Couenne

0.5

Generated by Doxygen 1.8.9.1

Thu Oct 8 2015 23:09:38

Contents

Chapter 1

Namespace Index

1.1 Namespace List

Here is a list of all documented namespaces with brief descriptions:

Chapter 2

Hierarchical Index

2.1 Class Hierarchy

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

- CoinMessages [external]
  - CbcMessage [external]
  - CglMessage [external]
  - ClpMessage [external]
  - CoinMessage [external]
  - LAP::LandPMessages [external]
  - LAP::LapMessages [external]
- CoinModelHash [external]
- CoinModelHash2 [external]
- CoinModelHashLink [external]
- CoinModelInfo2 [external]
- CoinModelLink [external]
- CoinModelLinkedList [external]
- CoinModelTriple [external]
- CoinMpsCardReader [external]
- CoinMpsIO [external]
- CoinOneMessage [external]
- CoinOtherFactorization [external]
  - CoinDenseFactorization [external]
  - CoinOslFactorization [external]
  - CoinSimpFactorization [external]
- CoinPackedMatrix [external]
- CoinPackedVectorBase [external]
  - CoinPackedVector [external]
  - CoinShallowPackedVector [external]
- CoinPair< S, T > [external]
- CoinParam [external]
  - CbcCbcParam [external]
  - CbcGenParam [external]
  - CbcOsiParam [external]
- CoinPrePostsolveMatrix [external]
  - CoinPostsolveMatrix [external]
  - CoinPresolveMatrix [external]
- CoinPresolveAction [external]
  - do_tighten_action [external]
  - doubleton_action [external]
  - drop_empty_cols_action [external]
  - drop_empty_rows_action [external]
  - drop_zero_coefficients_action [external]
  - dupcol_action [external]
  - duprow3_action [external]
  - duprow_action [external]
  - forcing_constraint_action [external]
  - gubrow_action [external]
  - implied_free_action [external]
  - isolated_constraint_action [external]
  - make_fixed_action [external]
  - remove_dual_action [external]
  - remove_fixed_action [external]
  - slack_doubleton_action [external]
  - slack_singleton_action [external]
  - subst_constraint_action [external]
  - tripleton_action [external]
  - twoxtwo_action [external]

SmartPtr< const Ipopt::Journalist >[external]
SmartPtr< const Ipopt::LowRankUpdateSymMatrixSpace >[external]
SmartPtr< const Ipopt::Matrix >[external]
SmartPtr< const Ipopt::MatrixSpace >[external]
SmartPtr< const Ipopt::MultiVectorMatrix >[external]
SmartPtr< const Ipopt::NLP >[external]
SmartPtr< const Ipopt::ScaledMatrixSpace >[external]
SmartPtr< const Ipopt::SymMatrix >[external]
SmartPtr< const Ipopt::SymMatrixSpace >[external]
SmartPtr< const Ipopt::SymScaledMatrixSpace >[external]
SmartPtr< const Ipopt::Vector >[external]
SmartPtr< const Ipopt::VectorSpace >[external]
Ipopt::SmartPtr< Couenne::SmartAsl >[external]
SmartPtr< Ipopt::AmplSuffixHandler >[external]
SmartPtr< Ipopt::AugSystemSolver >[external]
SmartPtr< Ipopt::BacktrackingLSAcceptor >[external]
SmartPtr< Ipopt::CompoundMatrix >[external]
SmartPtr< Ipopt::CompoundMatrixSpace >[external]
SmartPtr< Ipopt::CompoundSymMatrix >[external]
SmartPtr< Ipopt::CompoundSymMatrixSpace >[external]
SmartPtr< Ipopt::CompoundVector >[external]
SmartPtr< Ipopt::CompoundVectorSpace >[external]
SmartPtr< Ipopt::ConvergenceCheck >[external]
SmartPtr< Ipopt::DenseGenMatrix >[external]
SmartPtr< Ipopt::DenseSymMatrix >[external]
SmartPtr< Ipopt::DenseVector >[external]
SmartPtr< Ipopt::DiagMatrix >[external]
SmartPtr< Ipopt::DiagMatrixSpace >[external]
SmartPtr< Ipopt::EqMultiplierCalculator >[external]
SmartPtr< Ipopt::ExpandedMultiVectorMatrix >[external]
SmartPtr< Ipopt::ExpansionMatrix >[external]
SmartPtr< Ipopt::ExpansionMatrixSpace >[external]
SmartPtr< Ipopt::GenKKTSolverInterface >[external]
SmartPtr< Ipopt::HessianUpdater >[external]
SmartPtr< Ipopt::IdentityMatrixSpace >[external]
SmartPtr< Ipopt::InexactNewtonNormalStep >[external]
SmartPtr< Ipopt::InexactNormalStepCalculator >[external]
SmartPtr< Ipopt::InexactNormalTerminationTester >[external]
SmartPtr< Ipopt::InexactPDSolver >[external]
SmartPtr< Ipopt::IpoptAdditionalCq >[external]
SmartPtr< Ipopt::IpoptAdditionalData >[external]
SmartPtr< Ipopt::IpoptAlgorithm >[external]
SmartPtr< Ipopt::IpoptCalculatedQuantities >[external]
SmartPtr< Ipopt::IpoptData >[external]
SmartPtr< Ipopt::IpoptNLP >[external]
SmartPtr< Ipopt::IterateInitializer >[external]
SmartPtr< Ipopt::IteratesVectorSpace >[external]
SmartPtr< Ipopt::IterationOutput >[external]
SmartPtr< Ipopt::IterativeSolverTerminationTester >[external]
SmartPtr< Ipopt::Journal >[external]
Ipopt::SmartPtr< Ipopt::Journalist >[external]
Ipopt::SmartPtr< Ipopt::Journalist >[external]
SmartPtr< Ipopt::LineSearch >[external]
SmartPtr< Ipopt::Matrix >[external]

