()

```
void scaling_svmqp_data (
            svmqp * self,
            ABIPCone * k )
```

Customized scaling procedure for the svm qp formulation.

Definition at line 196 of file svm_qp_config.c.

4.193.2.9 solve_svmqp_linsys()

```
abip_int solve_svmqp_linsys (
            svmqp * self,
            abip_float * b,
            abip_float * pcg_warm_start,
            abip_int iter,
            abip_float error_ratio )
```

Customized linear system solver for the svm qp formulation.

Definition at line 894 of file svm_qp_config.c.

4.193.2.10 svmqp_A_times()

```
void svmqp_A_times (
            svmqp * self,
            const abip_float * x,
            abip_float * y )
```

Customized matrix-vector multiplication for the svm qp formulation with A untransposed.

Definition at line 128 of file svm_qp_config.c.

4.193.2.11 svmqp_AT_times()

```
void svmqp_AT_times (
            svmqp * self,
            const abip_float * x,
            abip_float * y )
```

Customized matrix-vector multiplication for the svm qp formulation with A transposed.

Definition at line 141 of file svm_qp_config.c.

4.193.2.12 svmqp_inner_conv_check()

```
abip_float svmqp_inner_conv_check (
            svmqp * self,
            ABIPWork * w )
```

Check whether the inner loop of the svm qp formulation has converged.

Definition at line 152 of file svm_qp_config.c.

4.193.2.13 un_scaling_svmqp_sol()

```
void un_scaling_svmqp_sol (
            svmqp * self,
            ABIPSolution * sol )
```

Get the unscaled solution of the original svm problem.

Definition at line 597 of file svm_qp_config.c.

