BlockIP

Generated by Doxygen 1.8.2

Thu May 29 2014 16:22:20

Contents

Chapter 1

Main Page

BlockIP implements a specialized primal-dual long-step path-following interior point algorithm for linear, separable convex quadratic, or separable convex nonlinear problems with primal block structure.

Defining f(x) as either

```
f(x)= c_1 x_1 + ...+ c_k x_k + c_{k+1} x_{k+1} +
      1/2 (x_1...x_k x_{k+1}) Q (x_1...x_k x_{k+1})
```

or

```
f(x)= f(x_1, ..., x_k,x_{k+1})
```

the problem is

```
min  f(x)
subj. to
   N_i x_i = b_i  i=1,...,k                        Block equations
   (L_1*x_1 + ...+ L_k*x_k) + x_{k+1} = b_{k+1}    Linking constraints
   0 <= x_i <= u_i   i=1,...,k+1                   Bounds
```

where x_i i=1..k are the variables for each block and x_{k+1} are the slacks of the linking constraints

The normal equations for the dual direction is computed in two parts. The part associated to blocks is solved through k Cholesky factorizations. The part associated to linking constraints is computed with a PCG using a power series preconditioner. Cholesky factorizations are currently performed with Ng-Peyton's sprsblkllt; code is ready for others solvers.

A detailed description of BlockIP can be found in

- J. Castro, Interior-point solver for convex separable block-angular problems, Research Report DR 2014/03, Dept. of Statistics and Operations Research, Universitat Politecnica de Catalunya, 2014 (submitted).

Previous papers related to some features of the package are:

- J. Castro, A specialized interior-point algorithm for multicommodity network flows, SIAM Journal on Optimization, 10 (2000), 852-877.
- J. Castro, An interior-point approach for primal block-angular problems, Computational Optimization and Applications 36 (2007) 195-219.
- J. Castro, J. Cuesta, Quadratic regularizations in an interior-point method for primal block-angular problems, Mathematical Programming, 130 (2011) 415-445.

Some features of the implementation are:

- it can deal with any problem with primal block-angular structure
- it deals with quadratic costs of variables
- it considers linear and quadratic costs for slacks of linking constraints
- it considers lower and upper bounds for linking constraints
- rhs terms b_i and b_{k+1} can not be infinity.
- it can deal with convex separable nonlinear problems (i.e., with diagonal Hessian)
- both Newton and second-order predictor-corrector directions are available

The user can also solve the dual direction systems by the Cholesky factorization of the full system A∗Theta∗A' (A includes block and linking constraints).

Instances are created by the BlockIP() constructors. There are two variants of the constructor: one for problems in standard form (0 <= variables <= ub and constraints = rhs, and linking constraints contain slacks), another for general problems (lb <= variables <= ub and lhs <= constraints <= rhs).

See the constructors for information about the input parameters needed to define an instance.

Original idea and implementation: Jordi Castro

Other programmers: Xavi Jimenez

Jordi Castro, May 2014

Dept. of Statistics and Operations Research

Universitat Politecnica de Catalunya

Barcelona, Catalonia

Chapter 2

Hierarchical Index

2.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

Chapter 3

Class Index

3.1 Class List

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

Chapter 4

File Index

4.1 File List

Here is a list of all files with brief descriptions:

Chapter 5

Class Documentation

5.1 BlockIP::BackupLnk Struct Reference

Public Attributes

- int lnk
- double rhs
- double ** a

5.1.1 Member Data Documentation

5.1.1.1 double** BlockIP::BackupLnk::a

5.1.1.2 int BlockIP::BackupLnk::lnk

5.1.1.3 double BlockIP::BackupLnk::rhs

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

- /home/jcastro2/intpoint/BlockIP/BlockIP/src_blockip-v2/BlockIP.h

5.2 BlockIP Class Reference

Main class for loading and solving problems.

```
#include <BlockIP.h>
```

Classes

- struct BackupLnk

Public Types

- enum TYPE_PROBLEM { LINEAR, QUADRATIC, NONLINEAR }
- enum TYPE_COMP_DY { CHOL_PWRS_PCG, FULL_CHOL, HYBRID_PCG, CHOL_THETA0_PCG }
- enum TYPE_DIRECTION { NEWTON, SECOND_ORDER, AUTOMATIC }
- enum TYPE_START_POINT { SIMPLE, QUAD_EQCONS_PROB }
- enum TYPE_REG { NO_REG, QUAD_REG, PROX_REG }
- enum OUTPUT { NONE, SCREEN, FILE, BOTH }

Public Member Functions

- BlockIP ()

  Constructor.
- BlockIP (TYPE_PROBLEM type_objective, double *&cost, double *&qcost, void(*fobj)(int n, double x[], double &fx, double Gx[], double Hx[], void *params), void *params, double *&ub, double *&rhs, int numBlocks, bool sameN, MatrixBlockIP N[], bool sameL, MatrixBlockIP L[])

  Create an instance with problem in standard form: 0 <= variables <= ub and constraints = rhs.
- BlockIP (TYPE_PROBLEM type_objective, double *&cost, double *&qcost, void(*fobj)(int n, double x[], double &fx, double Gx[], double Hx[], void *params), void *params, double *&lb, double *&ub, double *&lhs, double *&rhs, int numBlocks, bool sameN, MatrixBlockIP N[], bool sameL, MatrixBlockIP L[])

  Create an instance with general problem: lb <= variables <= ub and lhs <= constraints <= rhs.
- ~BlockIP ()

  Destructor.
- void initialize ()

  Initializes class attributes.
- void free_memory ()

  Free all the memory.
- void create_problem (TYPE_PROBLEM type_objective, double *&cost, double *&qcost, void(*fobj)(int n, double x[], double &fx, double Gx[], double Hx[], void *params), void *params, double *&ub, double *&rhs, int numBlocks, bool sameN, MatrixBlockIP N[], bool sameL, MatrixBlockIP L[])

  Create a problem in standard form: 0 <= variables <= ub and constraints = rhs.
- void create_problem (TYPE_PROBLEM type_objective, double *&cost, double *&qcost, void(*fobj)(int n, double x[], double &fx, double Gx[], double Hx[], void *params), void *params, double *&lb, double *&ub, double *&lhs, double *&rhs, int numBlocks, bool sameN, MatrixBlockIP N[], bool sameL, MatrixBlockIP L[])

  Create a general problem: lb <= variables <= ub and lhs <= constraints <= rhs.
- void convert_to_standard ()

  Convert any problem to standard form, restrictions= rhs, variables >= 0, <= ub.
- void check_input_problem_and_compute_dimensions ()

  Check the input problem and compute dimensions.
- void check_sameN ()

  Check if the problems satisfies the conditions when using sameN.
- int minimize ()
- void set_defaults_minimize ()
- int get_it ()

  Return number of IP iterations of minimization.
- int get_cgit ()

  Return number of overall PCG iterations.
- double get_fobj ()

  Return objective function.
- double get_dobj ()

  Return dual objective function.
- void set_inf (double inf=INF)

  Set threshold value to be used as infinity (x>inf -> x is inf)
- double get_inf ()

  Get infinity value to be considered.
- void set_m_pw_prec (int m_pw_prec=M_PW_PREC)

  *Set number of terms used as preconditioner of the power series expansion of (D-C'*B^{-1}*B)^{-1}.*
- int get_m_pw_prec ()

  *Get number of terms used as preconditioner of the power series expansion of (D-C'*B^{-1}*B)^{-1}.*
- void set_sigma (double sigma=SIGMA)

  Set reduction of the centrality parameter at each IP iteration.

- double get_sigma ()

  Get reduction of the centrality parameter at each IP iteration.
- void set_rho (double rho=RHO)

  Set reduction of the step-length for the primal and dual variables at each IP iteration.
- double get_rho ()

  Get reduction of the step-length for the primal and dual variables at each IP iteration.
- void set_optim_gap (double optim_gap=OPTIM_GAP)

  Set optimality gap tolerance.
- double get_optim_gap ()

  Get optimality gap tolerance.
- void set_optim_pfeas (double optim_pfeas=OPTIM_PFEAS)

  Set primal feasibility tolerance.
- double get_optim_pfeas ()

  Get primal feasibility tolerance.
- void set_optim_dfeas (double optim_dfeas=OPTIM_DFEAS)

  Set dual feasibility tolerance.
- double get_optim_dfeas ()

  Get dual feasibility tolerance.
- void set_output_freq (int output_freq=OUTPUT_FREQ)

  Set output information lines will be printed each output_freq IP iterations.
- int get_output_freq ()

  Get output information lines will be printed each output_freq IP iterations.
- void set_output (OUTPUT output=SCREEN)

  Set type of output.
- OUTPUT get_output ()

  Get type of output.
- void set_maxiter (int maxiter=MAXITER)

  Set maximum number of IP iterations.
- int get_maxiter ()

  Get maximum number of IP iterations.
- void set_init_pcgtol (double init_pcgtol)

  Set initial tolerance for the conjugate gradient (by default PCG_TOL_LIN if linear problem, PCG_TOL_QUAD if quadratic or -convex- nonlinear)
- double get_init_pcgtol ()

  Get initial tolerance for the conjugate gradient (by default PCG_TOL_LIN if linear problem, PCG_TOL_QUAD if quadratic or -convex- nonlinear)
- void set_min_pcgtol (double min_pcgtol=MIN_PCGTOL)

  Set minimum tolerance for the conjugate gradient.
- double get_min_pcgtol ()

  Get minimum tolerance for the conjugate gradient.
- void set_maxit_pcg (double maxit_pcg=0)

  Set maximum number of pcg iterations (if 0, then the value will be computed by the code)
- double get_maxit_pcg ()

  Get maximum number of pcg iterations (if 0, then the value will be computed by the code)
- void set_red_pcgtol (double red_pcgtol=RED_PCGTOL)

  Set reduction of pcg_tol at each IP iteration.
- double get_red_pcgtol ()

  Get reduction of pcg_tol at each IP iteration.
- void set_show_specrad (bool show_specrad=false)

  Set show_specrad at each IP iteration.
- bool get_show_specrad ()

Get show_specrad.

- void set_type_comp_dy (TYPE_COMP_DY type_comp_dy=TYPE_COMP_DY_DEFAULT)

  Set how dy direction is computed.
- TYPE_COMP_DY get_type_comp_dy ()

  Get how dy direction is computed.
- void set_type_direction (TYPE_DIRECTION type_direction=TYPE_DIRECTION_DEFAULT)

  Set which direction is computed.
- TYPE_DIRECTION get_type_direction ()

  Get which direction is computed.
- void set_type_start_point (TYPE_START_POINT type_start_point=TYPE_START_POINT_DEFAULT)

  Set how starting point is computed.
- TYPE_START_POINT get_type_start_point ()

  Get how starting point is computed.
- void set_type_reg (TYPE_REG type_reg=TYPE_REG_DEFAULT)

  Set type of regularization.
- TYPE_REG get_type_reg ()

  Get type of regularization.
- void set_factor_reg (double factor_reg=FACTOR_REG_DEFAULT)

  Set factor of regularization.
- double get_factor_reg ()

  Get factor of regularization.
- void set_deactivateLnk (bool deactivateLnk=DEACTIVATELNK)

  Set flag on deactivation of linking.
- bool get_deactivateLnk ()

  Get flag on deactivation of linking.
- int get_k_blocks ()

  Get the number of blocks.
- int get_km ()

  Get the number of constraints without linking constraints.
- int get_kn ()

  Get the number of variables without linking constraints.
- int get_l_link ()

  Get the number of linking constraints.
- int get_n_vars ()

  Get the number of variables of the problem to be optimized.
- int get_m_cons ()

  Get the number of constraints of the problem to be optimized.
- int get_num_vars ()

  Get the number of variables of the original problem.
- int get_num_cons ()

  Get the number of constraints of the original problem.
- void set_whoperm (WHO_PERMUTES whoperm_=CHOLESKY)

  Set whopermutes constraints-related information: the Cholesky factorization of the user of it.
- WHO_PERMUTES get_whoperm ()

  Get whopermutes constraints-related information: the Cholesky factorization of the user of it.
- const double * get_x ()

  Get primal variables of problem optimized.
- const double * get_y ()

  Get dual variables of problem optimized (of equality constraints)
- const double * get_z ()

  Get dual variable of problem optimized (of x>=0) i=1..n_vars.

- const double * get_w ()

  Get dual variable of problem optimized (of x<=u) i=1..n_vars.

- double get_x (int i)

  Get value of primal variable x(i) of problem optimized i=1..n_vars.

- double get_y (int i)

  Get value of dual variable y(i) of problem optimized (of equality constraints) i=1..m_cons.

- double get_z (int i)

  Get value of dual variable z(i) (of x>=0) of problem optimized i=1..n_vars.

- double get_w (int i)

  Get value of dual variable w(i) (of x<=u) of problem optimized i=1..n_vars.

- int min_PCG_Hv (int nn, double *v, double *Hv)
- int min_PCG_Hz_eq_r (int nn, double *zz, double *rr)
- int min_PCG_Theta0z_eq_r (int nn, double *zz, double *rr)
- void set_constant_fobj (double constant)

  Set the constant to add to the objective function.

- double get_constant_fobj ()

  Get the constant to add to the objective function.

- void set_names (string *blockNames, string *varNames, string *consNames, bool copy_vectors=true)

  Set the names of the problem TODO change copy_vectors.

- void get_names (const string *&blockNames, const string *&varNames, const string *&consNames)

  Get the names of the problem.

- void convert_to_std_and_write_mps (const char *filename)

  Convert the problem to standard form (if it is not already converted) and write a mps file.

- void write_mps (const char *filename)

  Write a mps file.

- void write_mps (const char *filename, TYPE_PROBLEM type_objective, double cost[], double qcost[], double lb[], double ub[], double lhs[], double rhs[], int numBlocks, bool sameN, MatrixBlockIP N[], bool sameL, MatrixBlockIP L[], string *blockNames=NULL, string *varNames=NULL, string *consNames=NULL, double constantFObj=0)

  Write a mps file.

- void read_mps (const char *filename)

  Create a problem from a mps file.

- void write_BlockIP_format (const char *filename)

  Write the problem in BlockIP format file.

- void read_BlockIP_format (const char *filename)

  Create a problem from a BlockIP format file.

- void write_problem (const char *filename)

  Write all data related to the problem into a file.

- StdForm * get_std_form (bool keepStdFormAfterDelete=false)

  Get the StdForm related to the problem.

- void create_names ()

  Create names for blocks, variables and constraints if does not exist.

- string * get_full_var_names ()
- string * get_full_cons_names ()

Public Attributes

- int k_blocks

  Number of diagonal blocks.
- int km

  Number of constraints without linking constraints.
- int kn

  Number of variables without slacks of linking.
- int m_cons

  Number of constraints including linking constraints.
- int n_vars

  Number of variables including slacks.
- int l_link

  Number of linking constraints.
- int * ini_n

  Begin to blocks 1..k and linking for variables.
- int * ini_m

  Begin to blocks 1..k and linking for constraints.
- double * cost

  Linear cost of variables including slacks.
- double * qcost

  Quadratic cost of variables including slacks.
- double * lb

  Lower bounds, including slack bounds.
- double * ub

  Upper bounds, including slack bounds.
- double * lhs

  Lower constraint limits, including linking constraints limits.
- double * rhs

  Upper constraint limits, including linking constraints limits.
- bool sameN

  Define if the same matrix is used for each N block.
- bool sameL

  Define if the same matrix is used for each L block.
- MatrixBlockIP * N

  Diagonal blocks.
- MatrixBlockIP * L

  Linking constraints blocks.
- MatrixBlockIP * A

  Full matrix (likely made from N and L)
- MatrixBlockIP * D

  *D = Theta_(k+1) + sum{i..k} L_i*Theta_i*L_i'.*
- MatrixBlockIP * Theta0
- void(* fobj)(int n, double x[], double &fx, double Gx[], double Hx[], void *params)

  User function to calculate the objective function in a point.
- void * params

  User parameters to perform objective function calculations.
- double fx
- double * Gx
- double * Hx

  Objective function value, Gradient and Hessian in the actual point.

- double dobj

  dual objective
- bool inStdForm

  To control if the problem is in standard form.
- string * blockNames

  Block names of N blocks.
- string * varNames

  Variable names including slacks.
- string * consNames

  Constraint names including linking constraints.
- TYPE_PROBLEM type_objective

  Type of objective function, linear, quadratic or non-linear.
- StdForm * stdForm

  Contains all the information to perform conversions between the original and standard problem.
- double constantFObj

  Constant to add in the objective function.
- int * origNVars

  Number of original variables for each block.
- int * origNCons

  Number of original constraints for each block.

Private Member Functions

- void min_initializations ()

  Initialize parameters, tolerances, etc for BlockIP minimization algorithm.
- void min_preprocess ()
- void min_normb_normc ()
- void min_Ax (double *Ax, const double *x)
- void min_Aty (double *Aty, const double *y)
- void min_compute_s ()
- void min_compute_mu ()
- void min_compute_sigma_psi ()
- void min_update_inactivelnk ()
- void min_KKT_residuals ()
- void min_starting_point ()
- void min_objective_functions_nonlin ()
- void min_objective_functions_linquad ()
- void eval_fobj (double x[], double &fx, double Gx[], double Hx[])
- void min_write_problem_information ()
- void min_write_current_point_info ()
- void min_steplengths (double &talpha_p, double &talpha_d, double tdx[], double tdz[], double tdw[])
- bool min_optimal ()
- void min_compute_Theta ()
- void min_compute_Theta0 ()
- void min_compute_direction ()
- void min_Newton_direction ()
- void min_second_order_predictor_corrector_direction ()
- void min_update_newpoint ()
- void min_test_newpoint ()
- void min_backup_inactive (int lnk)
- void min_restore_inactive (int lnk)
- void min_compute_dy_cholpcg (double *dy, double *rhsdy, bool only_solve=false)

- void min_compute_dy_fullchol (double *dy, double *rhsdy, bool only_solve=false)
- void min_free_memory ()
- void min_check_Newton_direction_is_correct ()
- void min_check_predictor_direction_is_correct ()
- void min_check_predictor_corrector_direction_is_correct ()
- void min_solve_NThetaNt (double *v)
- void min_Ctv (double *v, double *Ctv)
- void min_Cv (double *v, double *Cv)
- void min_debug_write_variables ()
- void min_debug_variables_of_inactiveInk ()
- void min_debug_variables_without_ub ()
- void min_update_qreg ()

Private Attributes

- int input_problem
- bool initialized

  To control if the attributes have been initialized.
- bool keepStdFormAfterDelete

  To keep stdForm when BlockIP will delete.
- bool deleteMatrices

  To delete matrices when created by BlockIP (read_mps)
- WHO_PERMUTES whoperm
- double inf

  Infinity value.
- double optim_gap

  Optimality gap tolerance.
- double optim_pfeas

  Primal feasibility tolerance.
- double optim_dfeas

  Dual feasibility tolerance.
- int maxiter

  Maximum number of IP iterations.
- int output_freq

  Output information lines will be printed each output_freq IP iterations.
- OUTPUT output

  Type of output.
- double epsmach

  Stores computed machine epsilon.
- double init_pcgtol

  Initial tolerance for the conjugate gradient (by default PCG_TOL_LIN if linear problem, PCG_TOL_QUAD if quadratic or -convex- nonlinear)
- double pcgtol

  Tolerance for conjugate gradient at current iteration.
- double min_pcgtol

  Minimum tolerance for the conjugate gradient.
- double maxit_pcg

  Maximum number of pcg iterations (if 0, then the value will be computed by the code)
- double red_pcgtol

  Reduction of pcg_tol at each IP iteration.
- int m_pw_prec

Number of terms of the power series expansion of (D-C'∗B^{-1}∗B)^{-1} used as preconditioner.

- int totcgit

  Overall number of CG iterations.

- int cgit

  number of CG iterations of this IPM iteration

- int it

  IPM iterations.

- double est_specrad

  Estimation of spectral radius of D^{-1}C'∗B^{-1}∗B (computed through Ritz values)

- bool show_specrad
- bool comp_specrad

  Whether the estimation of spectral radius has to be shown in the output.

- TYPE_COMP_DY type_comp_dy

  Whether the estimation of spectral radius has to be computed.

- TYPE_DIRECTION type_direction

  Type of direction (Newton, second order...) given by user.

- TYPE_DIRECTION this_type_direction

  Type of direction (Newton or second order) of this particular iteration.

- TYPE_START_POINT type_start_point
- TYPE_REG type_reg

  How starting point will be computed.

- double factor_reg

  Type of regularization performed.

- double ∗ x

  initial value for regularization

- double ∗ y
- double ∗ z
- double ∗ w
- double ∗ s

  primal and dual optimization variables

- double ∗ rb
- double ∗ rc
- double ∗ rxz
- double ∗ rsw

  residuals of KKT conditions

- double ∗ dxdz
- double ∗ dsdw

  for rhs of corrector system, from dx,dw,ds and dw of predictor direction

- double normb
- double normc
- double normrb
- double normrc

  norms of rhs, costs, rb, and rc

- double alpha_p
- double alpha_d

  step lengths

- double rho

  Reduction of the step-length for the primal and dual variables at each IP iteration.

- double ∗ dx
- double ∗ dy
- double ∗ dz
- double ∗ dw

Primal and dual optimization directions.

- double gap

  Duality gap.
- double mu
- double mu0

  Centrality parameter; mu0 is value of mu at starting point.
- double sigma

  Reduction of the centrality parameter at each IP iteration.
- double psi

  Reduction of dx∗dz vector in the rhs of corrector system (in predictor-corrector)
- double ∗ Theta

  Theta diagonal matrix.
- double ∗ Ax
- double ∗ Aty
- double ∗ r
- double ∗ rhsdy
- double ∗ rhspcg
- double ∗ Cvaux
- double ∗ Cvaux2

  Auxiliary vectors for interior-point algorithm.
- double ∗ r_cg
- double ∗ z_cg
- double ∗ p_cg

  Auxiliary vectors for PCG.
- double qreg

  Regularization term.
- double ∗ valpha
- double ∗ vbeta
- bool deactivateLnk

  Vectors to store information from PCG to later compute Ritz values.
- int numActiveLnk

  Number of active lnk.
- int numInactiveLnk

  Number of inactive lnk.
- int numInactiveLnkWithUb

  Number of inactive lnk with upper bound.
- bool ∗ isActiveLnk

  linking i is active if isActiveLnk[i] is true
- int ∗ listActiveLnk

  list of active linking, j=listActiveLnk[i], i=1..numActiveLnk, j in {1,..,l_link})
- int ∗ listInactiveLnk

  list of inactive linking, j=listInactiveLnk[i], i=1..numInactiveLnk, j in {1,..,l_link})
- int numFreeVars

  Number of variables marked as free.
- int numUnfreeVars

  Number of variables not marked as free.
- bool ∗ isFreeVar

  Variable i is marked as free if isFreeVar[i] is true.
- int ∗ listFreeVars

  List of variables marked as free, j=listFreeVars[i], i=1..numFreeVars, j in {1,..,n_vars})
- int ∗ listUnfreeVars

List of variables not marked as free, j=listUnfreeVars[i], i=1..numUnfreeVars, j in {1,..,n_vars})

- int numBackupLnk
- BackupLnk * backupLnk
- int numWithUb

  Number of variables with upper bound (not infinity)
- int numWithoutUb
- int * listWithoutUb
- int * listWithUb

Static Private Attributes

- static constexpr double INF = 1.0e128
- static const int M_PW_PREC = 1
- static constexpr double SIGMA = 0.1
- static constexpr double RHO = 0.995
- static constexpr double OPTIM_GAP = 1.0e-6
- static constexpr double OPTIM_GAP_SAFEGUARD = 1.0e-5
- static constexpr double OPTIM_PFEAS = 1.0e-6
- static constexpr double OPTIM_DFEAS = 1.0e-6
- static const int OUTPUT_FREQ = 1
- static const int MAXITER = 200
- static constexpr double PCG_TOL_LIN = 1.0e-2
- static constexpr double PCG_TOL_QUAD = 1.0e-3
- static constexpr double MIN_PCGTOL = 1.0e-8
- static constexpr double RED_PCGTOL = 0.95
- static const TYPE_COMP_DY TYPE_COMP_DY_DEFAULT = CHOL_PWRS_PCG
- static const TYPE_DIRECTION TYPE_DIRECTION_DEFAULT = AUTOMATIC
- static const TYPE_START_POINT TYPE_START_POINT_DEFAULT = SIMPLE
- static const TYPE_REG TYPE_REG_DEFAULT = NO_REG
- static constexpr double FACTOR_REG_DEFAULT = 1.0e-6
- static const bool DEACTIVATELNK = true

5.2.1 Detailed Description

Main class for loading and solving problems.

5.2.2 Member Enumeration Documentation

5.2.2.1 enum BlockIP::OUTPUT

Enumerator:

NONE

SCREEN

FILE

BOTH

5.2.2.2 enum BlockIP::TYPE_COMP_DY

Enumerator:

CHOL_PWRS_PCG

FULL_CHOL

HYBRID_PCG

CHOL_THETA0_PCG

5.2.2.3 enum BlockIP::TYPE_DIRECTION

Enumerator:

NEWTON

SECOND_ORDER

AUTOMATIC

5.2.2.4 enum BlockIP::TYPE_PROBLEM

Enumerator:

LINEAR

QUADRATIC

NONLINEAR

5.2.2.5 enum BlockIP::TYPE_REG

Enumerator:

NO_REG

QUAD_REG

PROX_REG

5.2.2.6 enum BlockIP::TYPE_START_POINT

Enumerator:

SIMPLE

QUAD_EQCONS_PROB

5.2.3 Constructor & Destructor Documentation

5.2.3.1 BlockIP::BlockIP ( )

Constructor.

5.2.3.2 BlockIP::BlockIP (TYPE_PROBLEM *type_objective,* double *&* *cost,* double *&* *qcost,* void(*)(int n, double x[], double &fx, double Gx[], double Hx[], void *params) *fobj,* void * *params,* double *&* *ub,* double *&* *rhs,* int *numBlocks,* bool *sameN,* MatrixBlockIP *N[],* bool *sameL,* MatrixBlockIP *L[]*)

Create an instance with problem in standard form: 0 <= variables <= ub and constraints = rhs.