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 documented files with brief descriptions:

Chapter 5

Namespace Documentation

5.1 Couenne Namespace Reference

general include file for different compilers

Classes

- class AuxRelation

  Base class definition for relations between auxiliaries.
- class BiProdDivRel

  Identifies 5-tuple of the form.
- class compareSol

  class for comparing solutions (used in tabu list)
- struct compExpr

  Structure for comparing expressions.
- struct compNode

  structure for comparing nodes in the dependence graph
- class CouenneAggrProbing

  Cut Generator for aggressive BT; i.e., an aggressive probing.
- class CouenneAmplInterface
- class CouenneBab
- class CouenneBranchingObject

  "Spatial" branching object.
- class CouenneBTPerfIndicator
- class CouenneChooseStrong
- class CouenneChooseVariable

  Choose a variable for branching.
- class CouenneComplBranchingObject

  "Spatial" branching object for complementarity constraints.
- class CouenneComplObject

  OsiObject *for complementarity constraints* $x_1 x_2 \geq, \leq, = 0$.
- class CouenneConstraint

  Class to represent nonlinear constraints.
- class CouenneCrossConv

Cut Generator that uses relationships between auxiliaries.

- class CouenneCutGenerator

  Cut Generator for linear convexifications.
- class CouenneDisjCuts

  Cut Generator for linear convexifications.
- class CouenneExprMatrix
- class CouenneFeasPump

  An implementation of the Feasibility pump that uses linearization and **Ipopt** *to find the two sequences of points.*
- class CouenneFixPoint

  Cut Generator for FBBT fixpoint.
- class CouenneFPpool

  Pool of solutions.
- class CouenneFPsolution

  Class containing a solution with infeasibility evaluation.
- class CouenneInfo

  Bonmin class for passing info between components of branch-and-cuts.
- class CouenneInterface
- class CouenneIterativeRounding

  An iterative rounding heuristic, tailored for nonconvex MINLPs.
- class CouenneMINLPInterface

  This is class provides an Osi interface for a Mixed Integer Linear Program expressed as a TMINLP (so that we can use it for example as the continuous solver in Cbc).
- class CouenneMultiVarProbe
- class CouenneObject

  OsiObject *for auxiliary variables $w=f(x)$.*
- class CouenneObjective

  Objective function.
- class CouenneOrbitBranchingObj

  "Spatial" branching object.
- class CouenneOSInterface
- class CouenneProblem

  Class for MINLP problems with symbolic information.
- class CouennePSDcon

  Class to represent positive semidefinite constraints ////////////////.
- class CouenneRecordBestSol
- class CouenneScalar
- class CouenneSdpCuts

  These are cuts of the form.
- class CouenneSetup
- class CouenneSolverInterface

  Solver interface class with a pointer to a Couenne cut generator.
- class CouenneSOSBranchingObject
- class CouenneSOSObject
- class CouenneSparseBndVec
- class CouenneSparseMatrix

  Class for sparse Matrixs (used in modifying distances in FP)
- class CouenneSparseVector
- class CouenneThreeWayBranchObj

Spatial, three-way branching object.