4.194 svm_qp_config.c

Go to the documentation of this file.

```
#include "svm_qp_config.h"
#define MIN_SCALE (1e-3)
#define MAX_SCALE (1e3)

abip_int init_svmqp(svmqp **self, ABIPData *d, ABIPSettings *stgs) {
  svmqp *this_svm = (svmqp *)abip_malloc(sizeof(svmqp));
  *self = this_svm;

  this_svm->pro_type = SVMQP;
  this_svm->m = d->m;
  this_svm->n = d->n;
  this_svm->p = d->m;
  this_svm->q = 1 + d->n + 2 * d->m;
  this_svm->lambda = d->lambda;
  abip_int m = this_svm->p;
  abip_int n = this_svm->q;

  this_svm->Q = (ABIPMatrix *)abip_malloc(sizeof(ABIPMatrix));
  this_svm->Q->m = n;
  this_svm->Q->n = n;
  this_svm->Q->i = (abip_int *)abip_malloc(sizeof(abip_int) * d->n);
```

```
this_svm->Q->p = (abip_int *)abip_malloc(sizeof(abip_int) * (n + 1));
this_svm->Q->x = (abip_float *)abip_malloc(sizeof(abip_float) * d->n);

for (abip_int i = 0; i < d->n; i++) {
  this_svm->Q->i[i] = i;
  this_svm->Q->p[i] = i;
  this_svm->Q->x[i] = 1;
}

for (abip_int i = 0; i < 2 * d->m + 2; i++) {
  this_svm->Q->p[d->n + i] = d->n;
}

this_svm->sparsity = (((abip_float)d->A->p[d->n] / (d->m * d->n)) < 0.05);
this_svm->rho_dr =
    (abip_float *)abip_malloc((m + n + 1) * sizeof(abip_float));
for (int i = 0; i < m + n + 1; i++) {
  if (i < m) {
    this_svm->rho_dr[i] = stgs->rho_y;
  } else if (i < m + n) {
    this_svm->rho_dr[i] = stgs->rho_x;
  } else {
    this_svm->rho_dr[i] = stgs->rho_tau;
  }
}

this_svm->L = (ABIPLinSysWork *)abip_malloc(sizeof(ABIPLinSysWork));
this_svm->stgs = stgs;

this_svm->data = (ABIPData *)abip_malloc(sizeof(ABIPData));

abip_float *data_b = (abip_float *)abip_malloc(m * sizeof(abip_float));
memset(data_b, 0, m * sizeof(abip_float));
for (int i = 0; i < m; i++) {
  data_b[i] = 1;
}
this_svm->data->b = data_b;

abip_float *data_c = (abip_float *)abip_malloc(n * sizeof(abip_float));
memset(data_c, 0, this_svm->q * sizeof(abip_float));

for (int i = 0; i < d->m; i++) {
  data_c[i + d->n + 1] = 1.0 / (this_svm->m * this_svm->lambda);
}
this_svm->data->c = data_c;

d->c = (abip_float *)abip_malloc(n * sizeof(abip_float));
memcpy(d->c, data_c, n * sizeof(abip_float));

ABIPMatrix *data_A = (ABIPMatrix *)abip_malloc(sizeof(ABIPMatrix));
data_A->m = d->m;
data_A->n = d->n + 1;
data_A->p = (abip_int *)abip_malloc((data_A->n + 1) * sizeof(abip_int));
data_A->i =
    (abip_int *)abip_malloc((d->A->p[d->n] + d->m) * sizeof(abip_int));
data_A->x =
    (abip_float *)abip_malloc((d->A->p[d->n] + d->m) * sizeof(abip_float));
for (int i = 0; i < d->A->p[d->n]; i++) {
  d->A->x[i] *= d->b[d->A->i[i]];
}
memcpy(data_A->p, d->A->p, (d->n + 1) * sizeof(abip_int));
data_A->p[data_A->n] = d->A->p[d->n] + d->m;

memcpy(data_A->i, d->A->i, d->A->p[d->n] * sizeof(abip_int));
for (int i = d->A->p[d->n]; i < d->A->p[d->n] + d->m; i++) {
  data_A->i[i] = i - d->A->p[d->n];
}

memcpy(data_A->x, d->A->x, d->A->p[d->n] * sizeof(abip_float));
for (int i = d->A->p[d->n]; i < d->A->p[d->n] + d->m; i++) {
  data_A->x[i] = d->b[i - d->A->p[d->n]];
}
this_svm->data->A = data_A;
this_svm->A = (ABIPMatrix *)abip_malloc(sizeof(ABIPMatrix));
this_svm->b = (abip_float *)abip_malloc(this_svm->p * sizeof(abip_float));
this_svm->c = (abip_float *)abip_malloc(this_svm->q * sizeof(abip_float));
this_svm->D = (abip_float *)abip_malloc(this_svm->p * sizeof(abip_float));

this_svm->E = (abip_float *)abip_malloc(this_svm->q * sizeof(abip_float));
for (int i = 0; i < this_svm->q; i++) {
  this_svm->E[i] = 1.0;
}

this_svm->F = (abip_float *)abip_malloc(this_svm->m * sizeof(abip_float));
this_svm->H = (abip_float *)abip_malloc(this_svm->q * sizeof(abip_float));

this_svm->scaling_data = &scaling_svmqp_data;
```

```
  this_svm->un_scaling_sol = &un_scaling_svmqp_sol;
  this_svm->calc_residuals = &calc_svmqp_residuals;
  this_svm->init_spe_linsys_work = &init_svmqp_linsys_work;
  this_svm->solve_spe_linsys = &solve_svmqp_linsys;
  this_svm->free_spe_linsys_work = &free_svmqp_linsys_work;
  this_svm->spe_A_times = &svmqp_A_times;
  this_svm->spe_AT_times = &svmqp_AT_times;
  this_svm->inner_conv_check = &svmqp_inner_conv_check;

  return 0;
}

void svmqp_A_times(svmqp *self, const abip_float *x, abip_float *y) {
  ABIP(accum_by_A)(self->A, x, y);

  for (int i = 0; i < self->m; i++) {
    y[i] +=
        1 / self->D[i] * (x[self->n + 1 + i] - x[self->n + 1 + self->m + i]);
  }
}

void svmqp_AT_times(svmqp *self, const abip_float *x, abip_float *y) {
  ABIP(accum_by_Atrans)(self->A, x, y);
  for (int i = 0; i < self->m; i++) {
    y[self->n + 1 + i] += 1 / self->D[i] * x[i];
    y[self->n + 1 + self->m + i] -= 1 / self->D[i] * x[i];
  }
}

abip_float svmqp_inner_conv_check(svmqp *self, ABIPWork *w) {
  abip_int m = self->p;
  abip_int n = self->q;

  abip_float *Qu = (abip_float *)abip_malloc((m + n + 1) * sizeof(abip_float));
  abip_float *Mu = (abip_float *)abip_malloc((m + n) * sizeof(abip_float));

  memset(Mu, 0, (m + n) * sizeof(abip_float));

  self->spe_A_times(self, &w->u[m], Mu);

  self->spe_AT_times(self, w->u, &Mu[m]);
  ABIP(scale_array)(&Mu[m], -1, n);

  if (self->Q != ABIP_NULL) {
    ABIP(accum_by_A)(self->Q, &w->u[m], &Mu[m]);
  }

  memcpy(Qu, Mu, (m + n) * sizeof(abip_float));

  ABIP(add_scaled_array)(Qu, self->b, m, -w->u[m + n]);
  ABIP(add_scaled_array)(&Qu[m], self->c, n, w->u[m + n]);

  Qu[m + n] = -ABIP(dot)(w->u, Mu, m + n) / w->u[m + n] +