Parameters

type_objective	Type of objective function, linear, quadratic or non-linear
cost	Linear cost of variables including slacks
qcost	Quadratic cost of variables including slacks
fobj	User function to calculate the objective function in a point. If it is is not NULL, cost and qcost must be NULL
params	User parameters to perform objective function calculations. Only can be used when fobj is not NULL
ub	Upper bounds, including slack bounds
rhs	Upper constraint limits, including linking constraints limits
numBlocks	Number of diagonal blocks
sameN	Define if the same matrix is used for each N block. If true array N must have dimension 1
N	Diagonal blocks
sameL	Define if the same matrix is used for each L block If true array L must have dimension 1
L	Linking constraints blocks

5.2.3.3 BlockIP::BlockIP (TYPE_PROBLEM *type_objective,* double ∗& *cost,* double ∗& *qcost,* void(∗)(int n, double x[], double &fx, double Gx[], double Hx[], void ∗params) *fobj,* void ∗ *params,* double ∗& *lb,* double ∗& *ub,* double ∗& *lhs,* double ∗& *rhs,* int *numBlocks,* bool *sameN,* MatrixBlockIP *N[],* bool *sameL,* MatrixBlockIP *L[]*)

Create an instance with general problem: lb <= variables <= ub and lhs <= constraints <= rhs.

Parameters

type_objective	Type of objective function, linear, quadratic or non-linear
cost	Linear cost of variables including slacks
qcost	Quadratic cost of variables including slacks
fobj	User function to calculate the objective function in a point. If it is is not NULL, cost and qcost must be NULL
params	User parameters to perform objective function calculations. Only can be used when fobj is not NULL
lb	Lower bounds, including slack bounds
ub	Upper bounds, including slack bounds
lhs	Lower constraint limits, including linking constraints limits
rhs	Upper constraint limits, including linking constraints limits
numBlocks	Number of diagonal blocks
sameN	Define if the same matrix is used for each N block. If true array N must have dimension 1
N	Diagonal blocks
sameL	Define if the same matrix is used for each L block If true array L must have dimension 1
L	Linking constraints blocks

Note

The problem must be converted to standard form in order to be solved, it will be converted automatically when minimize function is called. After that call the problem only may be written in standard form. Before that call the problem can be written in standard form with convert_to_std_and_write_mps function or in general form with write_mps function.
If sameN is used each constraint must be the same type in each block, e.g., if constraint i is an equality in block j must be an equality in all other blocks.
If sameN is used each free variable with zero value in Hessian must be the same type in each block, e.g., if variable i is free variable with zero value in Hessian of block j, it must be a free variable with zero value in Hessians of all other blocks.

5.2.3.4 BlockIP::~BlockIP ( )

Destructor.

5.2.4 Member Function Documentation

5.2.4.1 void BlockIP::check_input_problem_and_compute_dimensions ( )

Check the input problem and compute dimensions.

5.2.4.2 void BlockIP::check_sameN ( )

Check if the problems satisfies the conditions when using sameN.

5.2.4.3 void BlockIP::convert_to_standard ( )

Convert any problem to standard form, restrictions= rhs, variables >= 0, <= ub.

5.2.4.4 void BlockIP::convert_to_std_and_write_mps (const char * *filename*)

Convert the problem to standard form (if it is not already converted) and write a mps file.

Parameters

filename	File name without extension

5.2.4.5 void BlockIP::create_names ( )

Create names for blocks, variables and constraints if does not exist.

5.2.4.6 void BlockIP::create_problem (TYPE_PROBLEM *type_objective,* double *& *cost,* double *& *qcost,* void(*)(int n, double x[], double &fx, double Gx[], double Hx[], void *params) *fobj,* void * *params,* double *& *ub,* double *& *rhs,* int *numBlocks,* bool *sameN,* MatrixBlockIP *N[],* bool *sameL,* MatrixBlockIP *L[]*)

Create a problem in standard form: 0 <= variables <= ub and constraints = rhs.

Parameters

type_objective	Type of objective function, linear, quadratic or non-linear
cost	Linear cost of variables including slacks
qcost	Quadratic cost of variables including slacks
fobj	User function to calculate the objective function in a point. If it is is not NULL, cost and qcost must be NULL
params	User parameters to perform objective function calculations. Only can be used when fobj is not NULL
ub	Upper bounds, including slack bounds
rhs	Upper constraint limits, including linking constraints limits
numBlocks	Number of diagonal blocks
sameN	Define if the same matrix is used for each N block. If true array N must have dimension 1
N	Diagonal blocks
sameL	Define if the same matrix is used for each L block If true array L must have dimension 1
L	Linking constraints blocks !

5.2.4.7 void BlockIP::create_problem (TYPE_PROBLEM *type_objective,* double *&* *cost,* double *&* *qcost,* void(*)(int n, double x[], double &fx, double Gx[], double Hx[], void *params) *fobj,* void * *params,* double *&* *lb,* double *&* *ub,* double *&* *lhs,* double *&* *rhs,* int *numBlocks,* bool *sameN,* MatrixBlockIP *N[],* bool *sameL,* MatrixBlockIP *L[]*)

Create a general problem: lb <= variables <= ub and lhs <= constraints <= rhs.

Parameters

type_objective	Type of objective function, linear, quadratic or non-linear
cost	Linear cost of variables including slacks
qcost	Quadratic cost of variables including slacks
fobj	User function to calculate the objective function in a point. If it is is not NULL, cost and qcost must be NULL
params	User parameters to perform objective function calculations. Only can be used when fobj is not NULL
lb	Lower bounds, including slack bounds
ub	Upper bounds, including slack bounds
lhs	Lower constraint limits, including linking constraints limits
rhs	Upper constraint limits, including linking constraints limits
numBlocks	Number of diagonal blocks
sameN	Define if the same matrix is used for each N block. If true array N must have dimension 1
N	Diagonal blocks
sameL	Define if the same matrix is used for each L block If true array L must have dimension 1
L	Linking constraints blocks

Note

The problem must be converted to standard form in order to be solved, it will be converted automatically when minimize function is called. After that call the problem only may be written in standard form. Before that call the problem can be written in standard form with convert_to_std_and_write_mps function or in general form with write_mps function.
If sameN is used each constraint must be the same type in each block, e.g., if constraint i is an equality in block j must be an equality in all other blocks.
If sameN is used each free variable with zero value in Hessian must be the same type in each block, e.g., if variable i is free variable with zero value in Hessian of block j, it must be a free variable with zero value in Hessians of all other blocks. !

5.2.4.8 void BlockIP::eval_fobj (double *x[],* double & *fx,* double *Gx[],* double *Hx[]*) [inline],[private]

5.2.4.9 void BlockIP::free_memory ( )

Free all the memory.

5.2.4.10 int BlockIP::get_cgit ( ) [inline]

Return number of overall PCG iterations.

5.2.4.11 double BlockIP::get_constant_fobj ( ) [inline]

Get the constant to add to the objective function.

5.2.4.12 bool BlockIP::get_deactivateLnk ( ) [inline]

Get flag on deactivation of linking.

5.2.4.13 double BlockIP::get_dobj ( ) [inline]

Return dual objective function.

5.2.4.14 double BlockIP::get_factor_reg ( ) [inline]

Get factor of regularization.

5.2.4.15 double BlockIP::get_fobj ( ) [inline]

Return objective function.

5.2.4.16 string * BlockIP::get_full_cons_names ( )

5.2.4.17 string * BlockIP::get_full_var_names ( )

5.2.4.18 double BlockIP::get_inf ( ) [inline]

Get infinity value to be considered.

5.2.4.19 double BlockIP::get_init_pcgtol ( ) [inline]

Get initial tolerance for the conjugate gradient (by default PCG_TOL_LIN if linear problem, PCG_TOL_QUAD if quadratic or -convex- nonlinear)

5.2.4.20 int BlockIP::get_it ( ) [inline]

Return number of IP iterations of minimization.

5.2.4.21 int BlockIP::get_k_blocks ( ) [inline]

Get the number of blocks.

Returns

field k_blocks

5.2.4.22 int BlockIP::get_km ( ) [inline]

Get the number of constraints without linking constraints.

Returns

field km

5.2.4.23 int BlockIP::get_kn ( ) [inline]

Get the number of variables without linking constraints.

Returns

field kn

5.2.4.24 int BlockIP::get_l_link ( ) [inline]

Get the number of linking constraints.

Returns

field l_link

5.2.4.25 int BlockIP::get_m_cons ( ) [inline]

Get the number of constraints of the problem to be optimized.

Returns

field m_cons

5.2.4.26 int BlockIP::get_m_pw_prec ( ) [inline]

Get number of terms used as preconditioner of the power series expansion of (D-C'∗B^{-1}∗B)^{-1}.

5.2.4.27 double BlockIP::get_maxit_pcg ( ) [inline]

Get maximum number of pcg iterations (if 0, then the value will be computed by the code)

5.2.4.28 int BlockIP::get_maxiter ( ) [inline]

Get maximum number of IP iterations.

5.2.4.29 double BlockIP::get_min_pcgtol ( ) [inline]

Get minimum tolerance for the conjugate gradient.

5.2.4.30 int BlockIP::get_n_vars ( ) [inline]

Get the number of variables of the problem to be optimized.

Returns

field n_vars

5.2.4.31 void BlockIP::get_names (const string ∗& *blockNames,* const string ∗& *varNames,* const string ∗& *consNames*)

Get the names of the problem.

Parameters

blockNames	Block names of N blocks, dimension k_blocks
varNames	Variable names including slacks, dimension n_vars
consNames	Constraint names including linking constraints, dimension m_cons

5.2.4.32 int BlockIP::get_num_cons ( ) [inline]

Get the number of constraints of the original problem.

Returns

Number of constraints of the original problem

5.2.4.33 int BlockIP::get_num_vars ( ) [inline]

Get the number of variables of the original problem.

Returns

Number of variables of the original problem

5.2.4.34 double BlockIP::get_optim_dfeas ( ) [inline]

Get dual feasibility tolerance.

5.2.4.35 double BlockIP::get_optim_gap ( ) [inline]

Get optimality gap tolerance.

5.2.4.36 double BlockIP::get_optim_pfeas ( ) [inline]

Get primal feasibility tolerance.

5.2.4.37 BlockIP::OUTPUT BlockIP::get_output ( ) [inline]

Get type of output.

5.2.4.38 int BlockIP::get_output_freq ( ) [inline]

Get output information lines will be printed each output_freq IP iterations.

5.2.4.39 double BlockIP::get_red_pcgtol ( ) [inline]

Get reduction of pcg_tol at each IP iteration.

5.2.4.40 double BlockIP::get_rho ( ) [inline]

Get reduction of the step-length for the primal and dual variables at each IP iteration.

5.2.4.41 bool BlockIP::get_show_specrad ( ) [inline]

Get show_specrad.

5.2.4.42 double BlockIP::get_sigma ( ) [inline]

Get reduction of the centrality parameter at each IP iteration.

5.2.4.43 StdForm * BlockIP::get_std_form (bool *keepStdFormAfterDelete* = false) [inline]

Get the StdForm related to the problem.

Parameters

keepStdForm-AfterDelete	Set if the StdForm related to the problem will be deleted or not when BlockIP will delete

Returns

The StdForm related to the problem, if the problem was created in standard form, return NULL.

5.2.4.44 BlockIP::TYPE_COMP_DY BlockIP::get_type_comp_dy ( ) [inline]

Get how dy direction is computed.

5.2.4.45 BlockIP::TYPE_DIRECTION BlockIP::get_type_direction ( ) [inline]

Get which direction is computed.

5.2.4.46 BlockIP::TYPE_REG BlockIP::get_type_reg ( ) [inline]

Get type of regularization.

5.2.4.47 BlockIP::TYPE_START_POINT BlockIP::get_type_start_point ( ) [inline]

Get how starting point is computed.

5.2.4.48 const double * BlockIP::get_w ( ) [inline]

Get dual variable of problem optimized (of x<=u) i=1..n_vars.

5.2.4.49 double BlockIP::get_w (int *i*) [inline]

Get value of dual variable w(i) (of x<=u) of problem optimized i=1..n_vars.

5.2.4.50 WHO_PERMUTES BlockIP::get_whoperm ( ) [inline]

Get whopermutes constraints-related information: the Cholesky factorization of the user of it.

5.2.4.51 const double * BlockIP::get_x ( ) [inline]

Get primal variables of problem optimized.

5.2.4.52 double BlockIP::get_x (int *i*) [inline]

Get value of primal variable x(i) of problem optimized i=1..n_vars.

5.2.4.53 const double * BlockIP::get_y ( ) [inline]

Get dual variables of problem optimized (of equality constraints)

5.2.4.54 double BlockIP::get_y (int *i*) [inline]

Get value of dual variable y(i) of problem optimized (of equality constraints) i=1..m_cons.

5.2.4.55 const double * BlockIP::get_z ( ) [inline]

Get dual variable of problem optimized (of x>=0) i=1..n_vars.

5.2.4.56 double BlockIP::get_z (int *i*) [inline]

Get value of dual variable z(i) (of x>=0) of problem optimized i=1..n_vars.

5.2.4.57 void BlockIP::initialize ( )

Initializes class attributes.

5.2.4.58 void BlockIP::min_Aty (double * *Aty,* const double * *y*) [private]

Computes and returns A'*y A is the constraints matrix, formed by k diagonal blocks N and last rows with k L matrices plus the Identity for slacks. Computes Ni'*yi + Li'*y_{k+1} for all block i=1..k; and I*y_{k+1} for last block k+1.

Parameters

y[1:km+l],:	dual variables. Values of duals y_{k+1} of inactive linking constraints must be 0.
Aty[kn+l],:	result of A'*y. Only values of duals y_{k+1} of active linking constraints are added (equivalently, the components of y_{k+1} of inactives are 0).

5.2.4.59 void BlockIP::min_Ax (double * *Ax,* const double * *x*) [private]

Computes and returns A*x A is the constraints matrix, formed by k diagonal blocks N and last rows with k L matrices plus the Identity for slacks. Computes Ni*xi for all block i=1..k; and sum{1..k}Li*xi + x_{k+1}, for the active constraints

Parameters

x[1:kn+l],:	primal variables
Ax[km+l],:	result of A*x. Only values of active linking onstraints are filled

5.2.4.60 void BlockIP::min_backup_inactive (int *lnk*) [private]

5.2.4.61 void BlockIP::min_check_Newton_direction_is_correct ( ) [private]

For debugging purposes, checks direction satisfies Newton system

| -H A' I -I | |dx| |rc| | A | |dy| = |rb| | Z X | |dz| |rxz| | -W S | |dw| |rsw|

If QUAD_REG then it considers H:= H+qreg for block variables

5.2.4.62 void BlockIP::min_check_predictor_corrector_direction_is_correct ( ) [private]

For debugging purposes, checks predictor-corrector direction satisfies system

| -H A' I -I | |dx| |rc | | A | |dy| = |rb | | Z X | |dz| |sigma∗mu-XZe-Dx_p Dz_p e| | -W S | |dw| |sigma∗mu-SWe-Ds_p Dw_p e|

If QUAD_REG then it considers H:= H+qreg for block variables

5.2.4.63 void BlockIP::min_check_predictor_direction_is_correct ( ) [private]

For debugging purposes, checks predictor direction satisfies system

| -H A' I -I | |dx| |rc | | A | |dy| = |rb | | Z X | |dz| |-XZe| | -W S | |dw| |-SWe|

If QUAD_REG then it considers H:= H+qreg for block variables

5.2.4.64 void BlockIP::min_compute_direction ( ) [private]

Decides whether to use standard Newton direction or second order direction (Mehrotra direction).

5.2.4.65 void BlockIP::min_compute_dy_cholpcg (double ∗ *dy,* double ∗ *rhsdy,* bool *only_solve =* false) [private]

Solves (A∗Theta∗A')dy = rhsdy by Cholesky for blocks and PCG for linking constraints. The structure of (A∗Theta∗-A') dy = rhsdy is

| B C | |dy1| |rhsdy1| | C' D | |dy2| = |rhsdy2|

so it can be solved as

(D-C'∗B^{-1}∗C) dy2 = rhsdy2-C'∗B^{-1}∗rhsdy1 B dy1= rhsdy1-C∗dy2

where B is made of k_blocks N_i∗Theta_i∗N_i'.

Note that system (A∗Theta∗A') is really of dimension km+numActiveLnk, then we need to *implicitly* consider only the active linking constraints when applying the PCG.

Parameters

dy[1:km+l],:	on input is the last dy solution computed, to be used as starting solution fof PCG; on output is the solution of (A∗Theta∗A)dy= rhsdy
rhsdy[1:km+l],:	right-hand-side of PCG system.
only_solve,:	if true, only numeric solve (neither symbolic nor numeric factorization are needed) since a previous call with the same A∗Theta∗A' but different rhsdy was made (for instance, in corrector step of predictor-corrector heuristic)

5.2.4.66 void BlockIP::min_compute_dy_fullchol (double ∗ *dy,* double ∗ *rhsdy,* bool *only_solve =* false) [private]

Solves (A∗Theta∗A')dy = rhsdy by Cholesky of full system. Note that system (A∗Theta∗A') is really of dimension km+numActiveLnk, but we reuse the initial symbolic factorization, where the dimension of the system was km+l_link. Instead of using a row of the identity matrix for the rows of inactive constraints to avoid a nonsingular system, we consider a 0, since Cholesky deals with 0 pivots (semidefinie matrices). This way, the values of dy are correctly computed.

Parameters

dy[1:km+l],:	on output is the solution of (A∗Theta∗A)dy= rhsdy
rhsdy[1:km+l],:	on input right-hand-side of (A∗Theta∗A)dy= rhsdy
only_solve,:	if true, only numeric solve (neither symbolic nor numeric factorization are needed) since a previous call with the same A∗Theta∗A' but different rhsdy was made (for instance, in corrector step of predictor-corrector heuristic)

5.2.4.67 void BlockIP::min_compute_mu ( ) [private]

Computes and returns the centrality at current point, computed as mu= (x'z+s'w)/((kn+number_of_active_linking) + (number of variables with upper bounds)). For efficiency of the implementation, w and s of variables without upper bound are 0 and a finite (large number)

5.2.4.68 void BlockIP::min_compute_s ( ) [private]

Computes the slacks of upper bounds of block variables and slacks of linking constraints: s= u-x. Only slacks of upper bounds of slacks of active linking constraints filled (for inactive linking, s is 0). If some u=inf, then s=u-x∼=inf which is OK for deactivating the complementarity constraints s∗w=sigma∗mu. There is no need in setting s=inf for vars in listWithoutUb since s=u-x will be very close (if not equal) to inf. This also applies to (non-splitted) free vars, whose upper bound is also inf.

5.2.4.69 void BlockIP::min_compute_sigma_psi ( ) [private]

Computes sigma (reduction of the centrality parameter at each IP iteration) and psi (reduction of dx∗dz vector in the rhs of corrector system of predictor-corrector heuristic)

5.2.4.70 void BlockIP::min_compute_Theta ( ) [private]

5.2.4.71 void BlockIP::min_compute_Theta0 ( ) [private]

5.2.4.72 void BlockIP::min_Ctv (double ∗ *v*, double ∗ *Ctv*) [private]

Given v[1:km], this routine computes C'∗v= sum{i in 1..k} L_i∗Theta_i∗N_i'∗v_i, which is a vector of dimension l_link. The result vector Ctv only contains values of active linking, components of inactive constraints are 0.

Parameters

v[1:km],:	input vector
Ctv[1:l_link],:	output vector containing C'∗v

5.2.4.73 void BlockIP::min_Cv (double ∗ *v*, double ∗ *Cv*) [private]

Given v[l], this routine computes C∗v= [N_i∗Theta_i∗L_i'∗v i=1..k], C∗v is a vector [1:km]. Product L_i'∗v for all components is made assuming that inactive components of v or inactive rows of L are 0 (otherwise, there is a bug in the code and the function will provide wrong results...)

Parameters

v[1:l_link],:	input vector
Cv[1:km],:	output vector containing C∗v

5.2.4.74 void BlockIP::min_debug_variables_of_inactiveInk ( ) [private]

For debugging purposes, check variables and data of inactive slacks and constraints are 0

5.2.4.75 void BlockIP::min_debug_variables_without_ub ( ) [private]

For debugging purposes, check variables and data of variables without upper bounds are 0

5.2.4.76 void BlockIP::min_debug_write_variables ( ) [private]

For debugging purposes, write variables to file

5.2.4.77 void BlockIP::min_free_memory ( ) [private]

Deletes arrays locally needed for the interior-point algorithm

5.2.4.78 void BlockIP::min_initializations ( ) [private]

Initialize parameters, tolerances, etc for BlockIP minimization algorithm.

5.2.4.79 void BlockIP::min_KKT_residuals ( ) [private]

Compute the residuals of mu-KKT equations Ax=b (primal feasibility) A'∗y+z-w-Gx=0 (dual feasibility) XZe= sigma∗mu∗e (complementarity x∗z) SWe= sigma∗mu∗e (complementarity s∗w)

The residuals are:

- rb[km+l]= b-Ax : primal infeasibility. Positions of inactive linking constraints are set to 0.
- rc[kn+l]= Gx-A'y-z+w : dual infeasibility. Positions of slacks of inactive linking constraints are set to 0. Gx is the gradient of the objective function. If it is nonlinear, is stored in Gx. If it is linear Gx=c. If it is quadratic, Gx= Qx+c. If QUAD_REG regularization then Qreg∗x term has bo be added to rc (rc=Gx+qreg∗x-A'y-z+w) only for block variables.
- rxz[kn+l]= sigma∗mu∗e- X∗z : complementarity residual for Xz= mu∗e, decreasing mu (see below) by sigma. Positions of inactive linking constraints are set to 0.
- rsw[kn+l]= sigma∗mu∗e- S∗w : complemetarity residual for Sw= mu∗e, decreasing mu (see below) by sigma. Positions of inactive linking constraints are set to 0.
- normrc: norm of rc weighted by (1+|c|)
- normrb: norm of rb weighted by (1+|b|)
- mu= (x'z+s'w)/(kn+length(activelnk)) : centrality parameter

Residuals rxz and rsw only needed if NEWTON direction is used. SECOND ORDER direction do not need rxz and rsw (predictor-corrector will compute later their own right-hand-sides for the complementarity equations).

5.2.4.80 void BlockIP::min_Newton_direction ( ) [private]

Computes the standard Newton direction by solving normal equations A∗Theta∗A' according to type_comp_dy:

The procedure:

For the primal convex nonlinear problem

(P) min f(x) subject to Ax=b, -x<=0, x-u<=0

the Lagrangian is

L(x,y,z,w)= f(x)+y'∗(b-Ax)+z'∗(-x)+w'∗(x-u)

and the gradient respect to x is (Gx is gradient of f(x); Gx=c if f(x) is linear, Gx= Qx+c if f(x) is quadratic):

L(x,y,z,w)= Gx - A'∗y - z + w

and the dual problem is

(D) min L(x,y,z,w) s. to L(x,y,z,w)=0 w>=0, z>=0

The condition L(x,y,z,w)=0 is the dual feasibility below.

The mu-KKT conditions are (Gx is gradienf of f(x)):

A'∗y+z-w-Gx=0 (dual feasibility) Ax=b (primal feasibility) XZe= sigma∗mu∗e (complementarity x∗z) SWe= sigma∗mu∗e (complementarity s∗w)

The Newton system is (S= U-X, and Hx is Hessian of f(x))

| -Hx A' I -I | |dx| |rc | | A | |dy| = |rb | | Z X | |dz| |rxz| | -W S | |dw| |rsw|

which is solved by:

Theta= (Hx + S^{-1}∗W + X^{-1}∗Z)^{-1}

r= rc + S^{-1}∗rsw - X^{-1}∗rxz (A∗Theta∗A')dy= rb+A∗Theta∗r dx= Theta∗(A'*dy-r) dw= S^{-1}*(rsw + W∗dx) dz= rc + dw + Hx∗dx - A'∗dy

If QUAD_REG regularization is applied then Gx is (Gx+Qreg∗x) and Hx is (Hx+Qreg). rc and Theta were previosuly computed considering (Gx+Qreg∗x) and (Hx+Qreg). Here we have to use (Hx+Qreg) too.

5.2.4.81 void BlockIP::min_normb_normc ( ) [private]

Compute the L2 norm of b[] and c[]. For the linking constraints, only the active are considered.

5.2.4.82 void BlockIP::min_objective_functions_linquad ( ) [private]

5.2.4.83 void BlockIP::min_objective_functions_nonlin ( ) [private]

5.2.4.84 bool BlockIP::min_optimal ( ) [private]

5.2.4.85 int BlockIP::min_PCG_Hv (int *nn,* double ∗ *v,* double ∗ *Hv*)

Computes Hv=(D-C'∗B^(-1)∗C)v Returns vector Hv of dimension l_link, assuming components of inactives are 0 (if there is no error in the code). Then we avoid using (packing and unpacking) a vector of dimension numActiveLnk.

Parameters

v[1:l_link],:	input vector
Hv[1:l_link],:	output vector containing H∗v

5.2.4.86 int BlockIP::min_PCG_Hz_eq_r (int *nn,* double ∗ *zz,* double ∗ *rr*)

Approximate solution of H∗zz = rr, using m_pw_prec terms of the power series expansion H^{-1}= [sum{i=0..infinity} (D^{-1}∗(C'B^{-1}B)C)^i]∗D^{-1} Recommended values are m_pw_prec= 1 or 2, to avoid expensive computations We use vectors of dimension l_link, assuming components of inactives are 0 (if there is no error in the code). Then we avoid using (packing and unpacking) a vector of dimension numActiveLnk.

Parameters

rr[1:l_link],:	rhs of system of equations
zz[1:l_link],:	solution of the system

5.2.4.87 int BlockIP::min_PCG_Theta0z_eq_r (int *nn,* double ∗ *zz,* double ∗ *rr*)

Solves Theta0∗zz = rr, where Theta0 are the components of Theta associated to active linking constraints. This preconditioner may be useful when the space of linking constraints is "close" (principal angles not large) to the space of block constraints. We use vectors of dimension l_link, assuming components of inactives are 0 (if there is no error in the code). Then we avoid using (packing and unpacking) a vector of dimension numActiveLnk.

Parameters

rr[1:l_link],:	rhs of system of equations
zz[1:l_link],:	solution of the system

5.2.4.88 void BlockIP::min_preprocess ( ) [private]

Simple check for

- Order matrices stored in general packed format
- Convert general matrices stored in column-wise packed format to row-wise packed format
- no-zero upper bounds (variables and slacks) not allowed. Set a very small upper bound to guarantee an interior solution. Zero variables are not removed to preserve the topology of the constraints for each block then we can maintain one single symbolic factorization.
- q positive semidefinite; is set to 0 if problem is linear
- initialization of activelnk if empty
- initialization of free vars arrays if empty and all variables are unfree
- check infinity upper bounds
- check no rhs term is infinity (otherwise the algorithm will fail)

5.2.4.89 void BlockIP::min_restore_inactive (int *lnk*) [private]

5.2.4.90 void BlockIP::min_second_order_predictor_corrector_direction ( ) [private]

Computes the second order predictor-corrector Mehrotra direction by solving normal equations A∗Theta∗A' according to type_comp_dy:

The procedure:

For the primal convex nonlinear problem

(P) min f(x) subject to Ax=b, -x<=0, x-u<=0

the Lagrangian is

L(x,y,z,w)= f(x)+y'∗(b-Ax)+z'∗(-x)+w'∗(x-u)

and the gradient respect to x is (Gx is gradient of f(x); Gx=c if f(x) is linear, Gx= Qx+c if f(x) is quadratic):

L(x,y,z,w)= Gx - A'∗y - z + w

and the dual problem is

(D) min L(x,y,z,w) s. to L(x,y,z,w)=0 w>=0, z>=0

The condition L(x,y,z,w)=0 is the dual feasibility below.