- class CouenneTNLP

  Class for handling NLPs using CouenneProblem.
- class CouenneTwoImplied

  Cut Generator for implied bounds derived from pairs of linear (in)equalities.
- class CouenneUserInterface
- class CouenneVarObject

  OsiObject *for variables in a MINLP.*
- class CouenneVTObject

  OsiObject *for violation transfer on variables in a MINLP.*
- class CouExpr
- class DepGraph

  Dependence graph.
- class DepNode

  vertex of a dependence graph.
- class Domain

  Define a dynamic point+bounds, with a way to save and restore previous points+bounds through a LIFO structure.
- class DomainPoint

  Define a point in the solution space and the bounds around it.
- class exprAbs

  class for $|f(x)|$
- class exprAux

  Auxiliary variable.
- class exprBinProd

  class for $\prod_{i=1}^{n} f_i(x)$ *with* $f_i(x)$ *all binary*
- class exprCeil

  class ceiling, $\lceil f(x) \rceil$
- class exprClone

  expression clone (points to another expression)
- class exprConst

  constant-type operator
- class exprCopy
- class exprCos

  class cosine, $\cos f(x)$
- class exprDiv

  class for divisions, $\frac{f(x)}{g(x)}$
- class expression

  Expression base class.
- class exprEvenPow

  Power of an expression (binary operator) with even exponent, $f(x)^k$ *with* $k \in \mathbb{Z}$ *constant even.*
- class exprExp

  class for the exponential, $e^{f(x)}$
- class exprFloor

  class floor, $\lfloor f(x) \rfloor$
- class exprGroup

  class Group, with constant, linear and nonlinear terms: $a_0 + \sum_{i=1}^{n} a_i x_i$
- class ExprHess

expression matrices.

- class exprIf
- class exprInv

  class inverse: $1/f(x)$
- class exprIVar

  variable-type operator.
- class ExprJac

  Jacobian of the problem (computed through Couenne expression classes).
- class exprLBCos

  class to compute lower bound of a cosine based on the bounds of its arguments
- class exprLBDiv

  class to compute lower bound of a fraction based on the bounds of both numerator and denominator
- class exprLBMul

  class to compute lower bound of a product based on the bounds of both factors
- class exprLBQuad

  class to compute lower bound of a fraction based on the bounds of both numerator and denominator
- class exprLBSin

  class to compute lower bound of a sine based on the bounds on its arguments
- class exprLog

  class logarithm, $\log f(x)$
- class exprLowerBound

  These are bound expression classes.
- class exprMax
- class exprMin
- class exprMul

  class for multiplications, $\prod_{i=1}^{n} f_i(x)$
- class exprMultiLin

  another class for multiplications, $\prod_{i=1}^{n} f_i(x)$
- class exprNorm

  Class for p*-norms,* $||f(x)||_p = \left(\sum_{i=1}^{n} f_i(x)^p\right)^{\frac{1}{p}}$.
- class exprOddPow

  Power of an expression (binary operator), $f(x)^k$ *with* k *constant.*
- class exprOp

  general n-ary operator-type expression: requires argument list.
- class exprOpp

  class opposite, $-f(x)$
- class exprPow

  Power of an expression (binary operator), $f(x)^k$ *with* k *constant.*
- class exprPWLinear
- class exprQuad

  class exprQuad, with constant, linear and quadratic terms
- class exprSignPow

  Power of an expression (binary operator), $f(x)|f(x)|^{k-1}$ *with* $k \in \mathbb{R}$ *constant.*
- class exprSin