              ABIP(dot)(w->u, self->b, m) - ABIP(dot)(&w->u[m], self->c, n);

  abip_float *tem = (abip_float *)abip_malloc((m + n + 1) * sizeof(abip_float));
  memcpy(tem, Qu, (m + n + 1) * sizeof(abip_float));
  ABIP(add_scaled_array)(tem, w->v_origin, m + n + 1, -1);

  abip_float error_inner =
      ABIP(norm)(tem, m + n + 1) /
      (1 + ABIP(norm)(Qu, m + n + 1) + ABIP(norm)(w->v_origin, m + n + 1));

  abip_free(Qu);
  abip_free(Mu);
  abip_free(tem);

  return error_inner;
}

void scaling_svmqp_data(svmqp *self, ABIPCone *k) {
  memcpy(self->b, self->data->b, self->p * sizeof(abip_float));

  memcpy(self->c, self->data->c, self->q * sizeof(abip_float));

  ABIP(copy_A_matrix)(&(self->A), self->data->A);

  abip_int m = self->p;
  abip_int n = self->n + 1;
  ABIPMatrix *A = self->A;
  ABIPMatrix *Q = self->Q;

  abip_float min_row_scale = MIN_SCALE * SQRTF((abip_float)n);
  abip_float max_row_scale = MAX_SCALE * SQRTF((abip_float)n);
  abip_float min_col_scale = MIN_SCALE * SQRTF((abip_float)m);
  abip_float max_col_scale = MAX_SCALE * SQRTF((abip_float)m);
```

```
abip_float *E_hat = self->E;
abip_float *D_hat = self->D;

for (int i = 0; i < n; i++) {
  E_hat[i] = 1;
}
for (int i = 0; i < m; i++) {
  D_hat[i] = 1;
}

abip_float *E = (abip_float *)abip_malloc(n * sizeof(abip_float));
memset(E, 0, n * sizeof(abip_float));

abip_float *E1 = (abip_float *)abip_malloc(n * sizeof(abip_float));
memset(E1, 0, n * sizeof(abip_float));

abip_float *E2 = (abip_float *)abip_malloc(n * sizeof(abip_float));
memset(E2, 0, n * sizeof(abip_float));

abip_float *D = (abip_float *)abip_malloc(m * sizeof(abip_float));
memset(D, 0, m * sizeof(abip_float));

abip_int origin_scaling = self->stgs->origin_scaling;
abip_int ruiz_scaling = self->stgs->ruiz_scaling;
abip_int pc_scaling = self->stgs->pc_scaling;
abip_int count;
abip_float mean_E;

if (A == ABIP_NULL && Q == ABIP_NULL) {
  origin_scaling = 0;
  ruiz_scaling = 0;
  pc_scaling = 0;
}

if (ruiz_scaling) {
  abip_int n_ruiz = 10;

  for (int ruiz_iter = 0; ruiz_iter < n_ruiz; ruiz_iter++) {
    count = 0;
    memset(E, 0, n * sizeof(abip_float));
    memset(E1, 0, n * sizeof(abip_float));
    memset(E2, 0, n * sizeof(abip_float));
    memset(D, 0, m * sizeof(abip_float));

    if (A != ABIP_NULL) {
      for (int j = 0; j < n; j++) {
        if (A->p[j] == A->p[j + 1]) {
          E1[j] = 0;
        } else {
          E1[j] =
              SQRTF(ABIP(norm_inf)(&A->x[A->p[j]], A->p[j + 1] - A->p[j]));
        }
      }
    }

    if (Q != ABIP_NULL) {
      for (int j = 0; j < n; j++) {
        if (Q->p[j] == Q->p[j + 1]) {
          E2[j] = 0;
        } else {
          E2[j] =
              SQRTF(ABIP(norm_inf)(&Q->x[Q->p[j]], Q->p[j + 1] - Q->p[j]));
        }
      }
    }

    for (int i = 0; i < n; i++) {
      E[i] = E1[i] < E2[i] ? E2[i] : E1[i];
    }

    if (k->q) {
      for (int i = 0; i < k->qsize; i++) {
        mean_E = ABIP(vec_mean)(&E[count], k->q[i]);
        for (int j = 0; j < k->q[i]; j++) {
          E[j + count] = mean_E;
        }
        count += k->q[i];
      }
    }

    if (k->rq) {
      for (int i = 0; i < k->rqsize; i++) {
        mean_E = ABIP(vec_mean)(&E[count], k->rq[i]);
        for (int j = 0; j < k->rq[i]; j++) {
          E[j + count] = mean_E;
        }
        count += k->rq[i];
```

```
        }
      }

      if (A != ABIP_NULL) {
        for (int i = 0; i < A->p[n]; i++) {
          if (D[A->i[i]] < ABS(A->x[i])) {
            D[A->i[i]] = ABS(A->x[i]);
          }
        }
        for (int i = 0; i < m; i++) {
          D[i] = SQRTF(D[i]);
          if (D[i] < min_row_scale)
            D[i] = 1;
          else if (D[i] > max_row_scale)
            D[i] = max_row_scale;
        }

        for (int i = 0; i < n; i++) {
          if (E[i] < min_col_scale)
            E[i] = 1;
          else if (E[i] > max_col_scale)
            E[i] = max_col_scale;
          for (int j = A->p[i]; j < A->p[i + 1]; j++) {
            A->x[j] /= E[i];
          }
        }
      }

      if (Q != ABIP_NULL) {
        for (int i = 0; i < n; i++) {
          for (int j = Q->p[i]; j < Q->p[i + 1]; j++) {
            Q->x[j] /= E[i];
          }
        }
        for (int i = 0; i < Q->p[n]; i++) {
          Q->x[i] /= E[Q->i[i]];
        }
      }

      if (A != ABIP_NULL) {
        for (int i = 0; i < A->p[n]; i++) {
          A->x[i] /= D[A->i[i]];
        }
      }

      for (int i = 0; i < n; i++) {
        E_hat[i] *= E[i];
      }

      for (int i = 0; i < m; i++) {
        D_hat[i] *= D[i];
      }
    }
  }

  if (origin_scaling) {
    memset(E, 0, n * sizeof(abip_float));
    memset(E1, 0, n * sizeof(abip_float));
    memset(E2, 0, n * sizeof(abip_float));
    memset(D, 0, m * sizeof(abip_float));

    count = 0;

    if (A != ABIP_NULL) {
      for (int i = 0; i < n; i++) {
        for (int j = A->p[i]; j < A->p[i + 1]; j++) {
          E1[i] += A->x[j] * A->x[j];
        }
        E1[i] = SQRTF(E1[i]);
      }
    }

    if (Q != ABIP_NULL) {
      for (int i = 0; i < n; i++) {
        for (int j = Q->p[i]; j < Q->p[i + 1]; j++) {
          E2[i] += Q->x[j] * Q->x[j];
        }
        E2[i] = SQRTF(E2[i]);
      }
    }

    for (int i = 0; i < n; i++) {
      E[i] = SQRTF(E1[i] < E2[i] ? E2[i] : E1[i]);
    }

    if (k->q) {
      for (int i = 0; i < k->qsize; i++) {