The mu-KKT conditions are (Gx is gradienf of f(x)):

A'∗y+z-w-Gx=0 (dual feasibility) Ax=b (primal feasibility) XZe= sigma∗mu∗e (complementarity x∗z) SWe= sigma∗mu∗e (complementarity s∗w)

The predictor-corrector direction is computed as (S= U-X, and Hx is Hessian of f(x)):

1. Predictor direction

| -Hx A' I -I | |dx_p| |rc | | A | |dy_p| = |rb | | Z X | |dz_p| |-XZe| | -W S | |dw_p| |-SWe|

1. Compute sigma and psi

1. Predictor+Corrector direction

| -Hx A' I -I | |dx| |rc | | A | |dy| = |rb | | Z X | |dz| |sigma∗mu-XZe-Dx_p Dz_p e| | -W S | |dw| |sigma∗mu-SWe-Ds_p Dw_p e|

Each of the above Newton-like systems are solved as:

| -Hx A' I -I | |dx| |rc | | A | |dy| = |rb | | Z X | |dz| |rhs_xz| | -W S | |dw| |rhs_sw|

Theta= (Hx + S^{-1}∗W + X^{-1}∗Z)^{-1}

r= rc + S^{-1}∗rhs_sw - X^{-1}∗rhs_xz (A∗Theta∗A')dy= rb+A∗Theta∗r dx= Theta∗(A'*dy-r) dw= S^{-1}*(rhs_sw + W∗dx) dz= rc + dw + Hx∗dx - A'∗dy

If QUAD_REG regularization is applied then Gx is (Gx+Qreg∗x) and Hx is (Hx+Qreg). rc and Theta were previosuly computed considering (Gx+Qreg∗x) and (Hx+Qreg). Here we have to use (Hx+Qreg) too.

5.2.4.91 void BlockIP::min_solve_NThetaNt (double ∗ *v*) [private]

Solves k_blocks systems N_i∗Theta_i∗N_i' dy_i= rhsdy_i i=1..k_blocks where N_i can be different matrices or the same one (if sameN).

Parameters

v[1:km],:	on input contains the righ-hand-side for the k_blocks systems
v[1:km],:	on output it contain the solutions to the k_blocks systems

5.2.4.92 void BlockIP::min_starting_point ( ) [private]

Computation of starting point:

1. simple initialization
2. Mehrotra-like initilization (solves quadratic equality constrained problem)

5.2.4.93 void BlockIP::min_steplengths (double & *talpha_p,* double & *talpha_d,* double *tdx[],* double *tdz[],* double *tdw[]*) [private]

5.2.4.94 void BlockIP::min_test_newpoint ( ) [private]

5.2.4.95 void BlockIP::min_update_inactiveInk ( ) [private]

Detect if new linking constraints must be considered inactive This is only done if we are close enough to the optimal solution; we use (gap $<$ 1.0), where gap was computed before as abs(pobj-dobj)/(1+abs(pobj)). This only works for linear/quadr. functions, but for nonlinear ones this update is not used (see below an explanation). Constraint i inactived if its slack x(i) is out of bounds and lagrange multiplier is close to 0: (x(i)$>>$0 and x(i)$<<$u(i) and y(i)$\sim$0) and (c(i)==0 and q(i)==0). The term (c(i)==0 and q(i)==0) guarantees that slacks appearing in the objective function are not removed. This could even work for nonlinear functions: if component of gradient and Hessian of slack is 0, the slack could be removed (∗∗∗∗ be careful: this can provide strange results if for some reason the first and second derivatives are 0 only in that iteration; for safety, the inactivation is not performed for nonlinear objective functions ∗∗∗) We also avoid inactivation of constraints with small bounds on slacks (i.e, u$<$0.1), since these constraints are always quasi-active

5.2.4.96 void BlockIP::min_update_newpoint ( ) [private]

5.2.4.97 void BlockIP::min_update_qreg ( ) [private]

Updates regularization term

5.2.4.98 void BlockIP::min_write_current_point_info ( ) [private]

5.2.4.99 void BlockIP::min_write_problem_information ( ) [private]

5.2.4.100 int BlockIP::minimize ( )

5.2.4.101 void BlockIP::read_BlockIP_format (const char * *filename*)

Create a problem from a BlockIP format file.

Parameters

filename	File name with extension

Note

The problem must be converted to standard form in order to be solved, it will be converted automatically when minimize function is called. After that call the problem only may be written in standard form. Before that call the problem can be written in standard form with convert_to_std_and_write_mps function or in general form with write_mps function. !

5.2.4.102 void BlockIP::read_mps (const char * *filename*)

Create a problem from a mps file.

Parameters

filename	File name with extension

Note

Token : separate the block name and the variable or constraint name When a constraint or variable does not have a block name it is considered a linking constraint or a slack
If some variable have as a name "Constant" with no block, this variable is considered a constant to add in the objective function
Slacks are optional, but if one is defined, all slacks must be defined, to relate an slack to one linking constraint, this slack must have a 1 coeficient in the related linking constraint.
In QUADOBJ only cannot be relation between two different variables.
The problem must be converted to standard form in order to be solved, it will be converted automatically when minimize function is called. After that call the problem only may be written in standard form. Before that call the problem can be written in standard form with convert_to_std_and_write_mps function or in general form with write_mps function. !

5.2.4.103 void BlockIP::set_constant_fobj (double *constant*) [inline]

Set the constant to add to the objective function.

5.2.4.104 void BlockIP::set_deactivateLnk (bool *deactivateLnk* = DEACTIVATELNK) [inline]

Set flag on deactivation of linking.

5.2.4.105 void BlockIP::set_defaults_minimize ( )

Set to default values all the parameters that control the interior-point algorithm

5.2.4.106 void BlockIP::set_factor_reg (double *factor_reg* = **FACTOR_REG_DEFAULT**) [inline]

Set factor of regularization.

5.2.4.107 void BlockIP::set_inf (double *inf* = **INF**) [inline]

Set threshold value to be used as infinity (x>inf -> x is inf)

5.2.4.108 void BlockIP::set_init_pcgtol (double *init_pcgtol*) [inline]

Set initial tolerance for the conjugate gradient (by default PCG_TOL_LIN if linear problem, PCG_TOL_QUAD if quadratic or -convex- nonlinear)

5.2.4.109 void BlockIP::set_m_pw_prec (int *m_pw_prec* = **M_PW_PREC**) [inline]

Set number of terms used as preconditioner of the power series expansion of (D-C'∗B^{-1}∗B)^{-1}.

5.2.4.110 void BlockIP::set_maxit_pcg (double *maxit_pcg* = 0) [inline]

Set maximum number of pcg iterations (if 0, then the value will be computed by the code)

5.2.4.111 void BlockIP::set_maxiter (int *maxiter* = **MAXITER**) [inline]

Set maximum number of IP iterations.

5.2.4.112 void BlockIP::set_min_pcgtol (double *min_pcgtol* = **MIN_PCGTOL**) [inline]

Set minimum tolerance for the conjugate gradient.

5.2.4.113 void BlockIP::set_names (string ∗ *blockNames*, string ∗ *varNames*, string ∗ *consNames*, bool *copy_vectors* = true)

Set the names of the problem TODO change copy_vectors.

Parameters

blockNames	Block names of N blocks
varNames	Variable names including slacks
consNames	Constraint names including linking constraints
copy_vectors	If is true the user must free the memory of arguments (arrays), If false the user must allocate the memory with new, after this call the user will not have control of the arguments (arrays) anymore and MatrixBlockIP will delete the arguments

Note

If some variable is NULL BlockIP keeps its own name

5.2.4.114 void BlockIP::set_optim_dfeas (double *optim_dfeas* = OPTIM_DFEAS) [inline]

Set dual feasibility tolerance.

5.2.4.115 void BlockIP::set_optim_gap (double *optim_gap* = OPTIM_GAP) [inline]

Set optimality gap tolerance.

5.2.4.116 void BlockIP::set_optim_pfeas (double *optim_pfeas* = OPTIM_PFEAS) [inline]

Set primal feasibility tolerance.

5.2.4.117 void BlockIP::set_output (OUTPUT *output* = SCREEN) [inline]

Set type of output.

5.2.4.118 void BlockIP::set_output_freq (int *output_freq* = OUTPUT_FREQ) [inline]

Set output information lines will be printed each output_freq IP iterations.

5.2.4.119 void BlockIP::set_red_pcgtol (double *red_pcgtol* = RED_PCGTOL) [inline]

Set reduction of pcg_tol at each IP iteration.

5.2.4.120 void BlockIP::set_rho (double *rho* = RHO) [inline]

Set reduction of the step-length for the primal and dual variables at each IP iteration.

5.2.4.121 void BlockIP::set_show_specrad (bool *show_specrad* = false) [inline]

Set show_specrad at each IP iteration.

5.2.4.122 void BlockIP::set_sigma (double *sigma* = SIGMA) [inline]

Set reduction of the centrality parameter at each IP iteration.

5.2.4.123 void BlockIP::set_type_comp_dy (TYPE_COMP_DY *type_comp_dy* = TYPE_COMP_DY_DEFAULT) [inline]

Set how dy direction is computed.

5.2.4.124 void BlockIP::set_type_direction (TYPE_DIRECTION *type_direction* = TYPE_DIRECTION_DEFAULT) [inline]

Set which direction is computed.

5.2.4.125 void BlockIP::set_type_reg (TYPE_REG *type_reg* = TYPE_REG_DEFAULT) [inline]

Set type of regularization.

5.2.4.126 void BlockIP::set_type_start_point (TYPE_START_POINT *type_start_point* = TYPE_START_POINT_DEFAULT) [inline]

Set how starting point is computed.

5.2.4.127 void BlockIP::set_whoperm (WHO_PERMUTES *whoperm_* = CHOLESKY) [inline]

Set whopermutes constraints-related information: the Cholesky factorization of the user of it.

5.2.4.128 void BlockIP::write_BlockIP_format (const char * *filename*)

Write the problem in BlockIP format file.

Parameters

filename	File name without extension

5.2.4.129 void BlockIP::write_mps (const char * *filename*)

Write a mps file.

5.2.4.130 void BlockIP::write_mps (const char * *filename*, TYPE_PROBLEM *type_objective*, double *cost[]*, double *qcost[]*, double *lb[]*, double *ub[]*, double *lhs[]*, double *rhs[]*, int *numBlocks*, bool *sameN*, MatrixBlockIP *N[]*, bool *sameL*, MatrixBlockIP *L[]*, string * *blockNames* = NULL, string * *varNames* = NULL, string * *consNames* = NULL, double *constantFObj* = 0)

Write a mps file.

Parameters

filename	File name without extension
type_objective	Type of objective function, linear or quadratic
cost	Linear cost of variables including slacks
qcost	Quadratic cost of variables including slacks
lb	Lower bounds, including slack bounds
ub	Upper bounds, including slack bounds
lhs	Lower constraint limits, including linking constraints limits
rhs	Upper constraint limits, including linking constraints limits
numBlocks	Number of diagonal blocks
sameN	Define if the same matrix is used for each N block. If true array N must have dimension 1
N	Diagonal blocks
sameL	Define if the same matrix is used for each L block If true array L must have dimension 1
L	Linking constraints blocks
blockNames	Block names of N blocks
varNames	Variable names including slacks
consNames	Constraint names including linking constraints

5.2.4.131 void BlockIP::write_problem (const char * *filename*)

Write all data related to the problem into a file.

Parameters

filename	File name where output the data

5.2.5 Member Data Documentation

5.2.5.1 MatrixBlockIP* BlockIP::A

Full matrix (likely made from N and L)

5.2.5.2 double BlockIP::alpha_d [private]

step lengths

5.2.5.3 double BlockIP::alpha_p [private]

5.2.5.4 double * BlockIP::Aty [private]

5.2.5.5 double* BlockIP::Ax [private]

5.2.5.6 BackupLnk* BlockIP::backupLnk [private]

5.2.5.7 string* BlockIP::blockNames

Block names of N blocks.

5.2.5.8 int BlockIP::cgit [private]

number of CG iterations of this IPM iteration

5.2.5.9 bool BlockIP::comp_specrad [private]

Whether the estimation of spectral radius has to be shown in the output.

5.2.5.10 string* BlockIP::consNames

Constraint names including linking constraints.

5.2.5.11 double BlockIP::constantFObj

Constant to add in the objective function.

5.2.5.12 double* BlockIP::cost

Linear cost of variables including slacks.

5.2.5.13 double * BlockIP::Cvaux [private]

5.2.5.14 double * BlockIP::Cvaux2 [private]

Auxiliary vectors for interior-point algorithm.

5.2.5.15 MatrixBlockIP* BlockIP::D

D = Theta_(k+1) + sum{i..k} L_i*Theta_i*L_i'.

5.2.5.16 const bool BlockIP::DEACTIVATELNK = true [static], [private]

5.2.5.17 bool BlockIP::deactivateLnk [private]

Vectors to store information from PCG to later compute Ritz values.
Setting deactivateLnk= false the user may avoid the deactivation of linking

5.2.5.18 bool BlockIP::deleteMatrices [private]

To delete matrices when created by BlockIP (read_mps)

5.2.5.19 double BlockIP::dobj

dual objective

5.2.5.20 double * BlockIP::dsdw [private]

for rhs of corrector system, from dx,dw,ds and dw of predictor direction

5.2.5.21 double * BlockIP::dw [private]

Primal and dual optimization directions.

5.2.5.22 double* BlockIP::dx [private]

5.2.5.23 double* BlockIP::dxdz [private]

5.2.5.24 double * BlockIP::dy [private]

5.2.5.25 double * BlockIP::dz [private]

5.2.5.26 double BlockIP::epsmach [private]

Stores computed machine epsilon.

5.2.5.27 double BlockIP::est_specrad [private]

Estimation of spectral radius of D^{-1}C'*B^{-1}*B (computed through Ritz values)

5.2.5.28 double BlockIP::factor_reg [private]

Type of regularization performed.

5.2.5.29 constexpr double BlockIP::FACTOR_REG_DEFAULT = 1.0e-6 [static], [private]

5.2.5.30 void(* BlockIP::fobj)(int n, double x[], double &fx, double Gx[], double Hx[], void *params)

User function to calculate the objective function in a point.

5.2.5.31 double BlockIP::fx

5.2.5.32 double BlockIP::gap [private]

Duality gap.

5.2.5.33 double * BlockIP::Gx

5.2.5.34 double * BlockIP::Hx

Objective function value, Gradient and Hessian in the actual point.

5.2.5.35 constexpr double BlockIP::INF = 1.0e128 [static], [private]

5.2.5.36 double BlockIP::inf [private]

Infinity value.

For the default values see the constant terms in BlockIP.h

5.2.5.37 int* BlockIP::ini_m

Begin to blocks 1..k and linking for constraints.

5.2.5.38 int* BlockIP::ini_n

Begin to blocks 1..k and linking for variables.

5.2.5.39 double BlockIP::init_pcgtol [private]

Initial tolerance for the conjugate gradient (by default PCG_TOL_LIN if linear problem, PCG_TOL_QUAD if quadratic or -convex- nonlinear)

5.2.5.40 bool BlockIP::initialized [private]

To control if the attributes have been initialized.

5.2.5.41 int BlockIP::input_problem [private]

5.2.5.42 bool BlockIP::inStdForm

To control if the problem is in standard form.

5.2.5.43 bool* BlockIP::isActiveLnk [private]

linking i is active if isActiveLnk[i] is true

5.2.5.44 bool* BlockIP::isFreeVar [private]

Variable i is marked as free if isFreeVar[i] is true.

5.2.5.45 int BlockIP::it [private]

IPM iterations.

5.2.5.46 int BlockIP::k_blocks

Number of diagonal blocks.

5.2.5.47 bool BlockIP::keepStdFormAfterDelete [private]

To keep stdForm when BlockIP will delete.

5.2.5.48 int BlockIP::km

Number of constraints without linking constraints.

5.2.5.49 int BlockIP::kn

Number of variables without slacks of linking.

5.2.5.50 MatrixBlockIP* BlockIP::L

Linking constraints blocks.

5.2.5.51 int BlockIP::l_link

Number of linking constraints.

5.2.5.52 double* BlockIP::lb

Lower bounds, including slack bounds.

5.2.5.53 double* BlockIP::lhs

Lower constraint limits, including linking constraints limits.

5.2.5.54 int* BlockIP::listActiveLnk [private]

list of active linking, j=listActiveLnk[i], i=1..numActiveLnk, j in {1,..,l_link})

5.2.5.55 int* BlockIP::listFreeVars [private]

List of variables marked as free, j=listFreeVars[i], i=1..numFreeVars, j in {1,..,n_vars})

5.2.5.56 int* BlockIP::listInactiveLnk [private]

list of inactive linking, j=listInactiveLnk[i], i=1..numInactiveLnk, j in {1,..,l_link})

5.2.5.57 int* BlockIP::listUnfreeVars [private]

List of variables not marked as free, j=listUnfreeVars[i], i=1..numUnfreeVars, j in {1,..,n_vars})

5.2.5.58 int* BlockIP::listWithoutUb [private]

5.2.5.59 int* BlockIP::listWithUb [private]

5.2.5.60 int BlockIP::m_cons

Number of constraints including linking constraints.

5.2.5.61 const int BlockIP::M_PW_PREC = 1 [static], [private]

5.2.5.62 int BlockIP::m_pw_prec [private]

Number of terms of the power series expansion of (D-C'*B^{-1}*B)^{-1} used as preconditioner.

5.2.5.63 double BlockIP::maxit_pcg [private]

Maximum number of pcg iterations (if 0, then the value will be computed by the code)

5.2.5.64 const int BlockIP::MAXITER = 200 [static], [private]

5.2.5.65 int BlockIP::maxiter [private]

Maximum number of IP iterations.

5.2.5.66 constexpr double BlockIP::MIN_PCGTOL = 1.0e-8 [static], [private]

5.2.5.67 double BlockIP::min_pcgtol [private]

Minimum tolerance for the conjugate gradient.

5.2.5.68 double BlockIP::mu [private]

5.2.5.69 double BlockIP::mu0 [private]

Centrality parameter; mu0 is value of mu at starting point.

5.2.5.70 MatrixBlockIP* BlockIP::N

Diagonal blocks.

5.2.5.71 int BlockIP::n_vars

Number of variables including slacks.

5.2.5.72 double BlockIP::normb [private]

5.2.5.73 double BlockIP::normc [private]

5.2.5.74 double BlockIP::normrb [private]

5.2.5.75 double BlockIP::normrc [private]

norms of rhs, costs, rb, and rc

5.2.5.76 int BlockIP::numActiveLnk [private]

Number of active lnk.

5.2.5.77 int BlockIP::numBackupLnk [private]

5.2.5.78 int BlockIP::numFreeVars [private]

Number of variables marked as free.

5.2.5.79 int BlockIP::numInactiveLnk [private]

Number of inactive lnk.

5.2.5.80 int BlockIP::numInactiveLnkWithUb [private]

Number of inactive lnk with upper bound.

5.2.5.81 int BlockIP::numUnfreeVars [private]

Number of variables not marked as free.

5.2.5.82 int BlockIP::numWithoutUb [private]

5.2.5.83 int BlockIP::numWithUb [private]

Number of variables with upper bound (not infinity)

5.2.5.84 constexpr double BlockIP::OPTIM_DFEAS = 1.0e-6 [static], [private]

5.2.5.85 double BlockIP::optim_dfeas [private]

Dual feasibility tolerance.

5.2.5.86 constexpr double BlockIP::OPTIM_GAP = 1.0e-6 [static], [private]

5.2.5.87 double BlockIP::optim_gap [private]

Optimality gap tolerance.

5.2.5.88 constexpr double BlockIP::OPTIM_GAP_SAFEGUARD = 1.0e-5 [static], [private]

5.2.5.89 constexpr double BlockIP::OPTIM_PFEAS = 1.0e-6 [static], [private]

5.2.5.90 double BlockIP::optim_pfeas [private]

Primal feasibility tolerance.

5.2.5.91 int* BlockIP::origNCons

Number of original constraints for each block.

5.2.5.92 int* BlockIP::origNVars

Number of original variables for each block.

5.2.5.93 OUTPUT BlockIP::output [private]

Type of output.

5.2.5.94 const int BlockIP::OUTPUT_FREQ = 1 [static], [private]

5.2.5.95 int BlockIP::output_freq [private]

Output information lines will be printed each output_freq IP iterations.

5.2.5.96 double * BlockIP::p_cg [private]

Auxiliary vectors for PCG.

5.2.5.97 void* BlockIP::params

User parameters to perform objective function calculations.

5.2.5.98 constexpr double BlockIP::PCG_TOL_LIN = 1.0e-2 [static], [private]

5.2.5.99 constexpr double BlockIP::PCG_TOL_QUAD = 1.0e-3 [static], [private]

5.2.5.100 double BlockIP::pcgtol [private]

Tolerance for conjugate gradient at current iteration.

5.2.5.101 double BlockIP::psi [private]

Reduction of dx*dz vector in the rhs of corrector system (in predictor-corrector)

5.2.5.102 double* BlockIP::qcost

Quadratic cost of variables including slacks.

5.2.5.103 double BlockIP::qreg [private]

Regularization term.

5.2.5.104 double * BlockIP::r [private]

5.2.5.105 double* BlockIP::r_cg [private]

5.2.5.106 double* BlockIP::rb [private]

5.2.5.107 double * BlockIP::rc [private]

5.2.5.108 constexpr double BlockIP::RED_PCGTOL = 0.95 [static], [private]

5.2.5.109 double BlockIP::red_pcgtol [private]

Reduction of pcg_tol at each IP iteration.

5.2.5.110 constexpr double BlockIP::RHO = 0.995 [static], [private]

5.2.5.111 double BlockIP::rho [private]

Reduction of the step-length for the primal and dual variables at each IP iteration.

5.2.5.112 double* BlockIP::rhs

Upper constraint limits, including linking constraints limits.

5.2.5.113 double * BlockIP::rhsdy [private]

5.2.5.114 double * BlockIP::rhspcg [private]

5.2.5.115 double * BlockIP::rsw [private]

residuals of KKT conditions

5.2.5.116 double * BlockIP::rxz [private]

5.2.5.117 double * BlockIP::s [private]

primal and dual optimization variables

5.2.5.118 bool BlockIP::sameL

Define if the same matrix is used for each L block.

5.2.5.119 bool BlockIP::sameN

Define if the same matrix is used for each N block.

5.2.5.120 bool BlockIP::show_specrad [private]

5.2.5.121 constexpr double BlockIP::SIGMA = 0.1 [static], [private]

5.2.5.122 double BlockIP::sigma [private]

Reduction of the centrality parameter at each IP iteration.

5.2.5.123 StdForm* BlockIP::stdForm

Contains all the information to perform conversions between the original and standard problem.

5.2.5.124 double* BlockIP::Theta [private]

Theta diagonal matrix.

5.2.5.125 MatrixBlockIP* BlockIP::Theta0

Stores Theta0 preconditioner (Theta_(k+1)) of active linking constraints)

5.2.5.126 TYPE_DIRECTION BlockIP::this_type_direction [private]

Type of direction (Newton or second order) of this particular iteration.

5.2.5.127 int BlockIP::totcgit [private]

Overall number of CG iterations.

5.2.5.128 TYPE_COMP_DY BlockIP::type_comp_dy [private]

Whether the estimation of spectral radius has to be computed.

How dy direction is computed

5.2.5.129 const TYPE_COMP_DY BlockIP::TYPE_COMP_DY_DEFAULT = CHOL_PWRS_PCG [static], [private]

5.2.5.130 TYPE_DIRECTION BlockIP::type_direction [private]

Type of direction (Newton, second order...) given by user.

5.2.5.131 const TYPE_DIRECTION BlockIP::TYPE_DIRECTION_DEFAULT = AUTOMATIC [static], [private]

5.2.5.132 TYPE_PROBLEM BlockIP::type_objective

Type of objective function, linear, quadratic or non-linear.

5.2.5.133 TYPE_REG BlockIP::type_reg [private]

How starting point will be computed.

5.2.5.134 const TYPE_REG BlockIP::TYPE_REG_DEFAULT = NO_REG [static], [private]

5.2.5.135 TYPE_START_POINT BlockIP::type_start_point [private]

5.2.5.136 const TYPE_START_POINT BlockIP::TYPE_START_POINT_DEFAULT = SIMPLE [static], [private]

5.2.5.137 double* BlockIP::ub

Upper bounds, including slack bounds.

5.2.5.138 double* BlockIP::valpha [private]

5.2.5.139 string* BlockIP::varNames

Variable names including slacks.

5.2.5.140 double * BlockIP::vbeta [private]

5.2.5.141 double * BlockIP::w [private]

5.2.5.142 WHO_PERMUTES BlockIP::whoperm [private]

for matrix factorizations

5.2.5.143 double* BlockIP::x [private]

initial value for regularization

5.2.5.144 double * BlockIP::y [private]

5.2.5.145 double * BlockIP::z [private]

5.2.5.146 double * BlockIP::z_cg [private]

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

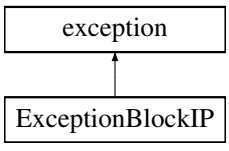
- /home/jcastro2/intpoint/BlockIP/BlockIP/src_blockip-v2/BlockIP.h
- /home/jcastro2/intpoint/BlockIP/BlockIP/src_blockip-v2/BlockIP.C
- /home/jcastro2/intpoint/BlockIP/BlockIP/src_blockip-v2/BlockIPminimize.C

5.3 ExceptionBlockIP Class Reference

Class for BlockIP exceptions.

`#include <ExceptionBlockIP.h>`

Inheritance diagram for ExceptionBlockIP:



Public Member Functions

- ExceptionBlockIP (TYPE_ERROR error) throw ()
- ExceptionBlockIP (TYPE_ERROR error, string file) throw ()
- ExceptionBlockIP (TYPE_ERROR error, string file, int line) throw ()
- ~ExceptionBlockIP () throw ()
- const char * what () const throw ()

Public Attributes

- TYPE_ERROR error
- string file
- int line
- string message

Private Member Functions

- void setMessage ()

5.3.1 Detailed Description

Class for BlockIP exceptions.