  class for $\sin f(x)$
- class exprStore

  storage class for previously evaluated expressions

- class exprSub

  class for subtraction, $f(x) - g(x)$
- class exprSum

  class sum, $\sum_{i=1}^{n} f_i(x)$
- class exprTrilinear

  class for multiplications
- class exprUBCos

  class to compute lower bound of a cosine based on the bounds of its arguments
- class exprUBDiv

  class to compute upper bound of a fraction based on the bounds of both numerator and denominator
- class exprUBMul

  class to compute upper bound of a product based on the bounds of both factors
- class exprUBQuad

  class to compute upper bound of a fraction based on the bounds of both numerator and denominator
- class exprUBSin

  class to compute lower bound of a sine based on the bounds on its arguments
- class exprUnary

  expression class for unary functions (sin, log, etc.)
- class exprUpperBound

  upper bound
- class exprVar

  variable-type operator
- class funtriplet
- class GlobalCutOff
- class InitHeuristic

  A heuristic that stores the initial solution of the NLP.
- class kpowertriplet
- class LinMap
- class MultiProdRel

  Identifies 5-ples of variables of the form.
- class NlpSolveHeuristic
- class powertriplet
- class PowRel

  Identifies 5-tuple of the form.
- class Qroot

  class that stores result of previous calls to rootQ into a map for faster access
- class quadElem
- class QuadMap
- class simpletriplet
- class SmartAsl
- class SumLogAuxRel

  Identifies 5-ples of variables of the form.
- class t_chg_bounds

  status of lower/upper bound of a variable, to be checked/modified in bound tightening

Typedefs

- typedef double CouNumber

  main number type in Couenne
- typedef CouNumber(∗ unary_function) (CouNumber)

  unary function, used in all exprUnary

Enumerations

- enum

  Define what kind of branching (two- or three-way) and where to start from: left, (center,) or right.
- enum nodeType

  type of a node in an expression tree
- enum linearity_type

  linearity of an expression, as returned by the method Linearity()
- enum pos

  position where the operator should be printed when printing the expression
- enum con_sign

  sign of constraint
- enum conv_type

  position and number of convexification cuts added for a lower convex (upper concave) envelope
- enum expr_type

  code returned by the method expression::code()
- enum convexity

  convexity type of an expression
- enum monotonicity

  monotonicity type of an expression
- enum dig_type

  type of digging when filling the dependence list
- enum cou_trig

  specify which trigonometric function is dealt with in trigEnvelope
- enum what_to_compare

  what term to compare: the sum of infeasibilities, the sum of numbers of infeasible terms, or the objective function
- enum Solver { Ipopt =0, FilterSQP, All }

  Solvers for solving nonlinear programs.

Functions

- CouNumber project (CouNumber a, CouNumber b, CouNumber c, CouNumber x0, CouNumber y0, CouNumber lb, CouNumber ub, int sign, CouNumber ∗xp=NULL, CouNumber ∗yp=NULL)

  Compute projection of point (x0, y0) on the segment defined by line ax + by + c <>= 0 (sign provided by parameter sign) and bounds [lb, ub] on x.
- CouNumber projectSeg (CouNumber x0, CouNumber y0, CouNumber x1, CouNumber y1, CouNumber x2, CouNumber y2, int sign, CouNumber ∗xp=NULL, CouNumber ∗yp=NULL)

  Compute projection of point (x0, y0) on the segment defined by two points (x1,y1), (x2, y2) – sign provided by parameter sign.
- void sparse2dense (int ncols, t_chg_bounds ∗chg_bds, int ∗&changed, int &nchanged)

translate sparse to dense vector (should be replaced)

- void CoinInvN (register const double *orig, register int n, register double *inverted)

  invert all contents
- void CoinCopyDisp (register const int *src, register int num, register int *dst, register int displacement)

  a CoinCopyN with a += on each element
- void draw_cuts (**OsiCuts** &, const CouenneCutGenerator *, int, expression *, expression *)

  allow to draw function within intervals and cuts introduced
- bool updateBound (register int sign, register CouNumber *dst, register CouNumber src)

  updates maximum violation.
- int compareExpr (const void *e0, const void *e1)

  independent comparison
- bool isInteger (CouNumber x)

  is this number integer?
- expression * getOriginal (expression *e)

  get original expression (can't make it an expression method as I need a non-const, what "this" would return)
- expression * Simplified (expression *complicated)