```

```
        mean_E = ABIP(vec_mean)(&E[count], k->q[i]);
        for (int j = 0; j < k->q[i]; j++) {
          E[j + count] = mean_E;
        }
        count += k->q[i];
      }
    }

    if (k->rq) {
      for (int i = 0; i < k->rqsize; i++) {
        mean_E = ABIP(vec_mean)(&E[count], k->rq[i]);
        for (int j = 0; j < k->rq[i]; j++) {
          E[j + count] = mean_E;
        }
        count += k->rq[i];
      }
    }

    if (A != ABIP_NULL) {
      for (int i = 0; i < A->p[n]; i++) {
        D[A->i[i]] += A->x[i] * A->x[i];
      }
      for (int i = 0; i < m; i++) {
        D[i] = SQRTF(SQRTF(D[i]));
        if (D[i] < min_row_scale)
          D[i] = 1;
        else if (D[i] > max_row_scale)
          D[i] = max_row_scale;
      }

      for (int i = 0; i < n; i++) {
        if (E[i] < min_col_scale)
          E[i] = 1;
        else if (E[i] > max_col_scale)
          E[i] = max_col_scale;
        for (int j = A->p[i]; j < A->p[i + 1]; j++) {
          A->x[j] /= E[i];
        }
      }
    }

    if (Q != ABIP_NULL) {
      for (int i = 0; i < n; i++) {
        for (int j = Q->p[i]; j < Q->p[i + 1]; j++) {
          Q->x[j] /= E[i];
        }
      }
      for (int i = 0; i < Q->p[n]; i++) {
        Q->x[i] /= E[Q->i[i]];
      }
    }

    if (A != ABIP_NULL) {
      for (int i = 0; i < A->p[n]; i++) {
        A->x[i] /= D[A->i[i]];
      }
    }

    for (int i = 0; i < n; i++) {
      E_hat[i] *= E[i];
    }

    for (int i = 0; i < m; i++) {
      D_hat[i] *= D[i];
    }
  }

  if (pc_scaling) {
    memset(E, 0, n * sizeof(abip_float));
    memset(E1, 0, n * sizeof(abip_float));
    memset(E2, 0, n * sizeof(abip_float));
    memset(D, 0, m * sizeof(abip_float));
    count = 0;
    abip_float alpha_pc = 1;
    if (A != ABIP_NULL) {
      for (int i = 0; i < n; i++) {
        for (int j = A->p[i]; j < A->p[i + 1]; j++) {
          E1[i] += POWF(ABS(A->x[j]), alpha_pc);
        }
        E1[i] = SQRTF(POWF(E1[i], 1 / alpha_pc));
      }
    }
    if (Q != ABIP_NULL) {
      for (int i = 0; i < n; i++) {
        for (int j = Q->p[i]; j < Q->p[i + 1]; j++) {
          E2[i] += POWF(ABS(Q->x[j]), alpha_pc);
        }
```

```
        E2[i] = SQRTF(POWF(E2[i], 1 / alpha_pc));
      }
    }

    for (int i = 0; i < n; i++) {
      E[i] = E1[i] < E2[i] ? E2[i] : E1[i];
    }

    if (k->q) {
      for (int i = 0; i < k->qsize; i++) {
        mean_E = ABIP(vec_mean)(&E[count], k->q[i]);
        for (int j = 0; j < k->q[i]; j++) {
          E[j + count] = mean_E;
        }
        count += k->q[i];
      }
    }

    if (k->rq) {
      for (int i = 0; i < k->rqsize; i++) {
        mean_E = ABIP(vec_mean)(&E[count], k->rq[i]);
        for (int j = 0; j < k->rq[i]; j++) {
          E[j + count] = mean_E;
        }
        count += k->rq[i];
      }
    }

    if (A != ABIP_NULL) {
      for (int i = 0; i < A->p[n]; i++) {
        D[A->i[i]] += POWF(ABS(A->x[i]), 2 - alpha_pc);
      }
      for (int i = 0; i < m; i++) {
        D[i] = SQRTF(POWF(D[i], 1 / (2 - alpha_pc)));
        if (D[i] < min_row_scale)
          D[i] = 1;
        else if (D[i] > max_row_scale)
          D[i] = max_row_scale;
      }

      for (int i = 0; i < n; i++) {
        if (E[i] < min_col_scale)
          E[i] = 1;
        else if (E[i] > max_col_scale)
          E[i] = max_col_scale;
        for (int j = A->p[i]; j < A->p[i + 1]; j++) {
          A->x[j] /= E[i];
        }
      }
    }

    if (Q != ABIP_NULL) {
      for (int i = 0; i < n; i++) {
        for (int j = Q->p[i]; j < Q->p[i + 1]; j++) {
          Q->x[j] /= E[i];
        }
      }
      for (int i = 0; i < Q->p[n]; i++) {
        Q->x[i] /= E[Q->i[i]];
      }
    }

    if (A != ABIP_NULL) {
      for (int i = 0; i < A->p[n]; i++) {
        A->x[i] /= D[A->i[i]];
      }
    }

    for (int i = 0; i < n; i++) {
      E_hat[i] *= E[i];
    }

    for (int i = 0; i < m; i++) {
      D_hat[i] *= D[i];
    }
  }

  abip_float sc =
      SQRTF(SQRTF(ABIP(norm_sq)(self->c, self->q) + ABIP(norm_sq)(self->b, m)));

  if (self->b != ABIP_NULL) {
    for (int i = 0; i < m; i++) {
      self->b[i] /= D_hat[i];
    }
  }

  for (int i = 0; i < n; i++) {
```

```
    self->c[i] /= E_hat[i];
  }

  if (sc < MIN_SCALE)
    sc = 1;
  else if (sc > MAX_SCALE)
    sc = MAX_SCALE;
  self->sc_b = 1 / sc;
  self->sc_c = 1 / sc;

  if (self->b != ABIP_NULL) {
    ABIP(scale_array)(self->b, self->sc_b * self->stgs->scale, m);
  }
  ABIP(scale_array)(self->c, self->sc_c * self->stgs->scale, self->q);

  for (int i = 0; i < m; i++) {
    self->F[i] =
        self->stgs->rho_y + 2 / self->stgs->rho_x / POWF(self->D[i], 2);
  }

  for (int i = 0; i < self->q; i++) {
    if (i < self->n)
      self->H[i] = self->stgs->rho_x + self->Q->x[i];
    else
      self->H[i] = self->stgs->rho_x;
  }

  abip_free(E);
  abip_free(E1);
  abip_free(E2);
  abip_free(D);
}

void un_scaling_svmqp_sol(svmqp *self, ABIPSolution *sol) {
  abip_int m = self->m;
  abip_int n = self->n;
  abip_float *x = sol->x;
  abip_float *y = sol->y;
  abip_float *s = sol->s;

  abip_float *w = (abip_float *)abip_malloc(n * sizeof(abip_float));
  abip_float *b = (abip_float *)abip_malloc(sizeof(abip_float));
  abip_float *xi = (abip_float *)abip_malloc(m * sizeof(abip_float));

  for (int i = 0; i < self->q; ++i) {