5.3.2 Constructor & Destructor Documentation

5.3.2.1 ExceptionBlockIP::ExceptionBlockIP (TYPE_ERROR *error*) throw ()

5.3.2.2 ExceptionBlockIP::ExceptionBlockIP (TYPE_ERROR *error,* string *file*) throw ()

5.3.2.3 ExceptionBlockIP::ExceptionBlockIP (TYPE_ERROR *error,* string *file,* int *line*) throw ()

5.3.2.4 ExceptionBlockIP::~ExceptionBlockIP ( ) throw ()

5.3.3 Member Function Documentation

5.3.3.1 void ExceptionBlockIP::setMessage ( ) [private]

5.3.3.2 const char * ExceptionBlockIP::what ( ) const throw ()

5.3.4 Member Data Documentation

5.3.4.1 TYPE_ERROR ExceptionBlockIP::error

5.3.4.2 string ExceptionBlockIP::file

5.3.4.3 int ExceptionBlockIP::line

5.3.4.4 string ExceptionBlockIP::message

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

- /home/jcastro2/intpoint/BlockIP/BlockIP/src_blockip-v2/ExceptionBlockIP.h
- /home/jcastro2/intpoint/BlockIP/BlockIP/src_blockip-v2/ExceptionBlockIP.C

5.4 MatrixBlockIP Class Reference

Class for manipulating matrices, and interfacing SparseChol.

`#include <MatrixBlockIP.h>`

Classes

- struct Order_ija

  Auxiliary struct for sorting matrices in ija format.
- struct Order_vector

  Auxiliary struct for sorting vectors.

Public Member Functions

- MatrixBlockIP (int blocks=1)

  Constructor.
- MatrixBlockIP (MatrixBlockIP *mbip)

  Copy constructor, does not copy SparseChol.
- ~MatrixBlockIP ()

  Destructor.
- void copy (MatrixBlockIP *mbip)

  Copy, does not copy SparseChol.
- void reset ()

  Comes back to the initial state.
- void restore ()

  Comes back to the state after create the matrix.
- void create_general_matrix_row_wise (int m, int n, int nz, int *&inirowa, int *&icola, double *&a)

  Native method to create a general row-wise packed matrix.
- void create_general_matrix_column_wise (int m, int n, int nz, int *&inicola, int *&irowa, double *&a)

  Native method to create a general column-wise packed matrix.
- void create_network_matrix (int num_arcs, int num_nodes, int *&src, int *&dst, bool oriented=true)

  Native method to create a network matrix.
- void create_identity_matrix (int dim)

  Native method to create a identity matrix of dimension dim.
- void create_idty_idty_matrix (int nrows)

  Native method to create a identity-identity matrix with two identities [I I].
- void create_diagonal_matrix (int dim, double *&d)

  Native method to create a diagonal matrix D= diag(d) of dimension dim.
- void create_diag_diag_matrix (int nrows, double *&d1, double *&d2)

  Native method to create a diagonal-diagonal matrix D= [D1 D2].
- void create_general_matrix_format_ija (int m, int n, int nz, int *&row_index, int *&col_index, double *&values)

  Creates a general matrix in format ija.
- void compute_full_matrix (int numBlocks, bool sameN, MatrixBlockIP N[], bool sameL, MatrixBlockIP L[], int numActiveLnk=0, int *listActiveLnk=NULL)

  Builds a packed rowwise format structured matrix based on diagonal blocks and linking constraints blocks.
- void analyze_D (int numBlocks, bool sameL, MatrixBlockIP L[])
- void analyze_D_diagonal (int numBlocks, bool sameL, MatrixBlockIP L[])
- void analyze_D_gen_sym_uptr (int numBlocks, bool sameL, MatrixBlockIP L[])
- void compute_D (int numBlocks, bool sameL, MatrixBlockIP L[], bool isActive[], int iniTheta[], double Theta[])

*Compute D = Theta_(numBlocks+1) + sum{i..numBlocks} L_i*Theta_i*L_i'.*

- void compute_D_diagonal (int numBlocks, bool sameL, MatrixBlockIP L[], bool isActive[], int iniTheta[], double Theta[])

  *Compute D = Theta_(numBlocks+1) + sum{i in 1..numBlocks} L_i*Theta_i*L_i' when D is diagonal.*
- void compute_D_gen_sym_uptr (int numBlocks, bool sameL, MatrixBlockIP L[], bool isActive[], int iniTheta[], double Theta[])

  *Compute D = Theta_(numBlocks+1) + sum{i..numBlocks} L_i*Theta_i*L_i' when D is a symmetric general matrix.*
- void native_to_general (bool delete_native_format=false)

  Calculates and stores the matrix in packed rowwise format.
- void ija_to_rowwise ()

  Calculates and stores the matrix in packed rowwise format.
- void network_to_general ()

  Calculates and stores the network matrix (either oriented or nonoriented) in packed rowwise format.
- void identity_to_general ()

  Calculates and stores the I matrix in packed rowwise format.
- void diagonal_to_general ()

  Calculates and stores the diagonal D matrix in packed rowwise format.
- void idty_idty_to_general ()

  Calculates and stores the matrix [I I] in packed rowwise format.
- void diag_diag_to_general ()

  Calculates and stores the matrix [D1 D2] in packed rowwise format.
- void network_to_ija_format ()

  Calculates and stores the network matrix in ija format. It considers both oriented and nonoriented cases.
- void identity_to_ija_format ()

  Calculates and stores the identity I matrix in ija format.
- void diagonal_to_ija_format ()

  Calculates and stores the diagonal D1 matrix in ija format.
- void idty_idty_to_ija_format ()

  Calculates and stores the matrix [I I] in ija format.
- void diag_diag_to_ija_format ()

  Calculates and stores the matrix [D1 D2] in ija format.
- void order_matrix ()

  Order the matrix when has been created in row-wise or column-wise format.
- void mul_Mv (double vout[], const double vin[])

  *Matrix-vector product vout=M*vin (vout(m),vin(n)) (driver)*
- void mul_Mtv (double vout[], const double vin[])

  *MatrixTranspose-vector product vout=M(t)*vin (vout(n),vin(m) (driver)*
- void add_mul_Mv (double vout[], const double vin[])

  *Add matrix-vector product vout += M*vin (vout(m),vin(n)) (driver)*
- void add_mul_Mtv (double vout[], const double vin[])

  *Add matrixTranspose-vector product vout += M(t)*vin (vout(n),vin(m) (driver)*
- void exist_var_in_row (int column, bool appear[], double coefs[])

  Given a column determines if exists an element for each row of the matrix, in case of exist returns the value.
- void add_new_column (int size, int irows[], double values[])

  Add a new column into the matrix.
- void add_new_columns (int num_columns, int size[], int *irows[], double *values[])

  Add new columns into the matrix.
- void change_columns_sign (int size, int columns[])

  Changes the sign of some columns.
- void change_rows_sign (int size, int rows[])

  Change the sign of some rows.

- void delete_rows (int size, int rows[])

  Delete some rows.
- int get_column (int column, int *&irows, double *&values, bool invert_sign=false)

  Gets a column of the matrix.
- void symbolic_fact_MMt (CHOL_SOLVER chslv=SPRSBLKLLT, int *prov_pfa=NULL, int *prov_ipfa=NULL, NUMBERING prov_pfa_numbering=NOT_COMPUTED)

  Interface routine to SparseChol symbolic factorization.
- void numeric_fact_MMt (double *Theta, int i_k=0)

  Interface routine to SparseChol numeric factorization.
- void numeric_solve_MMt (double *rhs, int i_k=0, WHO_PERMUTES whoperm=CHOLESKY)

  Interface routine to SparseChol numeric solve.
- void symbolic_fact_M (CHOL_SOLVER chslv=SPRSBLKLLT, int *prov_pfa=NULL, int *prov_ipfa=NULL, NUMBERING prov_pfa_numbering=NOT_COMPUTED)

  Interface routine to SparseChol symbolic factorization.
- void numeric_fact_M ()

  Interface routine to SparseChol numeric factorization.
- void numeric_solve_M (double *rhs, WHO_PERMUTES whoperm=CHOLESKY)

  Interface routine to SparseChol numeric solve.
- int get_pfa (int i)

  Interface routine to SparseChol get_pfa().
- int get_ipfa (int i)

  Interface routine to SparseChol get_ipfa().
- int get_maxlnz ()

  Interface routine to SparseChol get_maxlnz().
- int get_maxfillin ()

  Interface routine to SparseChol get_maxfillin().
- int get_njka ()

  Interface routine to SparseChol get_njka().
- int get_num_zero_pivots ()

  Interface routine to SparseChol get_num_zero_pivots().
- int get_num_semidef_matrix ()

  Interface routine to SparseChol get_semidef_matrix().
- void print_matrix (bool ija=true, bool start_one=true)

  Print the matrix.
- void print_matrix (ofstream &outfile, bool print_ija=true, bool start_one=true)

  Write the matrix into a file.
- void print_vector (double *v, int size, string name)

  Print some positions and name of a vector.
- void column_wise_to_row_wise_format ()

  Convert a matrix in column-wise format to row-wise format.
- void row_wise_to_column_wise_format ()

  Convert a matrix in column-wise format to row-wise format.

Public Attributes

- TYPE_MATRIX type

  Type of matrix.
- int m

  Number of rows.
- int n

  Number of columns.

- bool ija

  True if irowa, icola and a are stored.
- bool rowwise

  True if inirowa, icola and a are stored.
- bool columnwise

  True if the matrix is stored in column-wise packed format, incompatible with rowwise.
- int nz

  Number of non-zero elements.
- double * a

  Value of non-zero elements.
- int * inirowa

  Index to the first element of each row.
- int * icola

  Column position for each element.
- int * inicola

  Index to the first element of each column.
- int * irowa

  Row position for each element.
- TYPE_ORIENTATION type_orientation

  Type of orientation, by default, oriented.
- int num_arcs

  Number of arcs.
- int num_nodes

  Number of nodes.
- int * src

  Source of each arc.
- int * dst

  Destination of each arc.
- double * d1

  d1 for DIAGONAL and 1st submatrix of DIAG_DIAG
- double * d2

  d2 for 2nd submatrix of DIAG_DIAG
- SparseChol chol

  Sparse Cholesky class.
- int sizeL
- int * inirowLLt
- int * icolLLt
- int * blockD
- int * inivalD
- int * icolD
- double * valD
- bool made_symbfct_MMt
- bool made_symbfct_M
- bool made_analyze_D
- int num_blocks

Private Member Functions

- void free_memory ()

  Free all the memory allocated by the application - not the user.
- void mul_Mv_row_wise (double vout[], const double vin[])

  Matrix-vector product vout=M∗vin (vout(m),vin(n)) (general rowwise matrix)
- void mul_Mtv_row_wise (double vout[], const double vin[])

  MatrixTranspose-vector product vout=M(t)∗vin (vout(n),vin(m) (general rowwise matrix)
- void add_mul_Mv_row_wise (double vout[], const double vin[])

  Add matrix-vector product vout += M∗vin (vout(m),vin(n)) (general rowwise matrix)
- void add_mul_Mtv_row_wise (double vout[], const double vin[])

  Add matrixTranspose-vector product vout += M(t)∗vin (vout(n),vin(m) (general rowwise matrix)
- void mul_Mv_column_wise (double vout[], const double vin[])

  Matrix-vector product vout=M∗vin (vout(m),vin(n)) (general columnwise matrix)
- void mul_Mtv_column_wise (double vout[], const double vin[])

  MatrixTranspose-vector product vout=M(t)∗vin (vout(n),vin(m) (general columnwise matrix)
- void add_mul_Mv_column_wise (double vout[], const double vin[])

  Add matrix-vector product vout += M∗vin (vout(m),vin(n)) (general columnwise matrix)
- void add_mul_Mtv_column_wise (double vout[], const double vin[])

  Add matrixTranspose-vector product vout += M(t)∗vin (vout(n),vin(m) (general columnwise matrix)
- void mul_Mv_network (double vout[], const double vin[])

  Matrix-vector product vout=M∗vin (vout(m),vin(n)) (network matrix, either oriented or nonoriented)
- void mul_Mtv_network (double vout[], const double vin[])

  MatrixTranspose-vector product vout=M(t)∗vin (vout(n),vin(m) (network matrix, either oriented or nonoriented)
- void add_mul_Mv_network (double vout[], const double vin[])

  Add matrix-vector product vout += M∗vin (vout(m),vin(n)) (network matrix, either oriented or nonoriented)
- void add_mul_Mtv_network (double vout[], const double vin[])

  Add matrixTranspose-vector product vout += M(t)∗vin (vout(n),vin(m) (network matrix, either oriented or nonoriented)
- void mul_Mv_identity (double vout[], const double vin[])

  Identity-vector product vout=M∗vin (vout(m),vin(n))
- void add_mul_Mv_identity (double vout[], const double vin[])

  Add Identity-vector product vout += M∗vin (vout(m),vin(n))
- void mul_Mv_diagonal (double vout[], const double vin[])

  Diagonal-vector product vout=D∗vin (vout(m),vin(n)) D is diagonal, m=n.
- void add_mul_Mv_diagonal (double vout[], const double vin[])

  Add Diagonal-vector product vout += D∗vin (vout(m),vin(n)) D is diagonal, m=n.
- void mul_Mv_idty_idty (double vout[], const double vin[])

  Matrix-vector product vout= [I I]∗vin (vout(m),vin(n)) (M= [I I] matrix, n= 2∗m)
- void mul_Mtv_idty_idty (double vout[], const double vin[])

  MatrixTranspose-vector product vout= [I I]'∗vin (vout(n),vin(m) (M= [I I] matrix, n= 2∗m)
- void add_mul_Mv_idty_idty (double vout[], const double vin[])

  Add matrix-vector product vout += [I I]∗vin (vout(m),vin(n)) (M= [I I] matrix, n= 2∗m)
- void add_mul_Mtv_idty_idty (double vout[], const double vin[])

  Add matrixTranspose-vector product vout += [I I]'∗vin (vout(n),vin(m) (M= [I I] matrix, n= 2∗m)
- void mul_Mv_diag_diag (double vout[], const double vin[])

  Matrix-vector product vout= [D1 D2]∗vin (vout(m),vin(n)) (M= [D1 D2] matrix, n= 2∗m)
- void mul_Mtv_diag_diag (double vout[], const double vin[])

  MatrixTranspose-vector product vout= [D1 D2]'∗vin (vout(n),vin(m) (M= [D1 D2] matrix, n= 2∗m)
- void add_mul_Mv_diag_diag (double vout[], const double vin[])

  Add matrix-vector product vout += [D1 D2]∗vin (vout(m),vin(n)) (M= [D1 D2] matrix, n= 2∗m)
- void add_mul_Mtv_diag_diag (double vout[], const double vin[])

Add matrixTranspose-vector product vout += [D1 D2]'∗vin (vout(n),vin(m) (M= [D1 D2] matrix, n= 2∗m)

- void mul_Mv_gen_sym_uptr (double vout[], const double vin[])

  Matrix-vector product vout=M∗vin (vout(m),vin(n)) (general symmetric upper triangular rowwise matrix)
- void add_mul_Mv_gen_sym_uptr (double vout[], const double vin[])

  Add matrix-vector product vout += M∗vin (vout(m),vin(n)) (general symmetric upper triangular rowwise matrix)
- void order_packed_format (int begsize, int nz, int ∗&beg, int ∗&ind, double ∗&val)

  Order a matrix packed format (both column-wise and row-wise)

5.4.1 Detailed Description

Class for manipulating matrices, and interfacing SparseChol.

5.4.2 Constructor & Destructor Documentation

5.4.2.1 MatrixBlockIP::MatrixBlockIP (int *blocks* = 1)

Constructor.

Parameters

blocks	number of blocks where this matrix will appear;

Note

if blocks unknown at construction time, do not pass this parameter to the constructor; it will be updated later, before symbolic factorization is performed, by the minimization algorithm

5.4.2.2 MatrixBlockIP::MatrixBlockIP (MatrixBlockIP ∗ *mbip*)

Copy constructor, does not copy SparseChol.

Parameters

mbip	MatrixBlockIP to copy

5.4.2.3 MatrixBlockIP::∼MatrixBlockIP ( )

Destructor.

5.4.3 Member Function Documentation

5.4.3.1 void MatrixBlockIP::add_mul_Mtv (double *vout[]*, const double *vin[]*)

Add matrixTranspose-vector product vout += M(t)∗vin (vout(n),vin(m) (driver)

Parameters

vout	Result of the product
vin	Vector to product

5.4.3.2 void MatrixBlockIP::add_mul_Mtv_column_wise (double *vout[],* const double *vin[]*) [inline],[private]

Add matrixTranspose-vector product vout += M(t)∗vin (vout(n),vin(m) (general columnwise matrix)

Parameters

vout	Result of the product
vin	Vector to product

5.4.3.3 void MatrixBlockIP::add_mul_Mtv_diag_diag (double *vout[],* const double *vin[]*) [inline],[private]

Add matrixTranspose-vector product vout += [D1 D2]'∗vin (vout(n),vin(m) (M= [D1 D2] matrix, n= 2∗m)

Parameters

vout	Result of the product
vin	Vector to product

5.4.3.4 void MatrixBlockIP::add_mul_Mtv_idty_idty (double *vout[],* const double *vin[]*) [inline],[private]

Add matrixTranspose-vector product vout += [I I]'∗vin (vout(n),vin(m) (M= [I I] matrix, n= 2∗m)

Parameters

vout	Result of the product
vin	Vector to product

5.4.3.5 void MatrixBlockIP::add_mul_Mtv_network (double *vout[],* const double *vin[]*) [inline],[private]

Add matrixTranspose-vector product vout += M(t)∗vin (vout(n),vin(m) (network matrix, either oriented or nonoriented)

Note

It assumes vin has dimension m+1; last equation, related to last node, is redundant and not used, but it is needed for an efficient computation.
Element vin[m] is used (but backed-up at beginning and restored at ending)

Parameters

vout	Result of the product
vin	Vector to product

5.4.3.6 void MatrixBlockIP::add_mul_Mtv_row_wise (double *vout[],* const double *vin[]*) [inline],[private]

Add matrixTranspose-vector product vout += M(t)∗vin (vout(n),vin(m) (general rowwise matrix)

Parameters

vout	Result of the product
vin	Vector to product

5.4.3.7 void MatrixBlockIP::add_mul_Mv (double *vout[],* const double *vin[]*)

Add matrix-vector product vout += M∗vin (vout(m),vin(n)) (driver)

Parameters

vout	Result of the product
vin	Vector to product

5.4.3.8 void MatrixBlockIP::add_mul_Mv_column_wise (double *vout[],* const double *vin[]*) [inline], [private]

Add matrix-vector product vout += M∗vin (vout(m),vin(n)) (general columnwise matrix)

Parameters

vout	Result of the product
vin	Vector to product

5.4.3.9 void MatrixBlockIP::add_mul_Mv_diag_diag (double *vout[],* const double *vin[]*) [inline], [private]

Add matrix-vector product vout += [D1 D2]∗vin (vout(m),vin(n)) (M= [D1 D2] matrix, n= 2∗m)

Parameters

vout	Result of the product
vin	Vector to product

5.4.3.10 void MatrixBlockIP::add_mul_Mv_diagonal (double *vout[],* const double *vin[]*) [inline], [private]

Add Diagonal-vector product vout += D∗vin (vout(m),vin(n)) D is diagonal, m=n.

Parameters

vout	Result of the product
vin	Vector to product

5.4.3.11 void MatrixBlockIP::add_mul_Mv_gen_sym_uptr (double *vout[],* const double *vin[]*) [inline], [private]

Add matrix-vector product vout += M∗vin (vout(m),vin(n)) (general symmetric upper triangular rowwise matrix)

Parameters

vout	Result of the product. Must be allocated
vin	Vector to product

5.4.3.12 void MatrixBlockIP::add_mul_Mv_identity (double *vout[],* const double *vin[]*) [inline], [private]

Add Identity-vector product vout += M∗vin (vout(m),vin(n))

Identity matrix, m=n, just add input to output vector

Parameters

vout	Result of the product
vin	Vector to product

5.4.3.13 void MatrixBlockIP::add_mul_Mv_idty_idty (double *vout[],* const double *vin[]*) [inline], [private]

Add matrix-vector product vout += [I I]∗vin (vout(m),vin(n)) (M= [I I] matrix, n= 2∗m)

Parameters

vout	Result of the product
vin	Vector to product

5.4.3.14 void MatrixBlockIP::add_mul_Mv_network (double *vout[],* const double *vin[]*) [inline], [private]

Add matrix-vector product vout += M∗vin (vout(m),vin(n)) (network matrix, either oriented or nonoriented)

Note

It assumes vout has dimension m+1; last equation, related to last node, is redundant and not used, but it is needed for an efficient computation.
Element vout[m] is used (but backed-up at beginning and restored at ending)

Parameters

vout	Result of the product
vin	Vector to product

5.4.3.15 void MatrixBlockIP::add_mul_Mv_row_wise (double *vout[],* const double *vin[]*) [inline], [private]

Add matrix-vector product vout += M∗vin (vout(m),vin(n)) (general rowwise matrix)

Parameters

vout	Result of the product
vin	Vector to product

5.4.3.16 void MatrixBlockIP::add_new_column (int *size,* int *irows[],* double *values[]*)

Add a new column into the matrix.

Parameters

size	Number of non-zero elements of the new column
irows	Row position for each element, have to be ordered
values	Value of non-zero elements

5.4.3.17 void MatrixBlockIP::add_new_columns (int *num_columns,* int *size[],* int ∗ *irows[],* double ∗ *values[]*)

Add new columns into the matrix.

Parameters

num_columns	Number of columns to add
size	number of non-zero elements of the new column for each column
irows	Row position for each element and column, have to be ordered
values	Value of non-zero elements for each column

5.4.3.18 void MatrixBlockIP::analyze_D (int *numBlocks,* bool *sameL,* MatrixBlockIP *L[]*) [inline]

Create internal structure arrays to compute D = Theta_(numBlocks+1) + sum{i..numBlocks} L_i∗Theta_i∗L_i' It decides whether D is DIAGONAL or GEN_SYM_UPTR

Parameters

numBlocks	Number of diagonal blocks
sameL	Define if the same matrix is used for each L block
L	Linking constraints blocks

5.4.3.19 void MatrixBlockIP::analyze_D_diagonal (int *numBlocks,* bool *sameL,* MatrixBlockIP *L[]*)

Create internal structure arrays to compute D = Theta_(numBlocks+1) + sum{1..numBlocks} L_i∗Theta_i∗L_i' when D is a diagonal

Parameters

numBlocks	Number of diagonal blocks
sameL	Define if the same matrix is used for each L block
L	Linking constraints blocks

Note

valD (of analyze_D_gen_sym_uptr()) is reused, then it is allocated with ALLOC (see analyze_D_gen_sym_uptr for an explanation)

5.4.3.20 void MatrixBlockIP::analyze_D_gen_sym_uptr (int *numBlocks,* bool *sameL,* MatrixBlockIP *L[]*)

Create internal structure arrays to compute D = Theta_(numBlocks+1) + sum{i..numBlocks} L_i∗Theta_i∗L_i' when D is a general symmetric upper triangular matrix

Parameters

numBlocks	Number of diagonal blocks
sameL	Define if the same matrix is used for each L block
L	Linking constraints blocks

Note

icolLLt, inivalD, blockD, valD and icolD are allocated with ALLOC for performance purposes, so they have to be freed with FREE

5.4.3.21 void MatrixBlockIP::change_columns_sign (int *size,* int *columns[]*)

Changes the sign of some columns.

Parameters

size	Number of columns to change the sign
columns	Index to the columns to change the sign

5.4.3.22 void MatrixBlockIP::change_rows_sign (int *size*, int *rows[]*)

Change the sign of some rows.

Parameters

size	Number of rows to change the sign
rows	Index to the rows to change the sign

5.4.3.23 void MatrixBlockIP::column_wise_to_row_wise_format ( )

Convert a matrix in column-wise format to row-wise format.

5.4.3.24 void MatrixBlockIP::compute_D (int *numBlocks*, bool *sameL*, MatrixBlockIP *L[]*, bool *isActive[]*, int *iniTheta[]*, double *Theta[]*) [inline]

Compute D = Theta_(numBlocks+1) + sum{i..numBlocks} L_i∗Theta_i∗L_i'.

Parameters

numBlocks	Number of diagonal blocks
sameL	Define if the same matrix is used for each L block
isActive	Says what linking constrains are active
iniTheta	Indice to the first element of the Theta matrix of each block
Theta	numBlocks diagonal matrices with dimension n_i x n_i

5.4.3.25 void MatrixBlockIP::compute_D_diagonal (int *numBlocks*, bool *sameL*, MatrixBlockIP *L[]*, bool *isActive[]*, int *iniTheta[]*, double *Theta[]*)

Compute D = Theta_(numBlocks+1) + sum{i in 1..numBlocks} L_i∗Theta_i∗L_i' when D is diagonal.

Parameters

numBlocks	Number of diagonal blocks
sameL	Define if the same matrix is used for each L block
isActive	Says what linking constrains are active
iniTheta	Indice to the first element of the Theta matrix of each block
Theta	numBlocks diagonal matrices with dimension n_i x n_i

5.4.3.26 void MatrixBlockIP::compute_D_gen_sym_uptr (int *numBlocks*, bool *sameL*, MatrixBlockIP *L[]*, bool *isActive[]*, int *iniTheta[]*, double *Theta[]*)

Compute D = Theta_(numBlocks+1) + sum{i..numBlocks} L_i∗Theta_i∗L_i' when D is a symmetric general matrix.

Parameters

numBlocks	Number of diagonal blocks
sameL	Define if the same matrix is used for each L block

isActive	Says what linking constrains are active
iniTheta	Indice to the first element of the Theta matrix of each block
Theta	numBlocks diagonal matrices with dimension n_i x n_i

5.4.3.27 void MatrixBlockIP::compute_full_matrix (int *numBlocks*, bool *sameN*, MatrixBlockIP *N[]*, bool *sameL*, MatrixBlockIP *L[]*, int *numActiveLnk* = 0, int * *listActiveLnk* = NULL)

Builds a packed rowwise format structured matrix based on diagonal blocks and linking constraints blocks.

Parameters

numBlocks	Number of diagonal blocks
sameN	Define if the same matrix is used for each N block
N	Diagonal blocks
sameL	Define if the same matrix is used for each L block
L	Linking constraints blocks
numActiveLnk	Number of active lnk
listActiveLnk	List of active linking, j=listActiveLnk[i], i=1..numActiveLnk, j in {1,..,l_link}) If listActiveLnk is NULL all L rows are used

5.4.3.28 void MatrixBlockIP::copy (MatrixBlockIP * *mbip*)

Copy, does not copy SparseChol.