  Macro to return already simplified expression without having to do the if part every time simplify () is called.
- CouNumber zero_fun (CouNumber x)

  zero function (used by default by exprUnary)
- static CouNumber safeDiv (register CouNumber a, register CouNumber b, int sign)

  division that avoids NaN's and considers a sign when returning infinity
- CouNumber safeProd (register CouNumber a, register CouNumber b)

  product that avoids NaN's
- CouNumber trigNewton (CouNumber, CouNumber, CouNumber)

  common convexification method used by both cos and sin
- bool is_boundbox_regular (register CouNumber b1, register CouNumber b2)

  check if bounding box is suitable for a multiplication/division convexification constraint
- CouNumber inv (register CouNumber arg)

  the operator itself
- CouNumber oppInvSqr (register CouNumber x)

  derivative of inv (x)
- CouNumber inv_dblprime (register CouNumber x)

  inv_dblprime, second derivative of inv (x)
- void unifiedProdCuts (const CouenneCutGenerator *, **OsiCuts** &, int, CouNumber, CouNumber, CouNumber, int, CouNumber, CouNumber, CouNumber, int, CouNumber, CouNumber, CouNumber, t_chg_bounds *, enum expression::auxSign)

  unified convexification of products and divisions
- void upperEnvHull (const CouenneCutGenerator *cg, **OsiCuts** &cs, int xi, CouNumber x0, CouNumber xl, CouNumber xu, int yi, CouNumber y0, CouNumber yl, CouNumber yu, int wi, CouNumber w0, CouNumber wl, CouNumber wu)

  better cuts than those from unifiedProdCuts
- double * computeMulBrDist (const **OsiBranchingInformation** *info, int xi, int yi, int wi, int brind, double *brpt, int nPts=1)

  compute distance from future convexifications in set $\{(x, y, w) : w = xy\}$ *with x,y,w bounded.*
- CouNumber opp (register CouNumber arg)

  operator opp: returns the opposite of a number
- CouNumber safe_pow (CouNumber base, CouNumber exponent, bool signpower=false)

  compute power and check for integer-and-odd inverse exponent

- void addPowEnvelope (const CouenneCutGenerator *, **OsiCuts** &, int, int, CouNumber, CouNumber, CouNumber, CouNumber, CouNumber, int, bool=false)

  add upper/lower envelope to power in convex/concave areas
- CouNumber powNewton (CouNumber, CouNumber, unary_function, unary_function, unary_function)

  find proper tangent point to add deepest tangent cut
- CouNumber powNewton (CouNumber, CouNumber, funtriplet *)

  find proper tangent point to add deepest tangent cut
- CouNumber modulo (register CouNumber a, register CouNumber b)

  normalize angle within [0,b] (typically, pi or 2pi)
- CouNumber trigSelBranch (const CouenneObject *obj, const **OsiBranchingInformation** *info, expression *&var, double *&brpts, double *&brDist, int &way, enum cou_trig type)

  generalized procedure for both sine and cosine
- bool trigImpliedBound (enum cou_trig, int, int, CouNumber *, CouNumber *, t_chg_bounds *)

  generalized implied bound procedure for sine/cosine
- bool operator< (const CouenneFPsolution &one, const CouenneFPsolution &two)

  compare, base version
- CouNumber rootQ (int k)

  Find roots of polynomial $Q^k(x) = $ $\sum_{i=1}^{2k} ix^{i-1}$.

Variables

- const double large_bound = 1e9

  if |branching point| > this, change it
- const double Couenne_large_bound = 9.999e12

  used to declare LP unbounded
- const double maxNlpInf_0 = 1e-5

  A heuristic to call an NlpSolver if all CouenneObjects are close to be satisfied (for other integer objects, rounding is performed, if SOS's are not satisfied it does not run).

5.1.1 Detailed Description

general include file for different compilers

5.1.2 Enumeration Type Documentation

5.1.2.1 anonymous enum

Define what kind of branching (two- or three-way) and where to start from: left, (center,) or right.

The last is to help diversify branching through randomization, which may help when the same variable is branched upon in several points of the BB tree.

Definition at line 40 of file CouenneObject.hpp.