    sol->x[i] /= (self->E[i] * self->sc_b);
  }

  memcpy(w, x, n * sizeof(abip_float));
  b[0] = x[n];
  memcpy(xi, &x[n + 1], m * sizeof(abip_float));

  abip_free(x);
  abip_free(y);
  abip_free(s);

  sol->x = w;
  sol->y = b;
  sol->s = xi;
}

void calc_svmqp_residuals(svmqp *self, ABIPWork *w, ABIPResiduals *r,
                          abip_int ipm_iter, abip_int admm_iter) {
  DEBUG_FUNC

  abip_int n = w->n;
  abip_int m = w->m;

  abip_float *y = (abip_float *)abip_malloc(m * sizeof(abip_float));
  abip_float *x = (abip_float *)abip_malloc(n * sizeof(abip_float));
  abip_float *s = (abip_float *)abip_malloc(n * sizeof(abip_float));

  abip_float this_pr;
  abip_float this_dr;
  abip_float this_gap;

  if (admm_iter && r->last_admm_iter == admm_iter) {
    RETURN;
  }

  r->last_ipm_iter = ipm_iter;
  r->last_admm_iter = admm_iter;

  r->tau = ABS(w->u[n + m]);
  r->kap =
      ABS(w->v_origin[n + m]) /
      (self->stgs->normalize ? (self->stgs->scale * self->sc_c * self->sc_b)
```

```
                              : 1);

    memcpy(y, w->u, m * sizeof(abip_float));
    memcpy(x, &w->u[m], n * sizeof(abip_float));
    memcpy(s, &w->v_origin[m], n * sizeof(abip_float));

    ABIP(scale_array)(y, 1 / r->tau, m);
    ABIP(scale_array)(x, 1 / r->tau, n);
    ABIP(scale_array)(s, 1 / r->tau, n);

    abip_float *Ax = (abip_float *)abip_malloc(m * sizeof(abip_float));
    abip_float *Ax_b = (abip_float *)abip_malloc(m * sizeof(abip_float));

    memset(Ax, 0, m * sizeof(abip_float));
    self->spe_A_times(self, x, Ax);

    memcpy(Ax_b, Ax, m * sizeof(abip_float));
    ABIP(add_scaled_array)(Ax_b, self->b, m, -1);

    r->Ax_b_norm = ABIP(norm_inf)(Ax_b, m);

    ABIP(c_dot)(Ax, self->D, m);
    ABIP(c_dot)(Ax_b, self->D, m);

    this_pr = ABIP(norm_inf)(Ax_b, m) /
              (self->sc_b + MAX(ABIP(norm_inf)(Ax, m), self->sc_b * w->nm_inf_b));

    abip_float *Qx = (abip_float *)abip_malloc(n * sizeof(abip_float));
    abip_float *ATy = (abip_float *)abip_malloc(n * sizeof(abip_float));
    abip_float *Qx_ATy_c_s = (abip_float *)abip_malloc(n * sizeof(abip_float));

    memset(Qx, 0, n * sizeof(abip_float));
    abip_float xQx_2 = 0;

    if (self->Q != ABIP_NULL) {
      ABIP(accum_by_A)(self->Q, x, Qx);
      xQx_2 = ABIP(dot)(x, Qx, n) / (2 * self->sc_b * self->sc_c);
    }

    memset(ATy, 0, n * sizeof(abip_float));
    self->spe_AT_times(self, y, ATy);

    memcpy(Qx_ATy_c_s, Qx, n * sizeof(abip_float));
    ABIP(add_scaled_array)(Qx_ATy_c_s, ATy, n, -1);
    ABIP(add_scaled_array)(Qx_ATy_c_s, self->c, n, 1);
    ABIP(add_scaled_array)(Qx_ATy_c_s, s, n, -1);

    r->Qx_ATy_c_s_norm = ABIP(norm_inf)(Qx_ATy_c_s, n);

    ABIP(c_dot)(Qx, self->E, n);
    ABIP(c_dot)(ATy, self->E, n);
    ABIP(c_dot)(Qx_ATy_c_s, self->E, n);
    ABIP(c_dot)(s, self->E, n);

    this_dr = ABIP(norm_inf)(Qx_ATy_c_s, n) /
              (self->sc_c + MAX(self->sc_c * w->nm_inf_c, ABIP(norm_inf)(Qx, n)));

    abip_float cTx = ABIP(dot)(self->c, x, n) / (self->sc_b * self->sc_c);
    abip_float bTy = ABIP(dot)(self->b, y, m) / (self->sc_b * self->sc_c);

    this_gap = ABS(2 * xQx_2 + cTx - bTy) /
               (1 + MAX(2 * xQx_2, MAX(ABS(cTx), ABS(bTy))));

    r->pobj = xQx_2 + cTx;
    r->dobj = -xQx_2 + bTy;

    r->res_dif = MAX(MAX(ABS(this_pr - r->res_pri), ABS(this_dr - r->res_dual)),
                     ABS(this_gap - r->rel_gap));
    r->res_pri = this_pr;
    r->res_dual = this_dr;
    r->rel_gap = this_gap;
    r->error_ratio =
        MAX(r->res_pri / self->stgs->eps_p,
            MAX(r->res_dual / self->stgs->eps_d, r->rel_gap / self->stgs->eps_g));

    if (ABIP(dot)(self->c, &w->u[m], n) < 0) {
      ABIP(scale_array)(Qx, r->tau, n);
      ABIP(scale_array)(Ax, r->tau, m);
      r->res_unbdd = MAX(ABIP(norm)(Qx, n), ABIP(norm)(Ax, m)) /
                     (-ABIP(dot)(self->c, &w->u[m], n));
    } else {
      r->res_unbdd = INFINITY;
    }

    if (ABIP(dot)(self->b, w->u, m) > 0) {
      ABIP(scale_array)(ATy, r->tau, n);
      ABIP(scale_array)(s, r->tau, n);
```

```
    ABIP(add_scaled_array)(ATy, s, n, 1);

    r->res_infeas = ABIP(norm)(ATy, n) / ABIP(dot)(self->b, w->u, m);
  } else {
    r->res_infeas = INFINITY;
  }

  abip_free(x);
  abip_free(y);
  abip_free(s);
  abip_free(Ax);
  abip_free(Ax_b);
  abip_free(Qx);
  abip_free(ATy);
  abip_free(Qx_ATy_c_s);
}

cs *form_svmqp_kkt(svmqp *self) {
  abip_int n = self->n + 1;
  abip_int m = self->m;
  cs *LTL;

  cs *B1 = cs_spalloc(m, n, self->A->p[n], 1, 0);
  memcpy(B1->i, self->A->i, self->A->p[n] * sizeof(abip_int));
  memcpy(B1->p, self->A->p, (n + 1) * sizeof(abip_int));
  memcpy(B1->x, self->A->x, self->A->p[n] * sizeof(abip_float));

  cs *B2 = cs_spalloc(m, n, self->A->p[n], 1, 0);
  memcpy(B2->i, B1->i, B1->p[n] * sizeof(abip_int));
  memcpy(B2->p, B1->p, (n + 1) * sizeof(abip_int));
  memcpy(B2->x, B1->x, B1->p[n] * sizeof(abip_float));

  if (m > n) {
    cs *T1 = cs_spalloc(n, n, n, 1, 1);

    for (int i = 0; i < n; i++) {
      cs_entry(T1, i, i, self->H[i]);
    }

    cs *eye = cs_compress(T1);

    cs_spfree(T1);

    for (int i = 0; i < B1->p[n]; i++) {
      B1->x[i] /= self->F[B1->i[i]];
    }

    LTL = cs_add(eye, cs_multiply(cs_transpose(B2, 1), B1), 1, 1);

    cs_spfree(eye);
  } else {
    for (int i = 0; i < B2->n; i++) {
      for (int j = B2->p[i]; j < B2->p[i + 1]; j++) {