Parameters

mbip	MatrixBlockIP to copy

5.4.3.29 void MatrixBlockIP::create_diag_diag_matrix (int *nrows*, double *& *d1*, double *& *d2*)

Native method to create a diagonal-diagonal matrix D= [D1 D2].

where D1= diag(d1), D2= diag(d2) (dimension: nrows x (2nrows))

Parameters

nrows	Number of rows
d1	Values of the diagonal D1
d2	Values of the diagonal D2

5.4.3.30 void MatrixBlockIP::create_diagonal_matrix (int *dim*, double *& *d*)

Native method to create a diagonal matrix D= diag(d) of dimension dim.

Parameters

dim	Dimension of the matrix
d	Values of the diagonal elements

5.4.3.31 void MatrixBlockIP::create_general_matrix_column_wise (int *m,* int *n,* int *nz,* int *&* *inicola,* int *&* *irowa,* double *&* *a*)

Native method to create a general column-wise packed matrix.

Parameters

m	Number of rows
n	Number of columns
nz	Number of non-zero elements
inicola	Index to the first element of each column
irowa	Row position for each element
a	Value of non-zero elements

Note

If icola is not ordered order_matrix function must be called when the arrays are filled

5.4.3.32 void MatrixBlockIP::create_general_matrix_format_ija (int *m,* int *n,* int *nz,* int *&* *row_index,* int *&* *col_index,* double *&* *values*)

Creates a general matrix in format ija.

Parameters

m	Number of rows
n	Number of columns
nz	Number of non-zero elements
row_index	Row position for each element
col_index	Column position for each element
values	Value of non-zero elements

5.4.3.33 void MatrixBlockIP::create_general_matrix_row_wise (int *m,* int *n,* int *nz,* int *&* *inirowa,* int *&* *icola,* double *&* *a*)

Native method to create a general row-wise packed matrix.

Parameters

m	Number of rows
n	Number of columns
nz	Number of non-zero elements
inirowa	Index to the first element of each row
icola	Column position for each element
a	Value of non-zero elements

Note

If icola is not ordered order_matrix function must be called when the arrays are filled

5.4.3.34 void MatrixBlockIP::create_identity_matrix (int *dim*)

Native method to create a identity matrix of dimension dim.

Parameters

dim	Dimension of the matrix

5.4.3.35 void MatrixBlockIP::create_idty_idty_matrix (int *nrows*)

Native method to create a identity-identity matrix with two identities [I I].

Dimension: nrows x (2nrows)

Parameters

nrows	Number of rows

5.4.3.36 void MatrixBlockIP::create_network_matrix (int *num_arcs,* int *num_nodes,* int *&* *src,* int *&* *dst,* bool *oriented =* `true`)

Native method to create a network matrix.

Parameters

num_arcs	Number of arcs
num_nodes	Number of nodes
src	Source of each arc. Dimension num_arcs
dst	Destination of each arc. Dimension num_arcs
oriented	For oriented/nonoriented networks

5.4.3.37 void MatrixBlockIP::delete_rows (int *size,* int *rows[]*)

Delete some rows.

Parameters

size	Number of rows to be deleted
rows	Index to the rows to be deleted, must be ordered

5.4.3.38 void MatrixBlockIP::diag_diag_to_general ( ) [inline]

Calculates and stores the matrix [D1 D2] in packed rowwise format.

5.4.3.39 void MatrixBlockIP::diag_diag_to_ija_format ( )

Calculates and stores the matrix [D1 D2] in ija format.

5.4.3.40 void MatrixBlockIP::diagonal_to_general ( ) [inline]

Calculates and stores the diagonal D matrix in packed rowwise format.

5.4.3.41 void MatrixBlockIP::diagonal_to_ija_format ( )

Calculates and stores the diagonal D1 matrix in ija format.

5.4.3.42 void MatrixBlockIP::exist_var_in_row (int *column*, bool *appear[]*, double *coefs[]*)

Given a column determines if exists an element for each row of the matrix, in case of exist returns the value.

Parameters

column	The column to examine
appear	The array that says if a element exists or not for each row. The user must allocate the space before call the function
coefs	The array that have the value of the element if exists, if not the content in that position is indeterminate. The user must allocate the space before call the function

5.4.3.43 void MatrixBlockIP::free_memory ( ) [private]

Free all the memory allocated by the application - not the user.

5.4.3.44 int MatrixBlockIP::get_column (int *column*, int *&* *irows*, double *&* *values*, bool *invert_sign* = false)

Gets a column of the matrix.

Parameters

column	Index to the column
irows	Row index for each ealement. Must be freed with delete[]
values	Value for each non-zero element. Must be freed with delete[]
invert_sign	If true change the sign of each element in the column

Returns

Number of non-zero elements in the column

5.4.3.45 int MatrixBlockIP::get_ipfa (int *i*) [inline]

Interface routine to SparseChol get_ipfa().

Returns

ipfa[i]

Note

The user must guarantee ipfa previously computed in call to symbolic_fact().

5.4.3.46 int MatrixBlockIP::get_maxfillin ( ) [inline]

Interface routine to SparseChol get_maxfillin().

Returns

maxfillin

Note

The user must guarantee maxfillin previously computed in call to symbolic_fact().

5.4.3.47 int MatrixBlockIP::get_maxlnz ( ) [inline]

Interface routine to SparseChol get_maxlnz().

Returns

maxlnz

Note

The user must guarantee maxlnz previously computed in call to symbolic_fact().

5.4.3.48 int MatrixBlockIP::get_njka ( ) [inline]

Interface routine to SparseChol get_njka().

Returns

njka

Note

The user must guarantee njka previously computed in call to symbolic_fact().

5.4.3.49 int MatrixBlockIP::get_num_semidef_matrix ( ) [inline]

Interface routine to SparseChol get_semidef_matrix().

Returns

num_semidef_matrix

Note

It will 0 if no numeric factorization made.

5.4.3.50 int MatrixBlockIP::get_num_zero_pivots ( ) [inline]

Interface routine to SparseChol get_num_zero_pivots().

Returns

num_zero_pivots

Note

It will 0 if no numeric factorization made.

5.4.3.51 int MatrixBlockIP::get_pfa (int *i*) [inline]

Interface routine to SparseChol get_pfa().

Returns

pfa[i]

Note

The user must guarantee pfa previously computed in call to symbolic_fact().

5.4.3.52 void MatrixBlockIP::identity_to_general ( ) [inline]

Calculates and stores the I matrix in packed rowwise format.

5.4.3.53 void MatrixBlockIP::identity_to_ija_format ( )

Calculates and stores the identity I matrix in ija format.

5.4.3.54 void MatrixBlockIP::idty_idty_to_general ( ) [inline]

Calculates and stores the matrix [I I] in packed rowwise format.

5.4.3.55 void MatrixBlockIP::idty_idty_to_ija_format ( )

Calculates and stores the matrix [I I] in ija format.

5.4.3.56 void MatrixBlockIP::ija_to_rowwise ( )

Calculates and stores the matrix in packed rowwise format.

Note

ija format is loaded

5.4.3.57 void MatrixBlockIP::mul_Mtv (double *vout[],* const double *vin[]*)

MatrixTranspose-vector product vout=M(t)∗vin (vout(n),vin(m) (driver)

Parameters

vout	Result of the product. Must be allocated
vin	Vector to product

5.4.3.58 void MatrixBlockIP::mul_Mtv_column_wise (double *vout[],* const double *vin[]*) [inline], [private]

MatrixTranspose-vector product vout=M(t)∗vin (vout(n),vin(m) (general columnwise matrix)

Parameters

vout	Result of the product. Must be allocated
vin	Vector to product

5.4.3.59 void MatrixBlockIP::mul_Mtv_diag_diag (double *vout[],* const double *vin[]*) [inline], [private]

MatrixTranspose-vector product vout= [D1 D2]'∗vin (vout(n),vin(m) (M= [D1 D2] matrix, n= 2∗m)

Parameters

vout	Result of the product. Must be allocated
vin	Vector to product

5.4.3.60 void MatrixBlockIP::mul_Mtv_idty_idty (double *vout[],* const double *vin[]*) [inline],[private]

MatrixTranspose-vector product vout= [I I]'∗vin (vout(n),vin(m) (M= [I I] matrix, n= 2∗m)

Parameters

vout	Result of the product. Must be allocated
vin	Vector to product

5.4.3.61 void MatrixBlockIP::mul_Mtv_network (double *vout[],* const double *vin[]*) [inline],[private]

MatrixTranspose-vector product vout=M(t)∗vin (vout(n),vin(m) (network matrix, either oriented or nonoriented)

Note

It assumes vin has dimension m+1; last equation, related to last node, is redundant and not used, but it is needed for an efficient computation.
Element vin[m] is used (but backed-up at beginning and restored at ending)

Parameters

vout	Result of the product. Must be allocated
vin	Vector to product

5.4.3.62 void MatrixBlockIP::mul_Mtv_row_wise (double *vout[],* const double *vin[]*) [inline],[private]

MatrixTranspose-vector product vout=M(t)∗vin (vout(n),vin(m) (general rowwise matrix)

Parameters

vout	Result of the product. Must be allocated
vin	Vector to product

5.4.3.63 void MatrixBlockIP::mul_Mv (double *vout[],* const double *vin[]*)

Matrix-vector product vout=M∗vin (vout(m),vin(n)) (driver)

Parameters

vout	Result of the product. Must be allocated
vin	Vector to product

5.4.3.64 void MatrixBlockIP::mul_Mv_column_wise (double *vout[],* const double *vin[]*) [inline],[private]

Matrix-vector product vout=M∗vin (vout(m),vin(n)) (general columnwise matrix)

Parameters

vout	Result of the product. Must be allocated
vin	Vector to product

5.4.3.65 void MatrixBlockIP::mul_Mv_diag_diag (double *vout[],* const double *vin[]*) [inline], [private]

Matrix-vector product vout= [D1 D2]∗vin (vout(m),vin(n)) (M= [D1 D2] matrix, n= 2∗m)

Parameters

vout	Result of the product. Must be allocated
vin	Vector to product

5.4.3.66 void MatrixBlockIP::mul_Mv_diagonal (double *vout[],* const double *vin[]*) [inline], [private]

Diagonal-vector product vout=D∗vin (vout(m),vin(n)) D is diagonal, m=n.

Parameters

vout	Result of the product. Must be allocated
vin	Vector to product

5.4.3.67 void MatrixBlockIP::mul_Mv_gen_sym_uptr (double *vout[],* const double *vin[]*) [inline], [private]

Matrix-vector product vout=M∗vin (vout(m),vin(n)) (general symmetric upper triangular rowwise matrix)

Parameters

vout	Result of the product. Must be allocated
vin	Vector to product

5.4.3.68 void MatrixBlockIP::mul_Mv_identity (double *vout[],* const double *vin[]*) [inline], [private]

Identity-vector product vout=M∗vin (vout(m),vin(n))

Identity matrix, m=n, just copy input to output vector

Parameters

vout	Result of the product. Must be allocated
vin	Vector to product

5.4.3.69 void MatrixBlockIP::mul_Mv_idty_idty (double *vout[],* const double *vin[]*) [inline], [private]

Matrix-vector product vout= [I I]∗vin (vout(m),vin(n)) (M= [I I] matrix, n= 2∗m)

Parameters

vout	Result of the product. Must be allocated
vin	Vector to product

5.4.3.70 void MatrixBlockIP::mul_Mv_network (double *vout[],* const double *vin[]*) [inline], [private]

Matrix-vector product vout=M∗vin (vout(m),vin(n)) (network matrix, either oriented or nonoriented)

Note

It assumes vout has dimension m+1; last equation, related to last node, is redundant and not used, but it is needed for an efficient computation.
Element vout[m] is used (but backed-up at beginning and restored at ending)

Parameters

vout	Result of the product. Must be allocated
vin	Vector to product

5.4.3.71 void MatrixBlockIP::mul_Mv_row_wise (double *vout[],* const double *vin[]*) [inline], [private]

Matrix-vector product vout=M∗vin (vout(m),vin(n)) (general rowwise matrix)

Parameters

vout	Result of the product. Must be allocated
vin	Vector to product

5.4.3.72 void MatrixBlockIP::native_to_general (bool *delete_native_format* = false) [inline]

Calculates and stores the matrix in packed rowwise format.

Parameters

delete_native_format	If true only packed rowwise format will be stored

5.4.3.73 void MatrixBlockIP::network_to_general ( )

Calculates and stores the network matrix (either oriented or nonoriented) in packed rowwise format.

5.4.3.74 void MatrixBlockIP::network_to_ija_format ( )

Calculates and stores the network matrix in ija format. It considers both oriented and nonoriented cases.

5.4.3.75 void MatrixBlockIP::numeric_fact_M ( ) [inline]

Interface routine to SparseChol numeric factorization.

5.4.3.76 void MatrixBlockIP::numeric_fact_MMt (double ∗ *Theta,* int *i_k* = 0) [inline]

Interface routine to SparseChol numeric factorization.

5.4.3.77 void MatrixBlockIP::numeric_solve_M (double ∗ *rhs,* WHO_PERMUTES *whoperm* = CHOLESKY) [inline]

Interface routine to SparseChol numeric solve.

5.4.3.78 void MatrixBlockIP::numeric_solve_MMt (double * *rhs*, int *i_k* = 0, WHO_PERMUTES *whoperm* = CHOLESKY) [inline]

Interface routine to SparseChol numeric solve.

5.4.3.79 void MatrixBlockIP::order_matrix ( )

Order the matrix when has been created in row-wise or column-wise format.

5.4.3.80 void MatrixBlockIP::order_packed_format (int *begsize*, int *nz*, int *&* *beg*, int *&* *ind*, double *&* *val*) [private]

Order a matrix packed format (both column-wise and row-wise)

Parameters

begsize	Number of rows (if row-wise) or columns (if column-wise) + 1
nz	Number of nonzeros in the matrix.
beg	Rows (if row-wise) or columns (if column-wise) starts of the matrix.
ind	Columns (if row-wise) or rows (if column-wise) indices of nonzeros entries.
val	Values of the nonzero entries.

Note

Parameters beg, ind and val assumes to be of appropriate dimensions before the method is called, namely beg[begsize], ind[nz], val[nz]

5.4.3.81 void MatrixBlockIP::print_matrix (bool *print_ija* = true, bool *start_one* = true)

Print the matrix.

Parameters

print_ija	If print_ija is true then write triple (i,j,a) whenever possible (if matrix is in ija or packed rowwise format); if print_ija=false then writes (inirow, j, a) whenever possible (if matrix is in packed rowwise format).
start_one	Start vectors at position 1 (true) or zero (false)

5.4.3.82 void MatrixBlockIP::print_matrix (ofstream & *outfile*, bool *print_ija* = true, bool *start_one* = true)

Write the matrix into a file.

Parameters

outfile	Output file stream where the matrix will be printed

5.4.3.83 void MatrixBlockIP::print_vector (double * *v*, int *size*, string *name*)

Print some positions and name of a vector.

Parameters

v	Vector to print
size	Number of positions to print
name	Vector name

5.4.3.84 void MatrixBlockIP::reset ( )

Comes back to the initial state.

5.4.3.85 void MatrixBlockIP::restore ( )

Comes back to the state after create the matrix.

5.4.3.86 void MatrixBlockIP::row_wise_to_column_wise_format ( )

Convert a matrix in column-wise format to row-wise format.

5.4.3.87 void MatrixBlockIP::symbolic_fact_M (CHOL_SOLVER *chslv* = SPRSBLKLLT, int * *prov_pfa* = NULL, int * *prov_ipfa* = NULL, NUMBERING *prov_pfa_numbering* = NOT_COMPUTED) [inline]

Interface routine to SparseChol symbolic factorization.

5.4.3.88 void MatrixBlockIP::symbolic_fact_MMt (CHOL_SOLVER *chslv* = SPRSBLKLLT, int * *prov_pfa* = NULL, int * *prov_ipfa* = NULL, NUMBERING *prov_pfa_numbering* = NOT_COMPUTED) [inline]

Interface routine to SparseChol symbolic factorization.

5.4.4 Member Data Documentation

5.4.4.1 double* MatrixBlockIP::a

Value of non-zero elements.

5.4.4.2 int* MatrixBlockIP::blockD

5.4.4.3 SparseChol MatrixBlockIP::chol

Sparse Cholesky class.

For sparse Cholesky

5.4.4.4 bool MatrixBlockIP::columnwise

True if the matrix is stored in column-wise packed format, incompatible with rowwise.

5.4.4.5 double* MatrixBlockIP::d1

d1 for DIAGONAL and 1st submatrix of DIAG_DIAG

Diagonal format attributes

5.4.4.6 double* MatrixBlockIP::d2

d2 for 2nd submatrix of DIAG_DIAG

5.4.4.7 int* MatrixBlockIP::dst

Destination of each arc.

5.4.4.8 int* MatrixBlockIP::icola

Column position for each element.

5.4.4.9 int* MatrixBlockIP::icolD

5.4.4.10 int* MatrixBlockIP::icolLLt

5.4.4.11 bool MatrixBlockIP::ija

True if irowa, icola and a are stored.

5.4.4.12 int* MatrixBlockIP::inicola

Index to the first element of each column.

General column-wise packed format attributes

5.4.4.13 int* MatrixBlockIP::inirowa

Index to the first element of each row.

General row-wise packed format attributes

5.4.4.14 int* MatrixBlockIP::inirowLLt

5.4.4.15 int* MatrixBlockIP::inivalD

5.4.4.16 int* MatrixBlockIP::irowa

Row position for each element.

5.4.4.17 int MatrixBlockIP::m

Number of rows.

5.4.4.18 bool MatrixBlockIP::made_analyze_D

boolean to check whether analyze_D already made, needed to compute_D()

5.4.4.19 bool MatrixBlockIP::made_symbfct_M

5.4.4.20 bool MatrixBlockIP::made_symbfct_MMt

boolean to check whether symbolic factorization already made

5.4.4.21 int MatrixBlockIP::n

Number of columns.

5.4.4.22 int MatrixBlockIP::num_arcs

Number of arcs.

5.4.4.23 int MatrixBlockIP::num_blocks

number of replications of this matrix; this is needed to store space for num_blocks numerical factorization of M∗-Theta[i]∗M', for different Theta[i], i=0,...,num_blocks-1 This value is set by the minimization algorithm, which needs the factorizations.

5.4.4.24 int MatrixBlockIP::num_nodes

Number of nodes.

5.4.4.25 int MatrixBlockIP::nz

Number of non-zero elements.

Common packed format attributes

5.4.4.26 bool MatrixBlockIP::rowwise

True if inirowa, icola and a are stored.

5.4.4.27 int MatrixBlockIP::sizeL

For D Matrix

5.4.4.28 int∗ MatrixBlockIP::src

Source of each arc.

5.4.4.29 TYPE_MATRIX MatrixBlockIP::type

Type of matrix.

Common in all types matrix

5.4.4.30 TYPE_ORIENTATION MatrixBlockIP::type_orientation

Type of orientation, by default, oriented.

Network format attributes

5.4.4.31 double∗ MatrixBlockIP::valD

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

- /home/jcastro2/intpoint/BlockIP/BlockIP/src_blockip-v2/MatrixBlockIP.h
- /home/jcastro2/intpoint/BlockIP/BlockIP/src_blockip-v2/MatrixBlockIP.C

5.5 MatrixBlockIP::Order_ija Struct Reference

Auxiliary struct for sorting matrices in ija format.

Public Member Functions

- bool operator() (int i, int j)

  Comparison function.

Public Attributes

- int * row

  Pointer to rows of matrix elements.
- int * col

  Pointer to columns of matrix elements.

5.5.1 Detailed Description

Auxiliary struct for sorting matrices in ija format.

5.5.2 Member Function Documentation

5.5.2.1 bool MatrixBlockIP::Order_ija::operator() (int *i*, int *j*) [inline]

Comparison function.

5.5.3 Member Data Documentation

5.5.3.1 int* MatrixBlockIP::Order_ija::col

Pointer to columns of matrix elements.

5.5.3.2 int* MatrixBlockIP::Order_ija::row

Pointer to rows of matrix elements.

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

- /home/jcastro2/intpoint/BlockIP/BlockIP/src_blockip-v2/MatrixBlockIP.h

5.6 MatrixBlockIP::Order_vector Struct Reference

Auxiliary struct for sorting vectors.

Public Member Functions

- bool operator() (int i, int j)

  Comparison function.

Public Attributes

- int * v

  Pointer to vector elements.

5.6.1 Detailed Description

Auxiliary struct for sorting vectors.

5.6.2 Member Function Documentation

5.6.2.1 bool MatrixBlockIP::Order_vector::operator() (int *i*, int *j*) [inline]

Comparison function.

5.6.3 Member Data Documentation

5.6.3.1 int* MatrixBlockIP::Order_vector::v

Pointer to vector elements.

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

- /home/jcastro2/intpoint/BlockIP/BlockIP/src_blockip-v2/MatrixBlockIP.h

5.7 SparseChol Class Reference

Class for sparse Cholesky factorizations.

`#include <SparseChol.h>`

Classes

- struct SRC_DST_ARC

  Auxiliary struct for sorting network structure.

Public Member Functions

- SparseChol ()

  Constructor. It calls initialize().
- ~SparseChol ()

  Destructor. It calls free_mem().
- void reset (CHOL_SOLVER chol_solver=(CHOL_SOLVER) NULL, TYPE_MATRIX type_matrix=(TYPE_MATRIX) NULL, TYPE_ORIENTATION type_orientation=ORIENTED)

  Like calling destructor+constructor.

- void initialize (CHOL_SOLVER chol_solver=(CHOL_SOLVER) NULL, TYPE_MATRIX type_matrix=(TYPE_MATRIX) NULL, TYPE_ORIENTATION type_orientation=ORIENTED)

  Initialize attributes (set variables to 0, NULL...)
- void symbolic_fact_MMt (int m, int n, int nz, int *icola, int *inirowa, double *a, int k=1, int *prov_pfa=NULL, int *prov_ipfa=NULL, NUMBERING prov_pfa_numbering=NOT_COMPUTED)

  Computes symmetric ordering and symbolic factorization of AA' for a general matrix A.
- void symbolic_fact_MMt (int nnu, int nar, int *src, int *dst, int k=1, int *prov_pfa=NULL, int *prov_ipfa=NULL, NUMBERING prov_pfa_numbering_=NOT_COMPUTED)

  Computes symmetric ordering and symbolic factorization of AA' for a network matrix A.
- void symbolic_fact_MMt (int m, int k=1)

  Computes symbolic factorization of AA' for an IDENTITY or IDTY_IDTY matrix A.
- void symbolic_fact_MMt (int m, double *d1_in, int k=1)

  Computes symbolic factorization of AA' for a DIAG matrix A.
- void symbolic_fact_MMt (int m, double *d1_in, double *d2_in, int k=1)

  Computes symbolic factorization of AA' for a DIAG_DIAG matrix A.
- void numeric_fact_MMt (double *Theta, int i_k=0)

  *Computes numerical factorization of A*Theta*A'.*
- void numeric_solve_MMt (double *rhs, int i_k=0, WHO_PERMUTES whoperm=(WHO_PERMUTES) NULL)

  *Solve systems with matrix A*Theta*A' and right-hand-side rhs.*
- void symbolic_fact_M (int m, int nz, int *icola, int *inirowa, int *prov_pfa=NULL, int *prov_ipfa=NULL, NUMBERING prov_pfa_numbering=NOT_COMPUTED)

  Computes symmetric ordering and symbolic factorization of A for a general symmetric matrix.
- void symbolic_fact_M (int m)

  Computes symbolic factorization of DIAGONAL matrix A.
- void numeric_fact_M (double *a)

  Computes numerical factorization for a positive semidefinite matrix A.
- void numeric_solve_M (double *rhs, WHO_PERMUTES whoperm=(WHO_PERMUTES) NULL)

  Solve systems with a positive semidefinite symmetric matrix A and right-hand-side rhs.
- int get_pfa (int i)

  Provides pfa[i] with no check.
- int get_ipfa (int i)

  Provides ipfa[i] with no check.
- int get_maxlnz ()

  Provides maxlnz with no check.
- int get_maxfillin ()

  Provides maxfillin with no check.
- int get_njka ()

  Provides njka with no check.
- int get_num_zero_pivots ()

  Provides num_zero_pivots with no check.
- int get_num_semidef_matrix ()

  Provides num_semidef_matrix with no check.

Private Member Functions

- void free_mem ()

  Frees memory (uses C free(), not C++ delete[])
- void symbolic_fact_MMt_sprsblkllt_general (int n, int nz, int *icola, int *inirowa, double *a)

  Computes symmetric ordering and symbolic factorization for AA' using Ng-Peyton package for a general matrix.
- void symbolic_fact_MMt_sprsblkllt_network (int *src, int *dst)

Computes symmetric ordering and symbolic factorization for AA' using Ng-Peyton package for a network matrix.

- void get_indices_a_general (int *icola, int *inirowa, double *a)

  Computes data structure of arrays ia, ja, ka, la, va as detailed in Monma and Morton paper.
- void get_ipk_ipl_network (int *src, int *dst)

  Computes data structures with network sorted by src and dst, to be used later.
- void get_pfa_ipfa_network ()

  Computes row permutation of AA' for a network matrix A.
- void get_pfa_ipfa_general (int *inp_ia, int *inp_ja)

  Computes row permutation of AA'/A for a general matrix.
- void symbolic_AThetaAt_A ()

  Symbolic factorization of AThetaA' or symmetric A.
- void get_ilnz_network ()

  Computes variables and indices for later fast filling of lnz for a network matrix.
- void get_ilnz_ifillin_general (int *inp_ia, int *inp_ja)

  Compute arrays of indices to lnz(): array ilnz(njka), for a general matrix.
- void numeric_fact_MMt_sprsblkllt_general (double *Theta, int i_k)

  *Computes numerical factorization of A*Theta*A' when A is general matrix.*
- void numeric_fact_MMt_sprsblkllt_network (double *Theta, int i_k)

  *Computes numerical factorization of A*Theta*A' when A is a network.*
- void numeric_fact_MMt_identity (double *Theta, int i_k)

  *Numerical factorization of I*Theta*I'.*
- void numeric_fact_MMt_idty_idty (double *Theta, int i_k)

  Numerical factorization of [I I](Theta^+, Theta^-)*[I I]' matrix.*
- void numeric_fact_MMt_diagonal (double *Theta, int i_k)

  *Numerical factorization of D*Theta*D' (diagonal matrix)*
- void numeric_fact_MMt_diag_diag (double *Theta, int i_k)

  Numerical factorization of [D1 D2](Theta^+, Theta^-)*[D1 D2]' (DIAG_DIAG matrix)*
- void numeric_solve_MMt_sprsblkllt (double *rhs, int i_k, WHO_PERMUTES whoperm)

  Numerical solve for GENERAL and NETWORK matrices (require sprsblkllt)
- void numeric_solve_MMt_diag (double *rhs, int i_k)

  *Numerical solve for IDENTITY, IDTY_IDTY, DIAGONAL, DIAG_DIAG (M*M' is diagonal)*
- void symbolic_fact_M_sprsblkllt_general (int nz, int *icola, int *inirowa)

  Computes symmetric ordering and symbolic factorization of symmetric upper triangular A using Ng-Peyton package.
- void numeric_fact_M_sprsblkllt_general (double *a)

  Computes numerical factorization of A of symmetric upper triangular A using Ng-Peyton package.
- void numeric_fact_M_diagonal (double *d1)

  Numerical factorization of D diagonal positive semidefinite matrix.
- void numeric_solve_M_sprsblkllt_general (double *rhs, WHO_PERMUTES whoperm)

  Numerical solve for symmetric matrices.
- void numeric_solve_M_diagonal (double *rhs)

  Numerical solve for DIAGONAL matrix.