5.1.2.2 enum Couenne::pos

position where the operator should be printed when printing the expression

For instance, it is INSIDE for exprSum, exprMul, exprDiv, while it is PRE for exprLog, exprSin, exprExp...

Definition at line 30 of file CouenneTypes.hpp.

5.1.2.3 enum Couenne::Solver

Solvers for solving nonlinear programs.

Enumerator

EIpopt `Ipopt` interior point algorithm

EFilterSQP `filterSQP` Sequential Quadratic Programming algorithm

EAll Use all solvers.

Definition at line 47 of file CouenneMINLPInterface.hpp.

5.1.3 Function Documentation

5.1.3.1 CouNumber Couenne::project (CouNumber *a*, CouNumber *b*, CouNumber *c*, CouNumber *x0*, CouNumber *y0*, CouNumber *lb*, CouNumber *ub*, int *sign*, CouNumber * *xp* = `NULL`, CouNumber * *yp* = `NULL`)

Compute projection of point (x0, y0) on the segment defined by line ax + by + c <>= 0 (sign provided by parameter sign) and bounds [lb, ub] on x.

Return distance from segment, 0 if satisfied

5.1.3.2 CouNumber Couenne::projectSeg (CouNumber *x0*, CouNumber *y0*, CouNumber *x1*, CouNumber *y1*, CouNumber *x2*, CouNumber *y2*, int *sign*, CouNumber * *xp* = `NULL`, CouNumber * *yp* = `NULL`)

Compute projection of point (x0, y0) on the segment defined by two points (x1,y1), (x2, y2) – sign provided by parameter sign.

Return distance from segment, 0 if on it.

5.1.3.3 bool Couenne::updateBound (register int *sign*, register CouNumber * *dst*, register CouNumber *src*) `[inline]`

updates maximum violation.

Used with all impliedBound. Returns true if a bound has been modified, false otherwise

Definition at line 279 of file CouenneExpression.hpp.

5.1.3.4 double* Couenne::computeMulBrDist (const OsiBranchingInformation * *info*, int *xi*, int *yi*, int *wi*, int *brind*, double * *brpt*, int *nPts* = `1`)

compute distance from future convexifications in set $\{(x, y, w) : w = xy\}$ with x,y,w bounded.

Unified with exprDiv

5.1.3.5 CouNumber Couenne::rootQ (int *k*)

Find roots of polynomial $Q^k(x) = $ $\sum_{i=1}^{2k} ix^{i-1}$.

Used in convexification of powers with odd exponent

Chapter 6

Class Documentation

6.1 Couenne::AuxRelation Class Reference

Base class definition for relations between auxiliaries.

`#include <CouenneCrossConv.hpp>`

Inheritance diagram for Couenne::AuxRelation:

6.1.1 Detailed Description

Base class definition for relations between auxiliaries.

Definition at line 32 of file CouenneCrossConv.hpp.

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

- CouenneCrossConv.hpp

6.2 Couenne::BiProdDivRel Class Reference

Identifies 5-tuple of the form.

`#include <CouenneCrossConv.hpp>`

Inheritance diagram for Couenne::BiProdDivRel:

Collaboration diagram for Couenne::BiProdDivRel:

6.2.1 Detailed Description

Identifies 5-tuple of the form.

x_j := x_i / x_k x_p := x_i / x_q

x_l := x_j / x_p OR x_l := x_j x_k x_m := x_q / x_k x_m := x_p x_q

and generates, ONLY once, a cut

x_l = x_m (in both cases).

Definition at line 105 of file CouenneCrossConv.hpp.

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

- CouenneCrossConv.hpp

6.3 Couenne::CouenneExprMatrix::compare_pair_ind Struct Reference

6.3.1 Detailed Description

Definition at line 108 of file CouenneMatrix.hpp.

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

- CouenneMatrix.hpp

6.4 Couenne::CouenneSparseVector::compare_scalars Struct Reference

6.4.1 Detailed Description

Definition at line 70 of file CouenneMatrix.hpp.

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

- CouenneMatrix.hpp

6.5 Couenne::compareSol Class Reference

class for comparing solutions (used in tabu list)

`#include <CouenneFPpool.hpp>`

6.5.1 Detailed Description

class for comparing solutions (used in tabu list)

Definition at line 82 of file CouenneFPpool.hpp.