        B2->x[j] /= self->H[i];
      }
    }
    cs *diag = cs_spalloc(m, m, m, 1, 1);

    for (int i = 0; i < m; i++) {
      cs_entry(diag, i, i, self->F[i]);
    }
    cs *diag_F = cs_compress(diag);
    cs_spfree(diag);

    LTL = cs_add(diag_F, cs_multiply(B2, cs_transpose(B1, 1)), 1, 1);
  }
  cs_spfree(B2);
  cs_spfree(B1);

  for (int i = 0; i < LTL->n; i++) {
    for (int j = LTL->p[i]; j < LTL->p[i + 1]; j++) {
      if (LTL->i[j] > i) LTL->x[j] = 0;
    }
  }
  cs_dropzeros(LTL);
  return LTL;
}

void init_svmqp_precon(svmqp *self) {
  abip_int i;

  self->L->M = (abip_float *)abip_malloc(self->p * sizeof(abip_float));
  memset(self->L->M, 0, self->p * sizeof(abip_float));

  for (i = 0; i < self->A->n; i++) {
    for (int j = self->A->p[i]; j < self->A->p[i + 1]; j++) {
      self->L->M[self->A->i[j]] += self->A->x[j] * self->A->x[j] / self->H[i];
```

```
    }
  }

  for (i = 0; i < self->p; i++) {
    self->L->M[i] += 1 / (self->D[i] * self->D[i] * self->H[self->A->n + i]);
    self->L->M[i] +=
        1 / (self->E[i] * self->E[i] * self->H[self->A->n + self->p + i]);
  }

  ABIP(add_scaled_array)(self->L->M, self->rho_dr, self->p, 1.0);

  for (i = 0; i < self->p; i++) {
    self->L->M[i] = 1.0 / self->L->M[i];
  }
}

abip_float get_svmqp_pcg_tol(abip_int k, abip_float error_ratio,
                             abip_float norm_p) {
  if (k == -1) {
    return 1e-9 * norm_p;
  } else {
    return MAX(1e-9, 1e-5 * norm_p / POWF((k + 1), 2));
  }
}

abip_int init_svmqp_linsys_work(svmqp *self) {
  if (self->stgs->linsys_solver == 0) {  // mkl need lower triangle
    self->L->K = cs_transpose(form_svmqp_kkt(self), 1);
  } else if (self->stgs->linsys_solver == 1) {  // qdldl need upper triangle
    self->L->K = form_svmqp_kkt(self);
  } else if (self->stgs->linsys_solver == 2) {  // cholesky need upper triangle
    self->L->K = form_svmqp_kkt(self);
  } else if (self->stgs->linsys_solver == 3) {  // pcg doesn't need kkt matrix
    init_svmqp_precon(self);
    self->L->K = ABIP_NULL;
  } else if (self->stgs->linsys_solver ==
             4) {  // mkl-pardiso need lower triangle
    self->L->K = cs_transpose(form_svmqp_kkt(self), 1);
  } else if (self->stgs->linsys_solver ==
             5) {  //  dense cholesky need upper triangle
    self->L->K = form_svmqp_kkt(self);
  } else {
    printf("\nlinsys solver type error\n");
    return -1;
  }
  return ABIP(init_linsys_work)(self);
}

abip_int solve_svmqp_linsys(svmqp *self, abip_float *b,
                            abip_float *pcg_warm_start, abip_int iter,
                            abip_float error_ratio) {
  ABIP(timer) linsys_timer;
  ABIP(tic)(&linsys_timer);
  abip_int n = self->n;
  abip_int m = self->m;

  abip_int p = self->p;
  abip_int q = self->q;

  if (self->stgs->linsys_solver == 3) {  // pcg

    abip_int n = self->q;
    abip_int m = self->p;
    abip_int i;

    abip_float norm_p = ABIP(norm)(&b[m], n);

    abip_float *tem = (abip_float *)abip_malloc(sizeof(abip_float) * m);
    memcpy(tem, b, m * sizeof(abip_float));
    for (i = 0; i < m; i++) {
      tem[i] /= self->rho_dr[i];
    }
    self->spe_AT_times(self, tem, &b[m]);

    abip_free(tem);

    abip_float pcg_tol = get_svmqp_pcg_tol(iter, error_ratio, norm_p);
    abip_int cg_its =
        ABIP(solve_linsys)(self, &b[m], n, &pcg_warm_start[m], pcg_tol);

    if (iter >= 0) {
      self->L->total_cg_iters += cg_its;
    }

    ABIP(scale_array)(b, -1, m);
    self->spe_A_times(self, &b[m], b);
    for (i = 0; i < m; i++) {
```

```
        b[i] /= -self->rho_dr[i];
      }
    } else {  // direct methods

      abip_float *b2 = (abip_float *)abip_malloc(p * sizeof(abip_float));
      memcpy(b2, b, p * sizeof(abip_float));
      abip_float *tmp = (abip_float *)abip_malloc(q * sizeof(abip_float));
      memcpy(tmp, &b[p], sizeof(abip_float) * q);

      for (int i = 0; i < q; i++) {
        tmp[i] /= -self->H[i];
      }

      self->spe_A_times(self, tmp, b2);

      if (m > n + 1 && self->stgs->linsys_solver != 3) {
        for (int i = 0; i < p; i++) {
          b2[i] /= self->F[i];
        }

        abip_float *tmp1 =
            (abip_float *)abip_malloc((n + 1) * sizeof(abip_float));
        memset(tmp1, 0, (n + 1) * sizeof(abip_float));

        ABIP(accum_by_Atrans)(self->A, b2, tmp1);

        ABIP(solve_linsys)(self, tmp1, n + 1, ABIP_NULL, 0);

        abip_float *tmp2 = (abip_float *)abip_malloc(m * sizeof(abip_float));
        memset(tmp2, 0, m * sizeof(abip_float));

        ABIP(accum_by_A)(self->A, tmp1, tmp2);

        for (int i = 0; i < m; i++) {
          tmp2[i] /= self->F[i];
        }

        ABIP(add_scaled_array)(b2, tmp2, m, -1);

        abip_free(tmp1);
        abip_free(tmp2);
      } else {
        abip_int cg_its =
            ABIP(solve_linsys)(self, b2, m, pcg_warm_start, error_ratio);
        if (iter >= 0) {
          self->L->total_cg_iters += cg_its;
        }
      }

      memcpy(b, b2, m * sizeof(abip_float));
      abip_free(b2);
      abip_free(tmp);

    }  // if pcg

    self->spe_AT_times(self, b, &b[p]);

    for (int i = 0; i < q; i++) {
      b[p + i] /= self->H[i];
    }

    self->L->total_solve_time += ABIP(tocq)(&linsys_timer);

    return 0;
}

void free_svmqp_linsys_work(svmqp *self) { ABIP(free_linsys)(self); }
```