Static Private Member Functions

- static bool comp_field1 (const SRC_DST_ARC &i, const SRC_DST_ARC &j)

  Auxiliary routine to compare first field of type SRC_DST_ARC, used in sorting arcs.
- static bool comp_field2 (const SRC_DST_ARC &i, const SRC_DST_ARC &j)

  Auxiliary routine to compare second field of type SRC_DST_ARC, used in sorting arcs.

Private Attributes

- CHOL_SOLVER chol_solver

  Cholesky solver to be used.
- TYPE_MATRIX type_matrix

  Type of matrix (general, network, identity, etc),.
- TYPE_ORIENTATION type_orientation

  Type of arc orientation for network matrices (by default, oriented).
- int m

  Dimension of system A∗A'/A : m x m matrix.
- int njka

  Nonzeros in diagonal and subdiagonal of A∗A'/A.
- int maxfillin

  Nonzeros in factorization of A∗A'/A due to fill-in.
- int maxlnz

  Total nonzeros in factorization of A∗A'/A (= maxfillin+njka).
- int k

  Number of matrices A∗A' with same topology to be factorized, see below explanation of lnz() and pnlz() (k=1 for A)
- int num_zero_pivots

  Number of zero pivots found during factorizations.
- int num_semidef_matrix

  Number of semidefinite matrices found during factorizations.
- double ∗ d1

  Diagonal of DIAG or first diagonal of DIAG_DIAG [D1 D2] matrices.
- double ∗ d2

  Second diagonal of DIAG_DIAG [D1 D2] matrices.

Vectors related to symmetric row-column permutation:

- int ∗ pfa

  pfa[i]= k : original row k is now (after permutation) in position i
- int ∗ ipfa

  ipfa[k]= i : i is current row (after permutation) of original row k.
- NUMBERING pfa_numbering

  Numbering style of pfa (used in C and Fortran code).
- double ∗ permrhs

  Auxliary array to store the permutation of the rhs of the system to be solved.

When chol_solver is sprsblkllt and A is 'GENERAL' matrix :

- int ∗ ia

  ia(m+1): pointer to first in ja().
- int ∗ ja

  ja(njka): pair of rows (ia,ja) of A that generate arc in graph of A∗A' .
- int ∗ ka

  ka(njka+1): pointer to first in la() and va().
- int ∗ la

  la(nla): columns that intervene in arc (i,j) in graph of A∗A'.
- int nla

  Size of la() and va().
- double ∗ va

va(nla): premultiplied values associated to column of la().

When chol_solver is sprsblkllt and A is 'NETWORK' matrix:

- int nnu

  Number of nodes.
- int nar

  Number of arcs.
- int * inik

  inik(nnu+1): pointers to first arc sorted by source (compressed form).
- int * dst2

  dst2(nar): destination nodes of arcs in inik.
- int * arck_l

  arck_l(nar): arcs of inik, dst2.
- int * inil

  inil(nnu+1): pointers to first arc sorted by destination (compressed form).
- int * src2

  src2(nar): source nodes of arcs in inik
- int * arcl_k

  arcl_k(nar): arcs of inil, src2.

Variables needed by sprsblkllt:

- int * xadj

  xadj(m+1): indices to adjacency matrix ajcncy().
- int * ajcncy

  ajcncy(2(njka-m)): 2*arcs in graph of A*A'/A without diagonal entries (i,i) of A*A'/A.*
- int * workspak
- int * xlnz

  xlnz(m+1): indices to first column/row in lnz(.).
- int * xlindx

  xlindx(maxsuper): indices to lindx (compressed form).
- int * lindx

  lindx(maxsub): compressed form of sub and diagonal of factorization.
- int maxsub

  Maximum size of some vectors, computed by sprsblkllt.
- int * colcnt

  colnct(m): elements per column in factorization.
- int * snode

  snode(m): supernode of each column.
- int maxsuper

  Maximum size of some vectors, computed by sprsblkllt.
- int * xsuper

  xsuper(m+1): pointer to first column of supernode.
- int iwsiz

  Size of integer working space, required by sprsblkllt.
- int * split

  split(m): splitting of supernodes for exploiting cache memory.
- int tmpsiz

  Size of temporary working space, required by sprsblkllt.
- double * lnz

lnz(k∗maxlnz): sub(upper) and diagonal elements of ATheta(k)A' (and its Cholesky factorization, once performed).

- double ∗∗ plnz

  plnz(k): pointers to lnz(i).

Variables with indices for fast filling of matrix to factorize:

- int ∗ ilnz

  ilnz(njka): indices to lnz(.) of each nonzero element of A∗Theta∗A'.
- int ∗ ifillin

  ifillin(maxfillin): fill-in positions in lnz(.) to be cleaned before filling lnz() with A∗Theta∗A'.
- int maxiarc

  Size of iarc.
- int ∗ iarc

  iarc(maxiarc): indices to arcs intervening in each element nonzero of lnz().
- int ∗ ini_arc

  ini_arc(njka+1): begin of arcs in iarc for each element of lnz().

Static Private Attributes

- static constexpr double MIN_PIVOT = 0.0

  Threshold pivot value for Cholesky factorizations of "simple" (diag, idty...) matrices (no pivot less than or equal to MIN_PIVOT allowed)
- static constexpr double POSITIVE_PIVOT = 1.0E+128

  New value for pivots below MIN_PIVOT in Cholesky factorizations of "simple" matrices (diag, idty...)

5.7.1 Detailed Description

Class for sparse Cholesky factorizations.

5.7.2 Constructor & Destructor Documentation

5.7.2.1 SparseChol::SparseChol ( )

Constructor. It calls initialize().

5.7.2.2 SparseChol::∼SparseChol ( )

Destructor. It calls free_mem().

5.7.3 Member Function Documentation

5.7.3.1 bool SparseChol::comp_field1 (const SRC_DST_ARC & *i*, const SRC_DST_ARC & *j*) [static], [private]

Auxiliary routine to compare first field of type SRC_DST_ARC, used in sorting arcs.

5.7.3.2 bool SparseChol::comp_field2 (const SRC_DST_ARC & *i*, const SRC_DST_ARC & *j*) [static], [private]

Auxiliary routine to compare second field of type SRC_DST_ARC, used in sorting arcs.

5.7.3.3 void SparseChol::free_mem ( ) [private]

Frees memory (uses C free(), not C++ delete[])

5.7.3.4 void SparseChol::get_ilnz_ifillin_general (int * *inp_ia*, int * *inp_ja*) [private]

Compute arrays of indices to lnz(): array ilnz(njka), for a general matrix.

For each pair (i,j) stated by inp_ia and inp_ja of diagonal and subdiagonal (or supradiagonal, matrix is symmetric) elements of A*Theta*A'/A ilnz(.) points to position in lnz(.) Array ifillin() with indices to fill-in positions in lnz() is also computed (to fastly clean those positions when needed).

Parameters

inp	ia Vector of beginning of rows of A*A'/A(compressed form)
inp_ja	Vector of column indices A*A'/A(.

5.7.3.5 void SparseChol::get_ilnz_network ( ) [private]

Computes variables and indices for later fast filling of lnz for a network matrix.

Computes variables and indices for later fast filling of lnz(): maxiarc, iarc(maxiarc), ini_arc(njka+1)

5.7.3.6 void SparseChol::get_indices_a_general (int * *icola*, int * *inirowa*, double * *a*) [private]

Computes data structure of arrays ia, ja, ka, la, va as detailed in Monma and Morton paper.

It deals with matrices with empy (zero) rows by adding a fictitious diagonal zero entry. The zero diagonal pivot will be later modified by the sparse Cholesky solver to deal with this positive semidefinite matrix.

Parameters

icola	Vector of column indices.
inirowa	Vector of beginning of rows (compressed form)
a	Vector of matrix elements.

5.7.3.7 int SparseChol::get_ipfa (int *i*) [inline]

Provides ipfa[i] with no check.

Returns

ipfa[i]

Note

The user must guarantee ipfa previously computed in call to symbolic_fact_MMt().

5.7.3.8 void SparseChol::get_ipk_ipl_network (int * *src*, int * *dst*) [private]

Computes data structures with network sorted by src and dst, to be used later.

Parameters

src	Vector of arc sources.
dst	Vector of arc destinations.

5.7.3.9 int SparseChol::get_maxfillin ( ) [inline]

Provides maxfillin with no check.

return maxfillin

Note

The user must guarantee maxfillin previously computed in call to symbolic_fact_MMt().

5.7.3.10 int SparseChol::get_maxlnz ( ) [inline]

Provides maxlnz with no check.

Returns

maxlnz

Note

The user must guarantee maxlnz previously computed in call to symbolic_fact_MMt().

5.7.3.11 int SparseChol::get_njka ( ) [inline]

Provides njka with no check.

Returns

njka

Note

The user must guarantee njka previously computed in call to symbolic_fact_MMt().

5.7.3.12 int SparseChol::get_num_semidef_matrix ( ) [inline]

Provides num_semidef_matrix with no check.

Returns

num_semidef_matrix

Note

It will 0 if no numeric factorization made.

5.7.3.13 int SparseChol::get_num_zero_pivots ( ) [inline]

Provides num_zero_pivots with no check.

Returns

num_zero_pivots

Note

It will 0 if no numeric factorization made.

5.7.3.14 int SparseChol::get_pfa (int *i*) [inline]

Provides pfa[i] with no check.

Returns

pfa[i]

Note

The user must guarantee pfa previously computed in call to symbolic_fact_MMt().

5.7.3.15 void SparseChol::get_pfa_ipfa_general (int * *inp_ia*, int * *inp_ja*) [private]

Computes row permutation of AA'/A for a general matrix.

Computes row permutation of AA'/A using minimum degree ordering to the graph associated to AA'/A (AA'/A is symmetric matrix). It calls routines orddmd() of sprskblkllt to compute pfa (direct permutation) and ipfa(inverse permutation): pfa[i]= k : original row k is now (after permutation) in position i ipfa[k]= i : i is current row (after permutation) of original row k Both pfa and ipfa needed because permutation may be nonsymmetric

WARNING: ordmmd() is a fortran routine, thus permutation in pfa/ipfa is numbered from 1, not from 0 as in C. 1-numbering needed for sprsblkllt routines. After calling sfnit() and symfct() then they can be numbered in C 0-starting style

Parameters

inp	ia Vector of beginning of rows of A∗A'/A(compressed form)
inp_ja	Vector of column indices A∗A'/A(.

5.7.3.16 void SparseChol::get_pfa_ipfa_network ( ) [private]

Computes row permutation of AA' for a network matrix A.

Computes row permutation of AA' using minimum degree ordering to the graph associated to AA' (A is network matrix without last node). It calls routines orddmd() of sprskblkllt to compute pfa (direct permutation) and ipfa(inverse permutation):

pfa[i]= k : original row k is now (after permutation) in position i

ipfa[k]= i : i is current row (after permutation) of original row k

Both pfa and ipfa needed because permutation may be nonsymmetric

Note

ordmmd() is a fortran routine, thus permutation in pfa/ipfa is numbered from 1, not from 0 as in C. 1-numbering needed for sprsblkllt routines. After calling sfnit() and symfct() then they can be numbered in C 0-starting style.

5.7.3.17 void SparseChol::initialize (CHOL_SOLVER *chol_solver* = (CHOL_SOLVER) NULL, TYPE_MATRIX *type_matrix* = (TYPE_MATRIX) NULL, TYPE_ORIENTATION *type_orientation* = ORIENTED)

Initialize attributes (set variables to 0, NULL...)

Parameters

chol_solver	Cholesky solver to be used
type_matrix	Type of matrix (general, network, IDTY, ...)
type_orientation	(Only for network matrices) Whether arcs are oriented or nonoriented.

5.7.3.18 void SparseChol::numeric_fact_M (double * a) [inline]

Computes numerical factorization for a positive semidefinite matrix A.

A is either diagonal or general symmetric upper triangular (with diagonal) matrix in compressed rowwise order

Parameters

a(nz)	Matrix elements (only diagonal and upper triangle, or diagonal)

Note

symbolic_fact_M() must have been previously called.

5.7.3.19 void SparseChol::numeric_fact_M_diagonal (double * d1) [private]

Numerical factorization of D diagonal positive semidefinite matrix.

When matrix is DIAG just allocate some vectors and fill some minimum information.

Parameters

d1	diagonal terms of D, d1 of dimension m

5.7.3.20 void SparseChol::numeric_fact_M_sprsblkllt_general (double * a) [private]

Computes numerical factorization of A of symmetric upper triangular A using Ng-Peyton package.

A is general symmetric (diagonal and upper triangle) matrix in compressed rowwise order We assume the topology (icola,inirowa) of matrix A is the same that was used in the symbolic factorization, otherwise the code may fail.

Parameters

a(nz	or njka) Matrix elements

5.7.3.21 void SparseChol::numeric_fact_MMt (double * Theta, int i_k = 0) [inline]

Computes numerical factorization of A∗Theta∗A'.

Parameters

Theta(n)	Diagonal matrix Theta of A∗Theta∗A', of size number of columns of A.
i_k	Which matrix to factorize, from 0 to k-1.

Note

symbolic_fact_MMt() must have been previously called.

5.7.3.22 void SparseChol::numeric_fact_MMt_diag_diag (double * Theta, int i_k) [private]

Numerical factorization of [D1 D2]∗(Theta$^\wedge$+, Theta$^\wedge$-)∗[D1 D2]' (DIAG_DIAG matrix)

Numerical factorization of [D1 D2]∗(Theta$^\wedge$+, Theta$^\wedge$-)∗[D1 D2]': for DIAG_DIAG just stores D1∗(Theta$^\wedge$+)∗D1 + D2∗(Theta$^\wedge$-)∗D2 in lnz.

Theta(n=2∗m), where Theta(0:m-1) is Theta$^\wedge$+ and Theta(m:n-1) is Theta$^\wedge$-

int i_k: which matrix to factorize, from 0 to k-1

5.7.3.23 void SparseChol::numeric_fact_MMt_diagonal (double * *Theta,* int *i_k*) [private]

Numerical factorization of D*Theta*D' (diagonal matrix)

Numerical factorization of D*Theta*D': when D is diagonal just stores D*Theta*D in lnz.

Theta(n): diagonal matrix of size of D int i_k: which matrix to factorize, from 0 to k-1

5.7.3.24 void SparseChol::numeric_fact_MMt_identity (double * *Theta,* int *i_k*) [private]

Numerical factorization of I*Theta*I'.

Numerical factorization of I*Theta*I': when I is identity just stores Theta in lnz.

Theta(n): diagonal matrix of size dimension of I int i_k: which matrix to factorize, from 0 to k-1

5.7.3.25 void SparseChol::numeric_fact_MMt_idty_idty (double * *Theta,* int *i_k*) [private]

Numerical factorization of [I I]*(Theta^+, Theta^-)*[I I]' matrix.

Numerical factorization of [I I]*(Theta^+, Theta^-)*[I I]': for IDTY_IDTY just stores (Theta^+) + (Theta^-) in lnz.

Theta(n=2*m), where Theta(0:m-1) is Theta^+ and Theta(m:n-1) is Theta^-

int i_k: which matrix to factorize, from 0 to k-1

5.7.3.26 void SparseChol::numeric_fact_MMt_sprsblkllt_general (double * *Theta,* int *i_k*) [private]

Computes numerical factorization of A*Theta*A' when A is general matrix.

Parameters

Theta(n)	Diagonal matrix Theta of A*Theta*A', of size number of columns of A
i_k	Which matrix to factorize, from 0 to k-1

5.7.3.27 void SparseChol::numeric_fact_MMt_sprsblkllt_network (double * *Theta,* int *i_k*) [private]

Computes numerical factorization of A*Theta*A' when A is a network.

Computes numerical factorization of A*Theta*A' when A is either an ORIENTED or NONORIENTED network matrix. Exploits that get_ilnz_network computed indices for fast filling of lnz for the ORIENTED case.

If ORIENTED:

A=N of size m x nar (nar=n), and A*Theta*A'= N*Theta*N', where

Theta(n): diagonal matrix Theta of A*Theta*A', of size number of columns of A

If NONORIENTED:

A=[N -N] of size m x (2*nar) (n=2*nar) and A*Theta*A'= N*(Theta^+ + Theta^-)*N' where

Theta(n=2*nar), where Theta(0:nar-1) is Theta^+ and Theta(nar:n-1) is Theta^-

Parameters

Theta	Diagonal Theta matrix.
i_k	Which matrix to factorize, from 0 to k-1.

5.7.3.28 void SparseChol::numeric_solve_M (double * *rhs*, WHO_PERMUTES *whoperm* = (WHO_PERMUTES) NULL) [inline]

Solve systems with a positive semidefinite symmetric matrix A and right-hand-side rhs.

The solution is returned by the same vector rhs (then it is overwritten).

Parameters

rhs	On input vector rhs; on output the solution vector.
whoperm	If CHOLESKY, then the reordering by pfa/ipfa must internally be applied to rhs by the routine; if USER_OF_CHOLESKY, then the user already provides rhs in the permutted order given by pfa/ipfa.

Note

symbolic_fact_M() and numeric_fact_M() must have been previously called.

5.7.3.29 void SparseChol::numeric_solve_M_diagonal (double * *rhs*) [private]

Numerical solve for DIAGONAL matrix.

Solves M*sol = rhs, when M is a diagonal positive semidefinite matrix.

Parameters

rhs	on input rhs vector, on output it contains the solution.

Previous calls to symbolic_fact_M and num_fact_M required for a correct "factorization" of M (i.e., storage of in lnz of the diagonal terms)

5.7.3.30 void SparseChol::numeric_solve_M_sprsblkllt_general (double * *rhs*, WHO_PERMUTES *whoperm* = (WHO_PERMUTES) NULL) [private]

Numerical solve for symmetric matrices.

Solves A*sol = rhs. On output, rhs contains the solution. Previous calls to symbolic_fact_M and numeric_fact_M required for a correct factorization of A

whoperm: if CHOLESKY, then the reordering by pfa/ipfa must internally be applied to rhs by the routine; if USER_OF_CHOLESKY, then the user already provides rhs in the permutted order given by pfa/ipfa.

5.7.3.31 void SparseChol::numeric_solve_MMt (double * *rhs*, int *i_k* = 0, WHO_PERMUTES *whoperm* = (WHO_PERMUTES) NULL) [inline]

Solve systems with matrix A*Theta*A' and right-hand-side rhs.

The solution is returned by the same vector rhs (then it is overwritten).

Parameters

rhs	On input vector rhs; on output the solution vector.
whoperm	If CHOLESKY, then the reordering by pfa/ipfa must internally be applied to rhs by the routine; if USER_OF_CHOLESKY, then the user already provides rhs in the permutted order given by pfa/ipfa.
i_k	Which matrix to use, from 0 to k-1.

Note

symbolic_fact_MMt() and numeric_fact_MMt() must have been previously called.

5.7.3.32 void SparseChol::numeric_solve_MMt_diag (double * *rhs*, int *i_k*) [private]

Numerical solve for IDENTITY, IDTY_IDTY, DIAGONAL, DIAG_DIAG (M*M' is diagonal)

Solves (M*Theta(i_k)*M')*sol = rhs, when (M*Theta(i_k)*M' is a diagonal matrix. This holds for matrices: IDENTITY, IDTY_IDTY, DIAGONAL, DIAG_DIAG

On output, rhs contains the solution. Previous calls to symbolic_fact_MMt and num_fact required for a correct "factorization" of M*Theta(i_k)*M (i.e., storage of in lnz(i_k) of the diagonal M*Theta(i_k)*M matrix)

5.7.3.33 void SparseChol::numeric_solve_MMt_sprsblkllt (double * *rhs*, int *i_k* = 0, WHO_PERMUTES *whoperm* = (WHO_PERMUTES) NULL) [private]

Numerical solve for GENERAL and NETWORK matrices (require sprsblkllt)

Solves (ATheta(i_k)A')*sol = rhs. On output, rhs contains the solution. Previous calls to symbolic_fact_MMt and num_fact required for a correct factorization of ATheta(i_k)A'

whoperm: if CHOLESKY, then the reordering by pfa/ipfa must internally be applied to rhs by the routine; if USER_OF_CHOLESKY, then the user already provides rhs in the permutted order given by pfa/ipfa.

int i_k: which matrix to use, from 0 to k-1

5.7.3.34 void SparseChol::reset (CHOL_SOLVER *chol_solver* = (CHOL_SOLVER) NULL, TYPE_MATRIX *type_matrix* = (TYPE_MATRIX) NULL, TYPE_ORIENTATION *type_orientation* = ORIENTED)

Like calling destructor+constructor.

Parameters

chol_solver	Cholesky solver to be used
type_matrix	Type of matrix (general, network, IDTY, ...)
type_orientation	(Only for network matrices) Whether arcs are oriented or nonoriented.

5.7.3.35 void SparseChol::symbolic_AThetaAt_A ( ) [private]

Symbolic factorization of AThetaA' or symmetric A.

It only requires xadj, ajcncy, which must have been previously created. Uses sfinit() and symfct() of sprsblkllt. Also uses bfinit() to prepare for forthcoming numerical factorizations.

5.7.3.36 void SparseChol::symbolic_fact_M (int *m*, int *nz*, int * *icola*, int * *inirowa*, int * *prov_pfa* = NULL, int * *prov_ipfa* = NULL, NUMBERING *prov_pfa_numbering* = NOT_COMPUTED) [inline]

Computes symmetric ordering and symbolic factorization of A for a general symmetric matrix.

A is general symmetric upper triangular (with diagonal) matrix in compressed rowwise order

Parameters

m	Number of rows/columns of A.
nz	Number of nonzeros in A.
icola(nz)	Column of each element of A (only diagonal and upper triangle)
inirowa(m+1)	Pointers to first by row in icola().

prov_pfa(m),prov_ipfa(m)	Ordering provided by user, to be used; the routine then only performs the symbolic factorization using this ordering. If NULL, then no ordering is provided and it will be computed. NULL is the default value if the parameter not provided. No check performed about the correctness of the ordering provided.
prov_pfa_numbering	Numbering of provided ordering.

5.7.3.37 void SparseChol::symbolic_fact_M (int *m*)

Computes symbolic factorization of DIAGONAL matrix A.

When matrix is DIAG just allocate some vectors and fill some minimum information.

Parameters

m	Dimension of A.

5.7.3.38 void SparseChol::symbolic_fact_M_sprsblkllt_general (int *nz*, int * *icola*, int * *inirowa*) [private]

Computes symmetric ordering and symbolic factorization of symmetric upper triangular A using Ng-Peyton package.

A is general symmetric (diagonal and upper triangle) matrix in compressed rowwise order

Parameters

nz	Number of nonzeros in A (diagonal and upper triangle).
icola(nz)	Column of each element of A.
inirowa(m+1)	Pointers to first by row in icola().

5.7.3.39 void SparseChol::symbolic_fact_MMt (int *m*, int *n*, int *nz*, int * *icola*, int * *inirowa*, double * *a*, int *k* = 1, int * *prov_pfa* = NULL, int * *prov_ipfa* = NULL, NUMBERING *prov_pfa_numbering* = NOT_COMPUTED) [inline]

Computes symmetric ordering and symbolic factorization of AA' for a general matrix A.

A is general matrix in compressed rowwise order.

Parameters

m	Number of rows of A.
n	Number of columns of A.
nz	Number of nonzeros in A.
icola(nz)	Column of each element of A.
inirowa(m+1)	Pointers to first by row in icola().
a(nz)	Matrix elements.
k	Number of matrices with same topology to be factorized.
prov_pfa(m),prov_ipfa(m)	Ordering provided by user, to be used; the routine then only performs the symbolic factorization using this ordering. If NULL, then no ordering is provided and it will be computed. NULL is the default value if the parameter not provided. No check performed about the correctness of the ordering provided.
prov_pfa_numbering	Numbering of provided ordering.

5.7.3.40 void SparseChol::symbolic_fact_MMt (int *nnu,* int *nar,* int * *src,* int * *dst,* int *k* = 1, int * *prov_pfa* = NULL, int * *prov_ipfa* = NULL, NUMBERING *prov_pfa_numbering* = NOT_COMPUTED) [inline]

Computes symmetric ordering and symbolic factorization of AA' for a network matrix A.

A is node-arc incidence matrix (last node not considered)

Parameters

nnu	Number of nodes.
nar	Number of arcs.
src(nar)	Source node for each arc.
dst(nar)	Destination node for each arc.
k	Number of matrices with same topology to be factorized.
prov_pfa(m),*prov_ipfa(m)*	Ordering provided by user, to be used; the routine then only performs the symbolic factorization using this ordering. If NULL, then no ordering is provided and it will be computed. NULL is the default value if the parameter not provided. No check performed about the correctness of the ordering provided.
prov_pfa_numbering	Numbering of provided ordering.

5.7.3.41 void SparseChol::symbolic_fact_MMt (int *m,* int *k* = 1)

Computes symbolic factorization of AA' for an IDENTITY or IDTY_IDTY matrix A.

When matrix is IDENTITY or IDTY_IDTY, just allocate some vectors and fill some minimum information

Parameters

m	Dimension of A.
k	Number of matrices with same topology to be factorized.

5.7.3.42 void SparseChol::symbolic_fact_MMt (int *m,* double * *d1_in,* int *k* = 1)

Computes symbolic factorization of AA' for a DIAG matrix A.

When matrix is DIAG just copy d1, allocate some vectors and fill some minimum information.

Parameters

m	Dimension of A.
d1_in(m)	Vector of diagonal elements.
k	Number of matrices with same topology to be factorized.

Note

d1_in(m) is copied, and not free'd.

5.7.3.43 void SparseChol::symbolic_fact_MMt (int *m,* double * *d1_in,* double * *d2_in,* int *k* = 1)

Computes symbolic factorization of AA' for a DIAG_DIAG matrix A.

When matrix is [D1 D2] just copy d1,d2, allocate some vectors and fill some minimum information.

Parameters

m	Dimension of A.
d1_in(m)	Vector of diagonal elements of D1.
d2_in(m)	Vector of diagonal elements of D2.
k	Number of matrices with same topology to be factorized.