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

- CouenneFPpool.hpp

6.6 Couenne::compExpr Struct Reference

Structure for comparing expressions.

`#include <CouenneExprAux.hpp>`

6.6.1 Detailed Description

Structure for comparing expressions.

Used in compare() method for same-class expressions

Definition at line 210 of file CouenneExprAux.hpp.

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

- CouenneExprAux.hpp

6.7 Couenne::compNode Struct Reference

structure for comparing nodes in the dependence graph

`#include <CouenneDepGraph.hpp>`

Public Member Functions

- bool operator() (const DepNode *n0, const DepNode *n1) const
  structure for comparing nodes

6.7.1 Detailed Description

structure for comparing nodes in the dependence graph

Definition at line 25 of file CouenneDepGraph.hpp.

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

- CouenneDepGraph.hpp

6.8 Couenne::CouenneAggrProbing Class Reference

Cut Generator for aggressive BT; i.e., an aggressive probing.

`#include <CouenneAggrProbing.hpp>`

Inheritance diagram for Couenne::CouenneAggrProbing:

Collaboration diagram for Couenne::CouenneAggrProbing:

Public Member Functions

- CouenneAggrProbing (CouenneSetup *couenne, const **Ipopt::SmartPtr**< **Ipopt::OptionsList** > options)
  Constructor.
- CouenneAggrProbing (const CouenneAggrProbing &rhs)
  Copy constructor.
- ~CouenneAggrProbing ()
  Destructor.

- CouenneAggrProbing * clone () const

  Clone method (necessary for the abstract **CglCutGenerator** *class)*
- void generateCuts (const **OsiSolverInterface** &solver, **OsiCuts** &cuts, const **CglTreeInfo**=**CglTreeInfo**()) const

  The main **CglCutGenerator***; not implemented yet.*
- double probeVariable (int index, bool probeLower)

  Probe one variable (try to tigthen the lower or the upper bound, depending on the value of the second argument), so that we can generate the corresponding column cut.
- double probeVariable2 (int index, bool lower)

  Alternative probing algorithm.
- void setMaxTime (double value)

  Set/get maximum time to probe one variable.
- void setMaxFailedSteps (int value)

  Set/get maximum number of failed steps.
- void setMaxNodes (int value)

  Set/get maximum number of nodes to probe one variable.
- void setRestoreCutoff (bool value)

  Set/get restoreCutoff parameter (should we restore the initial cutoff value after each probing run?)

Static Public Member Functions

- static void registerOptions (**Ipopt::SmartPtr**< Bonmin::RegisteredOptions > roptions)

  Add list of options to be read from file.

Protected Attributes

- CouenneSetup * couenne_

  Pointer to the CouenneProblem representation.
- int numCols_

  Number of columns (want to have this handy)
- double maxTime_

  Maximum time to probe one variable.
- int maxFailedSteps_

  Maximum number of failed iterations.
- int maxNodes_

  Maximum number of nodes in probing.
- bool restoreCutoff_

  Restore initial cutoff (value and solution)?
- double initCutoff_

  Initial cutoff.

6.8.1 Detailed Description

Cut Generator for aggressive BT; i.e., an aggressive probing.

This probing strategy is very expensive and was initially developed to be run in parallel; hence, the user can choose to probe just a particular variable, without adding this cut generator to the list of cut generators normally employed by Couenne. However, it can also be used in the standard way; in that case, it chooses automatically the variables to

probe (in a very naive way, for the moment). TODO: Implement some way to automatically choose the variables TODO: Implement the generateCuts method, for use in Branch-and-Bound

Definition at line 37 of file CouenneAggrProbing.hpp.

6.8.2 Member Function Documentation

6.8.2.1 double Couenne::CouenneAggrProbing::probeVariable (int *index,* bool *probeLower*)

Probe one variable (try to tigthen the lower or the upper bound, depending on the value of the second argument), so that we can generate the corresponding column cut.

This runs the main algorithm. It returns the new bound (equal to the initial one if we could not tigthen)

6.8.2.2 double Couenne::CouenneAggrProbing::probeVariable2 (int *index,* bool *lower*)

Alternative probing algorithm.