4.195 source/util.c File Reference

```
#include "util.h"
#include "glbopts.h"
#include "linsys.h"
```

Macros

- #define _CRT_SECURE_NO_WARNINGS

Functions

- void ABIP() tic (ABIP(timer) *t)
- abip_float ABIP() tocq (ABIP(timer) *t)
- abip_float ABIP() toc (ABIP(timer) *t)
- abip_float ABIP() str_toc (char *str, ABIP(timer) *t)
- void ABIP() print_work (const ABIPWork *w)
- void ABIP() print_data (const ABIPData *d)
- void ABIP() print_array (const abip_float *arr, abip_int n, const char *name)
- void ABIP() free_info (ABIPInfo *info)
- void ABIP() free_cone (ABIPCone *k)
- void ABIP() free_data (ABIPData *d)
- void ABIP() free_sol (ABIPSolution *sol)
- void ABIP() set_default_settings (ABIPData *d)

    Default parameter settings.

4.195.1 Macro Definition Documentation

4.195.1.1 _CRT_SECURE_NO_WARNINGS

```
#define _CRT_SECURE_NO_WARNINGS
```

Definition at line 1 of file util.c.

4.195.2 Function Documentation

4.195.2.1 free_cone()

```
void ABIP() free_cone (
            ABIPCone * k )
```

Definition at line 148 of file util.c.

4.195.2.2 free_data()

```
void ABIP() free_data (
            ABIPData * d )
```

Definition at line 160 of file util.c.

4.195.2.3 free_info()

```
void ABIP() free_info (
            ABIPInfo * info )
```

Definition at line 142 of file util.c.

4.195.2.4 free_sol()

```
void ABIP() free_sol (
            ABIPSolution * sol )
```

Definition at line 182 of file util.c.

4.195.2.5 print_array()

```
void ABIP() print_array (
            const abip_float * arr,
            abip_int n,
            const char * name )
```

Definition at line 119 of file util.c.

4.195.2.6 print_data()

```
void ABIP() print_data (
            const ABIPData * d )
```

Definition at line 100 of file util.c.

4.195.2.7 print_work()

```
void ABIP() print_work (
            const ABIPWork * w )
```

Definition at line 80 of file util.c.

4.195.2.8 set_default_settings()

```
void ABIP() set_default_settings (
            ABIPData * d )
```

Default parameter settings.

Definition at line 203 of file util.c.

4.195.2.9 str_toc()

```
abip_float ABIP() str_toc (
            char * str,
            ABIP(timer) * t )
```

Definition at line 74 of file util.c.

4.195.2.10 tic()

```
void ABIP() tic (
            ABIP(timer) * t )
```

Definition at line 48 of file util.c.

4.195.2.11 toc()

```
abip_float ABIP() toc (
            ABIP(timer) * t )
```

Definition at line 68 of file util.c.

4.195.2.12 tocq()

```
abip_float ABIP() tocq (
            ABIP(timer) * t )
```

Definition at line 50 of file util.c.