Note

d1_in(m), d2_in(m) are copied, and not free'd.

5.7.3.44 void SparseChol::symbolic_fact_MMt_sprsblkllt_general (int *n*, int *nz*, int * *icola*, int * *inirowa*, double * *a*) [private]

Computes symmetric ordering and symbolic factorization for AA' using Ng-Peyton package for a general matrix.

A is general matrix in compressed rowwise order

Parameters

n	Number of columns of A.
nz	Number of nonzeros in A.
icola(nz)	Column of each element of A.
inirowa(m+1)	Pointers to first by row in icola().
a(nz)	Matrix elements.

5.7.3.45 void SparseChol::symbolic_fact_MMt_sprsblkllt_network (int * *src*, int * *dst*) [private]

Computes symmetric ordering and symbolic factorization for AA' using Ng-Peyton package for a network matrix.

Routines for both ORIENTED and NONORIENTED networks For ORIENTED the matrix is A=N For NONORIENTED the matrix is A=[N -N]. A∗A' is then N∗N'+(-N)(-N')= 2∗NN'. Then the the topology of A∗A' is the same for ORIENTED and NONORIENTED routines. Most routines are thus the same for the oriented and nonoriented case. The only difference is when computing A∗Theta∗A':

- for oriented network: A∗Theta∗A'=N∗Theta∗N'
- for nonoriented network: A∗Theta∗A'=[N -N]∗diag(Theta^+, Theta^-)∗[N -N']'= N∗(Theta^+)∗N'+N∗(Theta^-)∗N= N∗(Theta^+ + Theta^-)∗N'.

  Routine get_ilnz_network() computes indices for fast filling of lnz always considering the oriented network N. Therefore, the routine numeric_fact_MMt_sprsblkllt_network has to deal with both the oriented and nonoriented case later.

A is node-arc incidence matrix (last node not considered)

Parameters

src(nar)	Source node for each arc.
dst(nar)	Destination node for each arc,

5.7.4 Member Data Documentation

5.7.4.1 int* SparseChol::ajcncy [private]

ajcncy(2∗(njka-m)): 2∗arcs in graph of A∗A'/A without diagonal entries (i,i) of A∗A'/A.

5.7.4.2 int* SparseChol::arck_l [private]

arck_l(nar): arcs of inik, dst2.

5.7.4.3 int* SparseChol::arcl_k [private]

arcl_k(nar): arcs of inil, src2.

5.7.4.4 CHOL_SOLVER SparseChol::chol_solver [private]

Cholesky solver to be used.

5.7.4.5 int* SparseChol::colcnt [private]

colnct(m): elements per column in factorization.

5.7.4.6 double* SparseChol::d1 [private]

Diagonal of DIAG or first diagonal of DIAG_DIAG [D1 D2] matrices.

5.7.4.7 double* SparseChol::d2 [private]

Second diagonal of DIAG_DIAG [D1 D2] matrices.

5.7.4.8 int* SparseChol::dst2 [private]

dst2(nar): destination nodes of arcs in inik.

5.7.4.9 int* SparseChol::ia [private]

ia(m+1): pointer to first in ja().

When A is 'GENERAL' matrix:

ia,ja,ka,nla,la and va are internal indices for efficiently computing A*Theta*A' and symbolic factorization using the procedure described in "Monma, C. L. and A.J. Morton. 1987. Computational experience with a dual affine variant of Karmarkar's method for linear programming. Operations Research Letters V.6 n.6".

ia(m+1): pointer to first in ja(.)

ja(njka): pair of rows (ia,ja) of A that generate arc in graph of A*A'

ka(njka+1): pointer to first in la(.) and va(.)

nla: size of la() and va()

la(nla): columns that intervene in arc (i,j) in graph of A*A'

va(nla): premultiplied values associated to column of la(.)

5.7.4.10 int* SparseChol::iarc [private]

iarc(maxiarc): indices to arcs intervening in each element nonzero of lnz().

5.7.4.11 int* SparseChol::ifillin [private]

ifillin(maxfillin): fill-in positions in lnz(.) to be cleaned before filling lnz() with A*Theta*A'.

5.7.4.12 int* SparseChol::ilnz [private]

ilnz(njka): indices to lnz(.) of each nonzero element of A*Theta*A'.

Variables with indices for fast filling of lnz().

If A is 'GENERAL' matrix:

ilnz(njka): indices to lnz(.) of each nonzero element of A*Theta*A'/A.

ifillin(maxfillin): fill-in positions in lnz(.) to be cleaned before filling lnz() with A*Theta*A'.

If A is 'NETWORK' matrix:

iarc(maxiarc): indices to arcs intervening in each element nonzero of lnz().

maxiarc: size of iarc.

ini_arc(njka+1): begin of arcs in iarc for each element of lnz().

5.7.4.13 int* SparseChol::ini_arc [private]

ini_arc(njka+1): begin of arcs in iarc for each element of lnz().

5.7.4.14 int* SparseChol::inik [private]

inik(nnu+1): pointers to first arc sorted by source (compressed form).

5.7.4.15 int* SparseChol::inil [private]

inil(nnu+1): pointers to first arc sorted by destination (compressed form).

5.7.4.16 int* SparseChol::ipfa [private]

ipfa[k]= i : i is current row (after permutation) of original row k.

5.7.4.17 int SparseChol::iwsiz [private]

Size of integer working space, required by sprsblkllt.

5.7.4.18 int* SparseChol::ja [private]

ja(njka): pair of rows (ia,ja) of A that generate arc in graph of A*A' .

5.7.4.19 int SparseChol::k [private]

Number of matrices A*A' with same topology to be factorized, see below explanation of lnz() and pnlz() (k=1 for A)

5.7.4.20 int* SparseChol::ka [private]

ka(njka+1): pointer to first in la() and va().

5.7.4.21 int* SparseChol::la [private]

la(nla): columns that intervene in arc (i,j) in graph of A∗A'.

5.7.4.22 int* SparseChol::lindx [private]

lindx(maxsub): compressed form of sub and diagonal of factorization.

5.7.4.23 double* SparseChol::lnz [private]

lnz(k∗maxlnz): sub(upper) and diagonal elements of ATheta(k)A' (and its Cholesky factorization, once performed).

5.7.4.24 int SparseChol::m [private]

Dimension of system A∗A'/A : m x m matrix.

5.7.4.25 int SparseChol::maxfillin [private]

Nonzeros in factorization of A∗A'/A due to fill-in.

5.7.4.26 int SparseChol::maxiarc [private]

Size of iarc.

5.7.4.27 int SparseChol::maxlnz [private]

Total nonzeros in factorization of A∗A'/A (= maxfillin+njka).

5.7.4.28 int SparseChol::maxsub [private]

Maximum size of some vectors, computed by sprsblkllt.

5.7.4.29 int SparseChol::maxsuper [private]

Maximum size of some vectors, computed by sprsblkllt.

5.7.4.30 constexpr double SparseChol::MIN_PIVOT = 0.0 [static], [private]

Threshold pivot value for Cholesky factorizations of "simple" (diag, idty...) matrices (no pivot less than or equal to MIN_PIVOT allowed)

5.7.4.31 int SparseChol::nar [private]

Number of arcs.

5.7.4.32 int SparseChol::njka [private]

Nonzeros in diagonal and subdiagonal of A∗A'/A.

5.7.4.33 int SparseChol::nla [private]

Size of la() and va().

5.7.4.34 int SparseChol::nnu [private]

Number of nodes.

When A is 'NETWORK' matrix:

nnu, nar, ∗inik, ∗dst2, ∗arck_l, ∗inil, ∗src2, ∗arcl_k are internal indices and variables for efficiently computing A∗-Theta∗A' and symbolic factorization.

5.7.4.35 int SparseChol::num_semidef_matrix [private]

Number of semidefinite matrices found during factorizations.

5.7.4.36 int SparseChol::num_zero_pivots [private]

Number of zero pivots found during factorizations.

5.7.4.37 double∗ SparseChol::permrhs [private]

Auxliary array to store the permutation of the rhs of the system to be solved.

5.7.4.38 int∗ SparseChol::pfa [private]

pfa[i]= k : original row k is now (after permutation) in position i

Vector pfa of direct row permutation of matrix A∗A', and its inverse (ipfa) computed by Cholesky solver.

pfa[i]= k : original row k is now (after permutation) in position i.

ipfa[k]= i : i is current row (after permutation) of original row k.

Both pfa and ipfa needed because permutation may be nonsymmetric.

pfa_numbering for either C-0 or Fortran-1 numbering style.

permrhs(): auxiliary array, stores the permutation of the rhs when solving the system.

5.7.4.39 NUMBERING SparseChol::pfa_numbering [private]

Numbering style of pfa (used in C and Fortran code).

5.7.4.40 double∗∗ SparseChol::plnz [private]

plnz(k): pointers to lnz(i).

This is used when k matrices with same structure must be factorized; their share symbolic factorization, the only difference is in lnz(); we need a different lnz() for each of them; i=0..k-1, so first matrix is matrix 0, last matrix is k-1.

5.7.4.41 constexpr double SparseChol::POSITIVE_PIVOT = 1.0E+128 [static], [private]

New value for pivots below MIN_PIVOT in Cholesky factorizations of "simple" matrices (diag, idty...)

5.7.4.42 int* SparseChol::snode [private]

snode(m): supernode of each column.

5.7.4.43 int* SparseChol::split [private]

split(m): splitting of supernodes for exploiting cache memory.

5.7.4.44 int* SparseChol::src2 [private]

src2(nar): source nodes of arcs in inik

5.7.4.45 int SparseChol::tmpsiz [private]

Size of temporary working space, required by sprsblkllt.

5.7.4.46 TYPE_MATRIX SparseChol::type_matrix [private]

Type of matrix (general, network, identity, etc),.

5.7.4.47 TYPE_ORIENTATION SparseChol::type_orientation [private]

Type of arc orientation for network matrices (by default, oriented).

5.7.4.48 double* SparseChol::va [private]

va(nla): premultiplied values associated to column of la().

5.7.4.49 int* SparseChol::workspak [private]

5.7.4.50 int* SparseChol::xadj [private]

xadj(m+1): indices to adjacency matrix ajcncy().

Variables needed by sprsblkllt:

xadj(m+1): indices to adjacency matrix ajcncy().

ajcncy(2*(njka-m)): 2*arcs in graph of A*A'/A without diagonal entries (i,i) of A*A'/A.

colnct(m): elements per column in factorization.

snode(m): supernode of each column.

xsuper(m+1): pointer to first column of supernode.

xlindx(maxsuper): indices to lindx (compressed form).

lindx(maxsub): compressed form of sub and diagonal of factorization.

split(m): splitting of supernodes for exploiting cache memory.

xlnz(m+1): indices to first column/row in lnz(.).

lnz(k*maxlnz): sub(upper) and diagonal elements of ATheta(k)A' (and its Cholesky factorization, once performed).

plnz(k): pointer to lnz(i); this is used when k matrices with same structure must be factorized; their share symbolic factorization, thes only difference is in lnz(); we need a different lnz() for each of them; i=0..k-1, so first matrix is matrix 0, last matrix is k-1.

5.7.4.51 int* SparseChol::xlindx [private]

xlindx(maxsuper): indices to lindx (compressed form).

5.7.4.52 int* SparseChol::xlnz [private]

xlnz(m+1): indices to first column/row in lnz(.).

5.7.4.53 int* SparseChol::xsuper [private]

xsuper(m+1): pointer to first column of supernode.

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

- /home/jcastro2/intpoint/BlockIP/BlockIP/src_blockip-v2/SparseChol.h
- /home/jcastro2/intpoint/BlockIP/BlockIP/src_blockip-v2/SparseChol.C
- /home/jcastro2/intpoint/BlockIP/BlockIP/src_blockip-v2/SparseCholSprsblkllt.C

5.8 SparseChol::SRC_DST_ARC Struct Reference

Auxiliary struct for sorting network structure.

Public Attributes

- int src
  Source node of arc.
- int dst
  Destination node of arc.
- int arc
  Arc.

5.8.1 Detailed Description

Auxiliary struct for sorting network structure.

5.8.2 Member Data Documentation

5.8.2.1 int SparseChol::SRC_DST_ARC::arc

Arc.

5.8.2.2 int SparseChol::SRC_DST_ARC::dst

Destination node of arc.

5.8.2.3 int SparseChol::SRC_DST_ARC::src

Source node of arc.

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

- /home/jcastro2/intpoint/BlockIP/BlockIP/src_blockip-v2/SparseChol.h

5.9 StdForm Class Reference

Class for perform conversions between standard form and original problem.

`#include <StdForm.h>`

Classes

- struct Transform

Public Types

- enum TYPE_TRANSFORM {
  NON_ZERO_LB, _INF_LB, FREE_VAR, X_,
  SLACK, SURPLUS }

Public Member Functions

- StdForm ()

  Constructor.
- ~StdForm ()

  Destructor.
- void fobj_stdform (int n, double x[], double &fx, double Gx[], double Hx[], void *params)

  Function to calculate the objective function in a transformed variables point.
- void transformed_to_original_primal_variables (double *x)

  Converts a point from transformed variables to original variables space.
- void transformed_to_original_dual_variables (double *y, double *w, double *z)
- void original_to_transformed_primal_variables (double *&xout, double *xin)

  Converts a point from original variables to transformed variables space.
- void transform_linear_and_quadratic_cost (double *&cost, double *&qcost, double &constant)

  Transforms objective function from original variables to transformed variables space.
- void original_to_transformed_names (string *&outVarNames, string *inVarNames, string *&outConsNames, string *inConsNames)

  Converts the variable names from original variables to transformed variables space.
- void delete_new_variables (double *&xout, double *xin)

Public Attributes

- int numOrigVars

  Number of variables in the original problem.
- int numTransfVars

  Number of variables in the transformed (standardized) problem.
- int m_cons

  Number of constraints (including slacks) in the standardized problem.
- int numTransforms

  Number of transformations performed in variables.
- int * origVars

  Index to the original variables (that need a transformatiBon in objective function) in original problem, of size numTransforms.
- int * transfVars

  Index to the original variables (that need a transformation in objective function) in standardized problem, of size numTransforms.

- Transform * transforms

  Transformations performed in variables, of size numTransforms.
- double constantFObj

  Constant to add in the objective function.
- vector< int > * deletedRows

  Index to the rows that have been deleted.
- double * xOrig

  Optimal solution in the original problem.
- double * yOrig

  Optimal dual variables in the original problem.
- double * GxOrig

  Gradient in x in the original problem.
- double * HxOrig

  Hessian in x in the original problem.
- double * wOrig

  Optimal dual variables of x_i <= u_i bounds in the original problem.
- double * zOrig

  Optimal dual variables of x_i >= 0 bounds in the original problem.
- void(* fobj)(int n, double x[], double &fx, double Gx[], double Hx[], void *params)

  User function to calculate the objective function in a point.

5.9.1 Detailed Description

Class for perform conversions between standard form and original problem.

5.9.2 Member Enumeration Documentation

5.9.2.1 enum StdForm::TYPE_TRANSFORM

Enumerator:

NON_ZERO_LB

_INF_LB

FREE_VAR

X_

SLACK

SURPLUS

5.9.3 Constructor & Destructor Documentation

5.9.3.1 StdForm::StdForm ( )

Constructor.

5.9.3.2 StdForm::~StdForm ( )

Destructor.

5.9.4 Member Function Documentation

5.9.4.1 void StdForm::delete_new_variables (double *& *xout*, double * *xin*)

Parameters

xout	Vector in original variables space, is allocated by the function with new, must be freed with delete[]
xin	Vector in transformed variables space

5.9.4.2 void StdForm::fobj_stdform (int *n*, double *x[]*, double & *fx*, double *Gx[]*, double *Hx[]*, void * *params*)

Function to calculate the objective function in a transformed variables point.

Converts a point from transformed variables to original variables space and call the user function to calculate the objective function in that point, after that inverts the transformations to return the information in the transformed variables space

Parameters

n	Number of variables in transformed variables space
x	Point in transformed variables space
fx	Objective function value in x
Gx	Gradient in x
Hx	Hessian in x
params	User parameters to perform objective function calculations

5.9.4.3 void StdForm::original_to_transformed_names (string *& *outVarNames*, string * *inVarNames*, string *& *outConsNames*, string * *inConsNames*)

Converts the variable names from original variables to transformed variables space.

Parameters

outVarNames	Variable names including slacks, output in transformed variables space
inVarNames	Variable names including slacks, input in original variables space
outConsNames	Constraint names including linking constraints, output in transformed variables space
inConsNames	Constraint names including linking constraints, input in original variables space

5.9.4.4 void StdForm::original_to_transformed_primal_variables (double *& *xout*, double * *xin*)

Converts a point from original variables to transformed variables space.

Parameters

xout	Point in transformed variables space, is allocated by the function with new, must be freed with delete[]
xin	Point in original variables space

5.9.4.5 void StdForm::transform_linear_and_quadratic_cost (double *& *cost*, double *& *qcost*, double & *constant*)

Transforms objective function from original variables to transformed variables space.

Parameters

cost	Linear cost of variables including slacks, input in original variables space, output in transformed variables space
qcost	Quadratic cost of variables including slacks, input in original variables space, output in transformed variables space
constant	Constant to add in the objective function

5.9.4.6 void StdForm::transformed_to_original_dual_variables (double * y, double * w, double * z)

Converts the dual variables y of constraints, and (z,w) of lower and upper bounds from transformed to original space.

Constraints of the form lhs <= Ax <= rhs are transformed to Ax+s= rhs, 0<= s<= rhs-lhs. y are the multipliers of Ax+s= rhs, and yOrig_l and yOrig_r the multipliers for, respectively, lhs <= Ax and Ax <=rhs. It can be seen that

- if y(i)>=0 then yOrig_r(i)= y(i) and yOrig_l(i)=0;
- if y(i)<=0 then yOrig_l(i)= 0 and and yOrig_l(i)=-y(i). We will only use a single yOrig=y vector, such that when yOrig(i) is positive it is yOrig_r(i), and when negative it is -yOrig_l(i).

Multipliers z are related to upper bounds x<=u, while multipliers w are related to lower bounds l<= x.

Variables without upper bounds do not have z; we will set the corresponding components zOrig(i)=0.

Free variables do not have z and w. We will set the corresponding components of zOrig(i)=wOrig(i)=0.

For variables with the transformation NON_ZERO_LB (x∼= x-l) we have that zOrig(i)= z(i) and wOrig(i)= w(i).

For variables with the transformation _INF_LB (x∼= -x+u) we have that zOrig(i)= w(i) and wOrig(i)= 0 =z(i).

Parameters

y	Dual variables of constraint in standardized problem
w	Dual variables of x_i <= u_i bounds in standardized problem
z	Dual variables of x_i >= 0 bounds in standardized problem

Note

Dual variables in original problem are stored in yOrig, wOrig and zOrig

5.9.4.7 void StdForm::transformed_to_original_primal_variables (double * x)

Converts a point from transformed variables to original variables space.

Parameters

x	Point in transformed variables space

Note

Point in original variables space is stored in xOrig

5.9.5 Member Data Documentation

5.9.5.1 double StdForm::constantFObj

Constant to add in the objective function.

5.9.5.2 vector<int>* StdForm::deletedRows

Index to the rows that have been deleted.

5.9.5.3 void(* StdForm::fobj)(int n, double x[], double &fx, double Gx[], double Hx[], void *params)

User function to calculate the objective function in a point.

5.9.5.4 double* StdForm::GxOrig

Gradient in x in the original problem.

5.9.5.5 double* StdForm::HxOrig

Hessian in x in the original problem.

5.9.5.6 int StdForm::m_cons

Number of constraints (including slacks) in the standardized problem.

5.9.5.7 int StdForm::numOrigVars

Number of variables in the original problem.

5.9.5.8 int StdForm::numTransforms

Number of transformations performed in variables.

5.9.5.9 int StdForm::numTransfVars

Number of variables in the transformed (standardized) problem.

5.9.5.10 int* StdForm::origVars

Index to the original variables (that need a transformatiBon in objective function) in original problem, of size numTransforms.

5.9.5.11 Transform* StdForm::transforms

Transformations performed in variables, of size numTransforms.

5.9.5.12 int* StdForm::transfVars

Index to the original variables (that need a transformation in objective function) in standardized problem, of size numTransforms.

5.9.5.13 double* StdForm::wOrig

Optimal dual variables of x_i <= u_i bounds in the original problem.

5.9.5.14 double* StdForm::xOrig

Optimal solution in the original problem.

5.9.5.15 double* StdForm::yOrig

Optimal dual variables in the original problem.

5.9.5.16 double* StdForm::zOrig

Optimal dual variables of x_i >= 0 bounds in the original problem.

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

- /home/jcastro2/intpoint/BlockIP/BlockIP/src_blockip-v2/StdForm.h
- /home/jcastro2/intpoint/BlockIP/BlockIP/src_blockip-v2/StdForm.C

5.10 StdForm::Transform Struct Reference

`#include <StdForm.h>`

Public Attributes

- TYPE_TRANSFORM type
  < *To save data related to one transformation*
- double bound
- int x_index

5.10.1 Member Data Documentation

5.10.1.1 double StdForm::Transform::bound

The bound when the type of transformation is Non Zero Lower Bound (x = x∼ + lb) or -Infinity Lower Bound transformations (x = -x∼ + ub)

5.10.1.2 TYPE_TRANSFORM StdForm::Transform::type

< To save data related to one transformation

Type of transformation

5.10.1.3 int StdForm::Transform::x_index

Index to the x- variable when the type of transformation is Free Variable (x = x+ - x-)

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

- /home/jcastro2/intpoint/BlockIP/BlockIP/src_blockip-v2/StdForm.h

Chapter 6

File Documentation

6.1 /home/jcastro2/intpoint/BlockIP/BlockIP/sprsblkllt_Ng_Peyton/sprsblkllt.h File Reference

Functions

- void ORDMMD (int ∗nnux, int ∗xadj, int ∗ajcncy2, int ∗ipfa, int ∗pfa, int ∗iwsiz, int ∗iwork, int ∗nofsub, int ∗iflag)
- void SFINIT (int ∗nnux, int ∗nnza, int ∗xadj, int ∗ajcncy2, int ∗pfa, int ∗ipfa, int ∗colcnt, int ∗nnzl, int ∗nsub, int ∗nsuper, int ∗snode, int ∗xsuper, int ∗iwsiz, int ∗iwork, int ∗iflag)
- void SYMFCT (int ∗nnux, int ∗maxajc, int ∗xadj, int ∗ajcncy, int ∗pfa, int ∗ipfa, int ∗colcnt, int ∗maxsuper, int ∗xsuper, int ∗snode, int ∗maxsub, int ∗xlindx, int ∗lindx, int ∗xlnz, int ∗iwsiz, int ∗workspak, int ∗iflag)
- void BFINIT (int ∗nnux, int ∗maxsuper, int ∗xsuper, int ∗snode, int ∗xlindx, int ∗lindx, int ∗i, int ∗tmpsiz, int ∗split)
- void BLKFCT (int ∗nnux, int ∗maxsuper, int ∗xsuper, int ∗snode, int ∗split, int ∗xlindx, int ∗lindx, int ∗xlnz, double ∗lnzk, int ∗iwsiz, int ∗workspak, int ∗tmpsiz, double ∗tmpvec, int ∗iret, void(mmpy4)(int ∗m, int ∗n, int ∗q, int ∗xpnt, double ∗x, double ∗y, int ∗ldy), void(smxpy4)(int ∗m, int ∗n, double ∗y, int ∗apnt, double ∗a))
- void MMPY4 (int ∗m, int ∗n, int ∗q, int ∗xpnt, double ∗x, double ∗y, int ∗ldy)
- void SMXPY4 (int ∗m, int ∗n, double ∗y, int ∗apnt, double ∗a)
- void BLKSLV (int ∗maxsuper, int ∗xsuper, int ∗xlindx, int ∗lindx, int ∗xlnz, double ∗lnzk, double ∗vk)

6.1.1 Function Documentation

6.1.1.1 void BFINIT (int ∗ *nnux,* int ∗ *maxsuper,* int ∗ *xsuper,* int ∗ *snode,* int ∗ *xlindx,* int ∗ *lindx,* int ∗ *i,* int ∗ *tmpsiz,* int ∗ *split*)

6.1.1.2 void BLKFCT (int ∗ *nnux,* int ∗ *maxsuper,* int ∗ *xsuper,* int ∗ *snode,* int ∗ *split,* int ∗ *xlindx,* int ∗ *lindx,* int ∗ *xlnz,* double ∗ *lnzk,* int ∗ *iwsiz,* int ∗ *workspak,* int ∗ *tmpsiz,* double ∗ *tmpvec,* int ∗ *iret,* void(mmpy4)(int ∗m, int ∗n, int ∗q, int ∗xpnt, double ∗x, double ∗y, int ∗ldy) , void(smxpy4)(int ∗m, int ∗n, double ∗y, int ∗apnt, double ∗a))

6.1.1.3 void BLKSLV (int ∗ *maxsuper,* int ∗ *xsuper,* int ∗ *xlindx,* int ∗ *lindx,* int ∗ *xlnz,* double ∗ *lnzk,* double ∗ *vk*)

6.1.1.4 void MMPY4 (int ∗ *m,* int ∗ *n,* int ∗ *q,* int ∗ *xpnt,* double ∗ *x,* double ∗ *y,* int ∗ *ldy*)

6.1.1.5 void ORDMMD (int ∗ *nnux,* int ∗ *xadj,* int ∗ *ajcncy2,* int ∗ *ipfa,* int ∗ *pfa,* int ∗ *iwsiz,* int ∗ *iwork,* int ∗ *nofsub,* int ∗ *iflag*)

6.1.1.6 void SFINIT (int ∗ *nnux,* int ∗ *nnza,* int ∗ *xadj,* int ∗ *ajcncy2,* int ∗ *pfa,* int ∗ *ipfa,* int ∗ *colcnt,* int ∗ *nnzl,* int ∗ *nsub,* int ∗ *nsuper,* int ∗ *snode,* int ∗ *xsuper,* int ∗ *iwsiz,* int ∗ *iwork,* int ∗ *iflag*)

6.1.1.7 void SMXPY4 (int * *m,* int * *n,* double * *y,* int * *apnt,* double * *a*)

6.1.1.8 void SYMFCT (int * *nnux,* int * *maxajc,* int * *xadj,* int * *ajcncy,* int * *pfa,* int * *ipfa,* int * *colcnt,* int * *maxsuper,* int * *xsuper,* int * *snode,* int * *maxsub,* int * *xlindx,* int * *lindx,* int * *xlnz,* int * *iwsiz,* int * *workspak,* int * *iflag*)

6.2 /home/jcastro2/intpoint/BlockIP/BlockIP/src_blockip-v2/BlockIP.C File Reference

```
#include "BlockIP.h"
#include "ExceptionBlockIP.h"
#include "math.h"
#include <algorithm>
#include <iostream>
#include <sstream>
#include <fstream>
#include <limits>
#include <unordered_map>
#include <stdexcept>
```

Functions

- void write_matrix_BlockIP_format (MatrixBlockIP *M, ofstream &outfile)

  write matrix M for BlockIPformat
- bool myGetLine (ifstream &file, string &line, int &numLine)

6.2.1 Function Documentation

6.2.1.1 bool myGetLine (ifstream & *file,* string & *line,* int & *numLine*) [inline]

6.2.1.2 void write_matrix_BlockIP_format (MatrixBlockIP * *M,* ofstream & *outfile*) [inline]

write matrix M for BlockIPformat

6.3 /home/jcastro2/intpoint/BlockIP/BlockIP/src_blockip-v2/BlockIP.h File Reference

Definition of BlockIP.

```
#include "MatrixBlockIP.h"
#include "StdForm.h"
```

Classes

- class BlockIP

  Main class for loading and solving problems.
- struct BlockIP::BackupLnk

6.3.1 Detailed Description

Definition of BlockIP.