4.196 util.c

Go to the documentation of this file.

```
#define _CRT_SECURE_NO_WARNINGS
#include "util.h"

#include "glbopts.h"
#include "linsys.h"

#if (defined NOTIMER)

void ABIP(tic)(ABIP(timer) * t) {}

abip_float ABIP(tocq)(ABIP(timer) * t) { return NAN; }

#elif (defined _WIN32 || _WIN64 || defined _WINDLL)

void ABIP(tic)(ABIP(timer) * t) {
  QueryPerformanceFrequency(&t->freq);
  QueryPerformanceCounter(&t->tic);
}

abip_float ABIP(tocq)(ABIP(timer) * t) {
  QueryPerformanceCounter(&t->toc);
  return (1e3 * (t->toc.QuadPart - t->tic.QuadPart) /
          (abip_float)t->freq.QuadPart);
}

#elif (defined __APPLE__)

void ABIP(tic)(ABIP(timer) * t) {
  /* read current clock cycles */
  t->tic = mach_absolute_time();
}

abip_float ABIP(tocq)(ABIP(timer) * t) {
  uint64_t duration;

  t->toc = mach_absolute_time();
  duration = t->toc - t->tic;

  mach_timebase_info(&(t->tinfo));
  duration *= t->tinfo.numer;
  duration /= t->tinfo.denom;

  return (abip_float)duration / 1e6;
}

#else

void ABIP(tic)(ABIP(timer) * t) { clock_gettime(CLOCK_MONOTONIC, &t->tic); }

abip_float ABIP(tocq)(ABIP(timer) * t) {
  struct timespec temp;

  clock_gettime(CLOCK_MONOTONIC, &t->toc);

  if ((t->toc.tv_nsec - t->tic.tv_nsec) < 0) {
    temp.tv_sec = t->toc.tv_sec - t->tic.tv_sec - 1;
    temp.tv_nsec = 1e9 + t->toc.tv_nsec - t->tic.tv_nsec;
  } else {
    temp.tv_sec = t->toc.tv_sec - t->tic.tv_sec;
    temp.tv_nsec = t->toc.tv_nsec - t->tic.tv_nsec;
  }

  return (abip_float)temp.tv_sec * 1e3 + (abip_float)temp.tv_nsec / 1e6;
}

#endif

abip_float ABIP(toc)(ABIP(timer) * t) {
  abip_float time = ABIP(tocq)(t);
  abip_printf("time: %8.4f milli-seconds.\n", time);
  return time;
}

abip_float ABIP(str_toc)(char *str, ABIP(timer) * t) {
  abip_float time = ABIP(tocq)(t);
  abip_printf("%s - time: %8.4f milli-seconds.\n", str, time);
  return time;
}

void ABIP(print_work)(const ABIPWork *w) {
  abip_int i;
  abip_int l = w->n + w->m;
```

```
  abip_printf("\n u_t is \n");
  for (i = 0; i < l; i++) {
    abip_printf("%f\n", w->u_t[i]);
  }

  abip_printf("\n u is \n");
  for (i = 0; i < l; i++) {
    abip_printf("%f\n", w->u[i]);
  }

  abip_printf("\n v is \n");
  for (i = 0; i < l; i++) {
    abip_printf("%f\n", w->v[i]);
  }
}

void ABIP(print_data)(const ABIPData *d) {
  abip_printf("m = %i\n", (int)d->m);
  abip_printf("n = %i\n", (int)d->n);

  abip_printf("max_ipm_iters = %i\n", (int)d->stgs->max_ipm_iters);
  abip_printf("max_admm_iters = %i\n", (int)d->stgs->max_admm_iters);

  abip_printf("verbose = %i\n", (int)d->stgs->verbose);
  abip_printf("normalize = %i\n", (int)d->stgs->normalize);

  abip_printf("eps_p = %4f\n", d->stgs->eps_p);
  abip_printf("eps_d = %4f\n", d->stgs->eps_d);
  abip_printf("eps_g = %4f\n", d->stgs->eps_g);
  abip_printf("eps_inf = %4f\n", d->stgs->eps_inf);
  abip_printf("eps_unb = %4f\n", d->stgs->eps_unb);
  abip_printf("alpha = %4f\n", d->stgs->alpha);
  abip_printf("rho_y = %4f\n", d->stgs->rho_y);
}

void ABIP(print_array)(const abip_float *arr, abip_int n, const char *name) {
  abip_int i;
  abip_int j;
  abip_int k = 0;

  abip_int num_on_one_line = 10;

  abip_printf("\n");
  for (i = 0; i < n / num_on_one_line; ++i) {
    for (j = 0; j < num_on_one_line; ++j) {
      abip_printf("%s[%li] = %4f, ", name, (long)k, arr[k]);
      k++;
    }
    abip_printf("\n");
  }

  for (j = k; j < n; ++j) {
    abip_printf("%s[%li] = %4f, ", name, (long)j, arr[j]);
  }

  abip_printf("\n");
}

void ABIP(free_info)(ABIPInfo *info) {
  if (info) {
    abip_free(info);
  }
}

void ABIP(free_cone)(ABIPCone *k) {
  if (k) {
    if (k->q) {
      abip_free(k->q);
    }
    if (k->rq) {
      abip_free(k->rq);
    }
    abip_free(k);
  }
}

void ABIP(free_data)(ABIPData *d) {
  if (d) {
    if (d->b) {
      abip_free(d->b);
    }

    if (d->c) {
      abip_free(d->c);
    }

```

```
    if (d->stgs) {
      abip_free(d->stgs);
    }

    if (d->A) {
      ABIP(free_A_matrix)(d->A);
    }

    abip_free(d);
  }
}

void ABIP(free_sol)(ABIPSolution *sol) {
  if (sol) {
    if (sol->x) {
      abip_free(sol->x);
    }

    if (sol->y) {
      abip_free(sol->y);
    }

    if (sol->s) {
      abip_free(sol->s);
    }

    abip_free(sol);
  }
}

void ABIP(set_default_settings)(ABIPData *d) {
  abip_int n = d->n;
  abip_int m = d->m;
  abip_int nz = d->A == ABIP_NULL ? 0 : d->A->p[n];
  abip_float sparsity = (abip_float)nz / (m * n);

  d->stgs->normalize = 1;
  d->stgs->scale_E = 1;
  d->stgs->scale_bc = 1;
  d->stgs->max_ipm_iters = MAX_IPM_ITERS;
  d->stgs->max_admm_iters = MAX_ADMM_ITERS;
  d->stgs->eps = EPS;
  d->stgs->eps_p = EPS;
  d->stgs->eps_d = EPS;
  d->stgs->eps_g = EPS;
  d->stgs->eps_inf = EPS;
  d->stgs->eps_unb = EPS;
  d->stgs->alpha = ALPHA;
  d->stgs->cg_rate = CG_RATE;

  d->stgs->use_indirect = 0;

  d->stgs->scale = SCALE;
  d->stgs->rho_y = 1e-6;
  d->stgs->rho_x = 1;
  d->stgs->rho_tau = 1;
  d->stgs->verbose = VERBOSE;

  d->stgs->err_dif =
      0;  // tol between max(dres,pres,dgap) of two consecutive inters

  d->stgs->inner_check_period = 500;
  d->stgs->outer_check_period = 1;

  // 0:mkl_dss, 1:qdldl, 2:sparse cholesky, 3:pcg,  4:pardiso, 5:dense cholesky
  if (m * n > 1e12) {
    d->stgs->linsys_solver = 3;
  } else if (sparsity > 0.4) {
    d->stgs->linsys_solver = 5;
  } else {
    d->stgs->linsys_solver = 1;
  }

  d->stgs->prob_type = 3;          // 0:general_qp, 1:lasso, 2:svm, 3:QCP
  d->stgs->time_limit = INFINITY;  // in s

  d->stgs->psi = 1;  // for qp&socp
  // d->stgs->psi = 1.5; //for ml

  d->stgs->origin_scaling = 1;
  d->stgs->ruiz_scaling = 1;
  d->stgs->pc_scaling = 0;
}
```

Index