6.4 /home/jcastro2/intpoint/BlockIP/BlockIP/src_blockip-v2/BlockIPminimize.C File Reference

```
#include "math.h"
#include "misc.h"
#include "pcg.h"
#include "ritz_value.h"
#include <iostream>
#include "BlockIP.h"
#include "ExceptionBlockIP.h"
#include <limits>
```

Macros

- #define CHECK(V) if (V[i] != 0) cout << "Warning: Inactive " << listInactiveLnk[j] <<" i=" << i << " " << #V <<"[i]= " << V[i] <<endl;
- #define CHECK2(V) if (V[i] != 0) cout << "Warning: variable " << listWithoutUb[j] <<" i=" << i << " " << #V <<"[i]= " << V[i] <<endl;

Functions

- int PCG_Hv_wrapper (int nn, double *v, double *Hv, void *t)

  Nonmember wrapper function to callback PCG member function min_PCG_Hv.
- int PCG_Hz_eq_r_wrapper (int nn, double *zz, double *rr, int(*atimes)(int, double *, double *, void *), void *t)

  Non-member wrapper function to callback PCG member function min_PCG_Hz_eq_r_wrapper.
- int PCG_Theta0z_eq_r_wrapper (int nn, double *zz, double *rr, int(*atimes)(int, double *, double *, void *), void *t)

  Non-member wrapper function to callback PCG member function min_PCG_Theta0z_eq_r_wrapper.

6.4.1 Macro Definition Documentation

6.4.1.1 #define CHECK(V) if (V[i] != 0) cout << "Warning: Inactive " << listInactiveLnk[j] <<" i=" << i << " " << #V <<"[i]= " << V[i] <<endl;

6.4.1.2 #define CHECK2(V) if (V[i] != 0) cout << "Warning: variable " << listWithoutUb[j] <<" i=" << i << " " << #V <<"[i]= " << V[i] <<endl;

6.4.2 Function Documentation

6.4.2.1 int PCG_Hv_wrapper (int *nn*, double * *v*, double * *Hv*, void * *t*) [inline]

Nonmember wrapper function to callback PCG member function min_PCG_Hv.

6.4.2.2 int PCG_Hz_eq_r_wrapper (int *nn*, double * *zz*, double * *rr*, int(*)(int, double *, double *, void *) *atimes*, void * *t*) [inline]

Non-member wrapper function to callback PCG member function min_PCG_Hz_eq_r_wrapper.

6.4.2.3 int PCG_Theta0z_eq_r_wrapper (int *nn*, double * *zz*, double * *rr*, int(*)(int, double *, double *, void *) *atimes*, void * *t*) [inline]

Non-member wrapper function to callback PCG member function min_PCG_Theta0z_eq_r_wrapper.

6.5 /home/jcastro2/intpoint/BlockIP/BlockIP/src_blockip-v2/ExceptionBlockIP.C File Reference

```
#include "ExceptionBlockIP.h"
#include <iostream>
#include <sstream>
#include <cstring>
#include <string>
```

6.6 /home/jcastro2/intpoint/BlockIP/BlockIP/src_blockip-v2/ExceptionBlockIP.h File Reference

Definition of ExceptionBlockIP.

```
#include <exception>
#include <string>
```

Classes

- class ExceptionBlockIP

  Class for BlockIP exceptions.

Enumerations

- enum TYPE_ERROR {
  OUT_OF_MEMORY, NOIJA, NOIJA_GENERAL, CHOLMOD_NOT_IMPLEMENTED,
  SEMIDEFINITE, ORDMMD_PROBLEMS, SFINIT_PROBLEMS, SYMFCT_PROBLEMS,
  GET_ILNZ_IFILLIN_GENERAL_PROBLEMS, GET_ILNZ_IFILLIN_NETWORK_PROBLEMS, BLKFCT_PROBLEMS, COLUMN_OUT_OF_RANGE,
  ROW_OUT_OF_RANGE, HESS_NOT_ZERO, FOBJ_NULL, ZERO_BLOCKS,
  N_NULL, L_COLUMNS, L_ROWS, SAME_N_CONS,
  SAME_NL_FREE_VAR, SAME_NL_LB_INF_VAR, NONLINMPS, OPEN_FILE_WRITE,
  OPEN_FILE_READ, BLOCKIP_WRONG_FORMAT, MPS_WRONG_FORMAT, MPS_CONSTANT_DEF,
  MPS_CONTANT_CONS, MPS_SLACK_WITH_NO_LIN, MPS_BLOCK, MPS_CONS,
  MPS_VAR, MPS_VAR_REP, MPS_LIN_COST_REP, MPS_CONS_VAR_REP,
  MPS_CONS_VAR_BLOCK, MPS_LINK_WITHOUT_SLACK, MPS_SLACK_ALREADY_DEFINED, MPS_SLACK_COEF,
  MPS_FREE_CONS_BOUNDS, MPS_RHS_DEF, MPS_RHS_NOT_DEF, MPS_RANGES_DEF,
  MPS_BOUNDS_DEF, MPS_CONSTANT_BOUNDS, MPS_QUADOBJ, MPS_QUADOBJ_DEF,
  NOT_FOR_THIS_MATRIX_TYPE, QCOST_NOT_SEMIDEF, TYPE_STARTPOINT_TODO, MEHROTRA_NOT_IMPLEMENTED,
  HYBRIDPCG_NOT_IMPLEMENTED, INFINITY_RHS }

6.6.1 Detailed Description

Definition of ExceptionBlockIP.

6.6.2 Enumeration Type Documentation

6.6.2.1 enum TYPE_ERROR

Enumerator:

OUT_OF_MEMORY
NOIJA
NOIJA_GENERAL
CHOLMOD_NOT_IMPLEMENTED
SEMIDEFINITE
ORDMMD_PROBLEMS
SFINIT_PROBLEMS
SYMFCT_PROBLEMS
GET_ILNZ_IFILLIN_GENERAL_PROBLEMS
GET_ILNZ_IFILLIN_NETWORK_PROBLEMS
BLKFCT_PROBLEMS
COLUMN_OUT_OF_RANGE
ROW_OUT_OF_RANGE
HESS_NOT_ZERO
FOBJ_NULL
ZERO_BLOCKS
N_NULL
L_COLUMNS
L_ROWS
SAME_N_CONS
SAME_NL_FREE_VAR
SAME_NL_LB_INF_VAR
NONLINMPS
OPEN_FILE_WRITE
OPEN_FILE_READ
BLOCKIP_WRONG_FORMAT
MPS_WRONG_FORMAT
MPS_CONSTANT_DEF
MPS_CONTANT_CONS
MPS_SLACK_WITH_NO_LIN
MPS_BLOCK
MPS_CONS
MPS_VAR
MPS_VAR_REP
MPS_LIN_COST_REP
MPS_CONS_VAR_REP
MPS_CONS_VAR_BLOCK

MPS_LINK_WITHOUT_SLACK

MPS_SLACK_ALREADY_DEFINED

MPS_SLACK_COEF

MPS_FREE_CONS_BOUNDS

MPS_RHS_DEF

MPS_RHS_NOT_DEF

MPS_RANGES_DEF

MPS_BOUNDS_DEF

MPS_CONSTANT_BOUNDS

MPS_QUADOBJ

MPS_QUADOBJ_DEF

NOT_FOR_THIS_MATRIX_TYPE

QCOST_NOT_SEMIDEF

TYPE_STARTPOINT_TODO

MEHROTRA_NOT_IMPLEMENTED

HYBRIDPCG_NOT_IMPLEMENTED

INFINITY_RHS

6.7 /home/jcastro2/intpoint/BlockIP/BlockIP/src_blockip-v2/MatrixBlockIP.C File Reference

```
#include <iostream>
#include <algorithm>
#include <vector>
#include "MatrixBlockIP.h"
#include "ExceptionBlockIP.h"
```

Macros

- #define ALLOC(n, type) (type *)malloc((n)*sizeof(type))
- #define REALLOC(ptr, n, type) (type *)realloc(((char *)ptr),(n)*sizeof(type))
- #define FREE(p) if (p) {free(p); (p)= NULL;}
- #define TRY_ALLOC(ptr, n, type) if (!(ptr= ALLOC((n),type))) {throw ExceptionBlockIP(OUT_OF_MEMORY, __FILE__, __LINE__);}
- #define TRY_REALLOC(ptr, n, type) if (!(ptr= REALLOC((ptr),(n),type))) {throw ExceptionBlockIP(OUT_OF_MEMORY, __FILE__, __LINE__);}

6.7.1 Macro Definition Documentation

6.7.1.1 #define ALLOC(*n, type*) (type *)malloc((n)*sizeof(type))

6.7.1.2 #define FREE(*p*) if (p) {free(p); (p)= NULL;}

6.7.1.3 #define REALLOC(*ptr, n, type*) (type *)realloc(((char *)ptr),(n)*sizeof(type))

6.7.1.4 #define TRY_ALLOC(*ptr, n, type*) if (!(ptr= ALLOC((n),type))) {throw ExceptionBlockIP(OUT_OF_MEMORY, __FILE__, __LINE__);}

6.7.1.5 #define TRY_REALLOC(ptr, n, type) if (!(ptr= REALLOC((ptr),(n),type))) {throw ExceptionBlockIP(OUT_OF_MEMORY, __FILE__, __LINE__);}

6.8 /home/jcastro2/intpoint/BlockIP/BlockIP/src_blockip-v2/MatrixBlockIP.h File Reference

Definition of MatrixBlockIP.

```
#include "SparseChol.h"
#include <vector>
#include <fstream>
```

Classes

- class MatrixBlockIP

  Class for manipulating matrices, and interfacing SparseChol.
- struct MatrixBlockIP::Order_ija

  Auxiliary struct for sorting matrices in ija format.
- struct MatrixBlockIP::Order_vector

  Auxiliary struct for sorting vectors.

6.8.1 Detailed Description

Definition of MatrixBlockIP.

6.9 /home/jcastro2/intpoint/BlockIP/BlockIP/src_blockip-v2/misc.C File Reference

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <float.h>
#include "misc.h"
```

Macros

- #define SMALLdp 2.22507385850720e-308 /* Smallest (pos) binary float */
- #define HUGEdp 1.79769313486232e+308 /* Largest binary float */
- #define abs(x) ((x)>0.0?(x):-(x))

Functions

- void write_double (double *v, int i, int j, char *nom)
- void write_int (int *v, int i, int j, char *nom)
- int idamax (int n, double *sx, int incx)
- void daxpy (int n, double sa, double *sx, int incx, double *sy, int incy)
- void daxpyx (int n, double sa, double *sx, int incx, double *sy, int incy)
- void dcopy (int n, double *sx, int incx, double *sy, int incy)
- double ddot (int n, double *sx, int incx, double *sy, int incy)
- double dnrm2 (int n, double *sx, int incx)

- void dexopy (int n, double *v, double *x, double *y, int itype)
- void dfill (int n, double *v, double val)
- double r2mach ()
- double norm2 (int nn, double *v)
- double prod_esc (int nn, double *v1, double *v2)
- void dcopia (int nn, double *v1, double *v2)
- int indmin (int nn, double *vv)
- int indmax (int nn, double *vv)

6.9.1 Macro Definition Documentation

6.9.1.1 #define abs(x) ((x)>0.0?(x):-(x))

6.9.1.2 #define HUGEdp 1.79769313486232e+308 /* Largest binary float */

6.9.1.3 #define SMALLdp 2.22507385850720e-308 /* Smallest (pos) binary float */

6.9.2 Function Documentation

6.9.2.1 void daxpy (int *n*, double *sa*, double * *sx*, int *incx*, double * *sy*, int *incy*)

6.9.2.2 void daxpyx (int *n*, double *sa*, double * *sx*, int *incx*, double * *sy*, int *incy*)

6.9.2.3 void dcopia (int *nn*, double * *v1*, double * *v2*)

6.9.2.4 void dcopy (int *n*, double * *sx*, int *incx*, double * *sy*, int *incy*)

6.9.2.5 double ddot (int *n*, double * *sx*, int *incx*, double * *sy*, int *incy*)

6.9.2.6 void dexopy (int *n*, double * *v*, double * *x*, double * *y*, int *itype*)

6.9.2.7 void dfill (int *n*, double * *v*, double *val*)

6.9.2.8 double dnrm2 (int *n*, double * *sx*, int *incx*)

6.9.2.9 int idamax (int *n*, double * *sx*, int *incx*)

6.9.2.10 int indmax (int *nn*, double * *vv*)

6.9.2.11 int indmin (int *nn*, double * *vv*)

6.9.2.12 double norm2 (int *nn*, double * *v*)

6.9.2.13 double prod_esc (int *nn*, double * *v1*, double * *v2*)

6.9.2.14 double r2mach ( )

6.9.2.15 void write_double (double * *v*, int *i*, int *j*, char * *nom*)

6.9.2.16 void write_int (int * *v*, int *i*, int *j*, char * *nom*)

6.10 /home/jcastro2/intpoint/BlockIP/BlockIP/src_blockip-v2/misc.h File Reference

Functions

- void write_double (double *, int, int, char *)
- void write_int (int *, int, int, char *)
- int idamax (int, double *, int=1)
- void daxpy (int, double, double *, int, double *, int)
- void daxpyx (int, double, double *, int, double *, int)
- void dcopy (int, double *, int, double *, int)
- double ddot (int, double *, int, double *, int)
- double dnrm2 (int, double *, int=1)
- void dexopy (int, double *, double *, double *, int)
- void dfill (int, double *, double)
- double r2mach ()
- double norm2 (int, double *)
- double prod_esc (int, double *, double *)
- void dcopia (int, double *, double *)
- int indmax (int, double *)
- int indmin (int, double *)

6.10.1 Function Documentation

6.10.1.1 void daxpy (int , double , double * , int , double * , int)

6.10.1.2 void daxpyx (int , double , double * , int , double * , int)

6.10.1.3 void dcopia (int , double * , double *)

6.10.1.4 void dcopy (int , double * , int , double * , int)

6.10.1.5 double ddot (int , double * , int , double * , int)

6.10.1.6 void dexopy (int , double * , double * , double * , int)

6.10.1.7 void dfill (int , double * , double)

6.10.1.8 double dnrm2 (int , double * , int = 1)

6.10.1.9 int idamax (int , double * , int = 1)

6.10.1.10 int indmax (int , double *)

6.10.1.11 int indmin (int , double *)

6.10.1.12 double norm2 (int , double *)

6.10.1.13 double prod_esc (int , double * , double *)

6.10.1.14 double r2mach ( )

6.10.1.15 void write_double (double * , int , int , char *)

6.10.1.16 void write_int (int * , int , int , char *)

6.11 /home/jcastro2/intpoint/BlockIP/BlockIP/src_blockip-v2/pcg.C File Reference

```
#include <math.h>
#include <float.h>
#include "misc.h"
#include "pcg.h"
```

Functions

- int pcg (int n, int(∗atimes)(int, double ∗, double ∗, void ∗), int(∗msolve)(int, double ∗, double ∗, int(∗atimes)(int, double ∗, double ∗, void ∗), void ∗), double ∗x, double ∗b, double ∗r, double ∗z, double ∗p, int itmax, double eps, double ∗err, void ∗v, double ∗valpha, double ∗vbeta)

6.11.1 Function Documentation

6.11.1.1 int pcg (int *n,* int(∗)(int, double ∗, double ∗, void ∗) *atimes,* int(∗)(int, double ∗, double ∗, int(∗atimes)(int, double ∗, double ∗, void ∗), void ∗) *msolve,* double ∗ *x,* double ∗ *b,* double ∗ *r,* double ∗ *z,* double ∗ *p,* int *itmax,* double *eps,* double ∗ *err,* void ∗ *v,* double ∗ *valpha,* double ∗ *vbeta*)

6.12 /home/jcastro2/intpoint/BlockIP/BlockIP/src_blockip-v2/pcg.h File Reference

Functions

- int pcg (int n, int(∗atimes)(int, double ∗, double ∗, void ∗), int(∗msolve)(int, double ∗, double ∗, int(∗atimes)(int, double ∗, double ∗, void ∗), void ∗), double ∗x, double ∗b, double ∗r, double ∗z, double ∗p, int itmax, double eps, double ∗err, void ∗, double ∗valpha=0, double ∗vbeta=0)

6.12.1 Function Documentation

6.12.1.1 int pcg (int *n,* int(∗)(int, double ∗, double ∗, void ∗) *atimes,* int(∗)(int, double ∗, double ∗, int(∗atimes)(int, double ∗, double ∗, void ∗), void ∗) *msolve,* double ∗ *x,* double ∗ *b,* double ∗ *r,* double ∗ *z,* double ∗ *p,* int *itmax,* double *eps,* double ∗ *err,* void ∗ , double ∗ *valpha* = 0, double ∗ *vbeta* = 0)

6.13 /home/jcastro2/intpoint/BlockIP/BlockIP/src_blockip-v2/ritz_value.C File Reference

```
#include <stdio.h>
#include <math.h>
```

Macros

- #define DSTERF dsterf_

Functions

- void DSTERF (int ∗n, double ∗d, double ∗e, int ∗info)
- int ritz_value (int itcg, double ∗valpha, double ∗vbeta)

6.13.1 Macro Definition Documentation

6.13.1.1 #define DSTERF dsterf_

6.13.2 Function Documentation

6.13.2.1 void DSTERF (int * *n,* double * *d,* double * *e,* int * *info*)

6.13.2.2 int ritz_value (int *itcg,* double * *valpha,* double * *vbeta*)

Parameters

itcg	The order of the matrix
valpha	On entry, the itcg diagonal elements of the tridiagonal matrix. On exit, if info = 0, the eigenvalues in ascending order
vbeta	On entry, the (itcg-1) subdiagonal elements of the tridiagonal matrix. On exit, vbeta has been destroyed.

Returns

= 0: successful exit $<$ 0: if -i, the i-th argument had an illegal value $>$ 0: the algorithm failed to find all of the eigenvalues in a total of 30*itcg iterations; if i, then i elements of vbeta have not converged to zero.

Note

The user must guarantee valpha and vbeta have at least dimension itcg

6.14 /home/jcastro2/intpoint/BlockIP/BlockIP/src_blockip-v2/ritz_value.h File Reference

Functions

- int ritz_value (int itcg, double *valpha, double *vbeta)

6.14.1 Function Documentation

6.14.1.1 int ritz_value (int *itcg,* double * *valpha,* double * *vbeta*)

Parameters

itcg	The order of the matrix
valpha	On entry, the itcg diagonal elements of the tridiagonal matrix. On exit, if info = 0, the eigenvalues in ascending order
vbeta	On entry, the (itcg-1) subdiagonal elements of the tridiagonal matrix. On exit, vbeta has been destroyed.

Returns

= 0: successful exit $<$ 0: if -i, the i-th argument had an illegal value $>$ 0: the algorithm failed to find all of the eigenvalues in a total of 30*itcg iterations; if i, then i elements of vbeta have not converged to zero.

Note

The user must guarantee valpha and vbeta have at least dimension itcg

6.15 /home/jcastro2/intpoint/BlockIP/BlockIP/src_blockip-v2/SparseChol.C File Reference

```
#include <iostream>
#include <stdio.h>
#include <stdlib.h>
#include "SparseChol.h"
```

Macros

- #define ALLOC(n, type) (type *)malloc((n)*sizeof(type))
- #define REALLOC(ptr, n, type) (type *)realloc(((char *)ptr),(n)*sizeof(type))
- #define FREE(p) if (p) {free(p); (p)= NULL;}
- #define TRY_ALLOC(ptr, n, type) if (!(ptr= ALLOC((n),type))) {throw ExceptionBlockIP(OUT_OF_MEMORY, __FILE__, __LINE__);}
- #define TRY_REALLOC(ptr, n, type) if (!(ptr= REALLOC((ptr),(n),type))) {throw ExceptionBlockIP(OUT_OF_MEMORY, __FILE__, __LINE__);}
- #define FREE(p) if (p) {free(p); (p)= NULL;}

6.15.1 Macro Definition Documentation

6.15.1.1 #define ALLOC(*n, type*) (type *)malloc((n)*sizeof(type))

6.15.1.2 #define FREE(*p*) if (p) {free(p); (p)= NULL;}

6.15.1.3 #define FREE(*p*) if (p) {free(p); (p)= NULL;}

6.15.1.4 #define REALLOC(*ptr, n, type*) (type *)realloc(((char *)ptr),(n)*sizeof(type))

6.15.1.5 #define TRY_ALLOC(*ptr, n, type*) if (!(ptr= ALLOC((n),type))) {throw ExceptionBlockIP(OUT_OF_MEMORY, __FILE__, __LINE__);}

6.15.1.6 #define TRY_REALLOC(*ptr, n, type*) if (!(ptr= REALLOC((ptr),(n),type))) {throw ExceptionBlockIP(OUT_OF_MEMORY, __FILE__, __LINE__);}

6.16 /home/jcastro2/intpoint/BlockIP/BlockIP/src_blockip-v2/SparseChol.h File Reference

```
#include <iostream>
#include <stdio.h>
#include <stdlib.h>
#include "ExceptionBlockIP.h"
```

Classes

- class SparseChol

  Class for sparse Cholesky factorizations.
- struct SparseChol::SRC_DST_ARC

  Auxiliary struct for sorting network structure.

Enumerations

- enum CHOL_SOLVER { CHOLMOD, SPRSBLKLLT }

  Cholesky solver to be used.
- enum WHO_PERMUTES { CHOLESKY, USER_OF_CHOLESKY }

  Who is in charge of permuting constraints-related variables.
- enum TYPE_MATRIX {
  GENERAL, NETWORK, IDENTITY, DIAGONAL,
  IDTY_IDTY, DIAG_DIAG, GEN_SYM_UPTR }

  Matrix type; IDTY_IDTY= [I I] ; DIAG_DIAG= [D1 D2].
- enum TYPE_ORIENTATION { ORIENTED, NONORIENTED }

  Type of arc orientation for NETWORK matrices.
- enum NUMBERING { C_0, FORTRAN_1, NOT_COMPUTED }

  Type of numbering for arrays: either start at 0 or at 1. Needed for mixing C and Fortran code.

6.16.1 Enumeration Type Documentation

6.16.1.1 enum CHOL_SOLVER

Cholesky solver to be used.

Enumerator:

CHOLMOD

SPRSBLKLLT

6.16.1.2 enum NUMBERING

Type of numbering for arrays: either start at 0 or at 1. Needed for mixing C and Fortran code.

Enumerator:

C_0

FORTRAN_1

NOT_COMPUTED

6.16.1.3 enum TYPE_MATRIX

Matrix type; IDTY_IDTY= [I I] ; DIAG_DIAG= [D1 D2].

Enumerator:

GENERAL

NETWORK

IDENTITY

DIAGONAL

IDTY_IDTY

DIAG_DIAG

GEN_SYM_UPTR

6.16.1.4 enum TYPE_ORIENTATION

Type of arc orientation for NETWORK matrices.

Enumerator:

ORIENTED

NONORIENTED

6.16.1.5 enum WHO_PERMUTES

Who is in charge of permuting constraints-related variables.

Enumerator:

CHOLESKY

USER_OF_CHOLESKY

6.17 /home/jcastro2/intpoint/BlockIP/BlockIP/src_blockip-v2/SparseCholSprsblkllt.C File Reference

```
#include <stdio.h>
#include <stdlib.h>
#include <algorithm>
#include "SparseChol.h"
#include "sprsblkllt.h"
```

Macros

- #define ALLOC(n, type) (type *)malloc((n)*sizeof(type))
- #define REALLOC(ptr, n, type) (type *)realloc(((char *)ptr),(n)*sizeof(type))
- #define FREE(p) if (p) {free(p); (p)= NULL;}
- #define TRY_ALLOC(ptr, n, type) if (!(ptr= ALLOC((n),type))) {throw ExceptionBlockIP(OUT_OF_MEMORY, __FILE__, __LINE__);}
- #define TRY_REALLOC(ptr, n, type) if (!(ptr= REALLOC((ptr),(n),type))) {throw ExceptionBlockIP(OUT_OF_MEMORY, __FILE__, __LINE__);}
- #define ASSIGN_C(ix, fix, cx) if (ix<=fix) cx= icola[ix]

6.17.1 Macro Definition Documentation

6.17.1.1 #define ALLOC(*n, type*) (type *)malloc((n)*sizeof(type))

6.17.1.2 #define ASSIGN_C(*ix, fix, cx*) if (ix<=fix) cx= icola[ix]

6.17.1.3 #define FREE(*p*) if (p) {free(p); (p)= NULL;}

6.17.1.4 #define REALLOC(*ptr, n, type*) (type *)realloc(((char *)ptr),(n)*sizeof(type))

6.17.1.5 #define TRY_ALLOC(*ptr, n, type*) if (!(ptr= ALLOC((n),type))) {throw ExceptionBlockIP(OUT_OF_MEMORY, __FILE__, __LINE__);}

6.17.1.6 #define TRY_REALLOC(*ptr, n, type*) if (!(ptr= REALLOC((ptr),(n),type))) {throw ExceptionBlockIP(OUT_OF_MEMORY, __FILE__, __LINE__);}

6.18 /home/jcastro2/intpoint/BlockIP/BlockIP/src_blockip-v2/StdForm.C File Reference

```
#include <iostream>
#include "StdForm.h"
#include "ExceptionBlockIP.h"
#include <sstream>
```

6.19 /home/jcastro2/intpoint/BlockIP/BlockIP/src_blockip-v2/StdForm.h File Reference

Definition of StdForm.

```
#include <vector>
```

Classes

- class StdForm

  Class for perform conversions between standard form and original problem.
- struct StdForm::Transform

6.19.1 Detailed Description

Definition of StdForm.