This one does not work yet! Do not use, will probably segfault.

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

- CouenneAggrProbing.hpp

6.9 Couenne::CouenneAmplInterface Class Reference

Inheritance diagram for Couenne::CouenneAmplInterface:

Collaboration diagram for Couenne::CouenneAmplInterface:

Public Member Functions

- CouenneProblem * getCouenneProblem ()
  Should return the problem to solve in algebraic form.
- **Ipopt::SmartPtr**< Bonmin::TMINLP > getTMINLP ()
  Should return the problem to solve as TMINLP.
- bool writeSolution (Bonmin::Bab &bab)
  Called after B&B finished.

6.9.1 Detailed Description

Definition at line 26 of file CouenneAmplInterface.hpp.

6.9.2 Member Function Documentation

6.9.2.1 CouenneProblem* Couenne::CouenneAmplInterface::getCouenneProblem ( ) [virtual]

Should return the problem to solve in algebraic form.

NOTE: Couenne is (currently) going to modify this problem!

Implements Couenne::CouenneUserInterface.

6.9.2.2 bool Couenne::CouenneAmplInterface::writeSolution (Bonmin::Bab & *bab*) [virtual]

Called after B&B finished.

Should write solution information.

Reimplemented from Couenne::CouenneUserInterface.

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

- CouenneAmplInterface.hpp

6.10 Couenne::CouenneBab Class Reference

Inheritance diagram for Couenne::CouenneBab:

Collaboration diagram for Couenne::CouenneBab:

Public Member Functions

- CouenneBab ()

  Constructor.
- virtual ~CouenneBab ()

  Destructor.
- virtual void branchAndBound (Bonmin::BabSetupBase &s)

  Carry out branch and bound.
- const double * bestSolution () const

  Get the best solution known to the problem (is optimal if MipStatus is FeasibleOptimal).
- double bestObj () const

  Return objective value of the bestSolution.
- double bestBound ()

  return the best known lower bound on the objective value

6.10.1 Detailed Description

Definition at line 21 of file CouenneBab.hpp.

6.10.2 Member Function Documentation

6.10.2.1 const double* Couenne::CouenneBab::bestSolution ( ) const

Get the best solution known to the problem (is optimal if MipStatus is FeasibleOptimal).

If no solution is known returns NULL.

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

- CouenneBab.hpp

6.11 Couenne::CouenneBranchingObject Class Reference

"Spatial" branching object.

`#include <CouenneBranchingObject.hpp>`

Inheritance diagram for Couenne::CouenneBranchingObject:

Collaboration diagram for Couenne::CouenneBranchingObject:

Public Member Functions

- CouenneBranchingObject (**OsiSolverInterface** ∗solver, const **OsiObject** ∗**originalObject**, **JnlstPtr** jnlst, CouenneCutGenerator ∗c, CouenneProblem ∗p, expression ∗var, int **way**, CouNumber brpoint, bool doFB↩ BT, bool doConvCuts)

  Constructor.
- CouenneBranchingObject (const CouenneBranchingObject &src)

  Copy constructor.
- virtual **OsiBranchingObject** ∗ clone () const

  cloning method
- virtual double branch (**OsiSolverInterface** ∗solver=NULL)

  Execute the actions required to branch, as specified by the current state of the branching object, and advance the object's state.
- virtual bool boundBranch () const

  does this branching object only change variable bounds?
- void setSimulate (bool s)

  set simulate_ field below
- expression ∗ variable ()

  return branching variable
- void branchCore (**OsiSolverInterface** ∗, int, int, bool, double, t_chg_bounds ∗&)

  Perform branching step.

Protected Attributes

- CouenneCutGenerator ∗ cutGen_

  Pointer to CouenneCutGenerator (if any); if not NULL, allows to do extra cut generation during branching.
- CouenneProblem ∗ problem_

  Pointer to CouenneProblem (necessary to allow FBBT)
- expression ∗ variable_

  The index of the variable this branching object refers to.
- **JnlstPtr** jnlst_

  SmartPointer to the Journalist.
- bool doFBBT_

  shall we do Feasibility based Bound Tightening (FBBT) at branching?
- bool doConvCuts_

  shall we add convexification cuts at branching?
- double downEstimate_

  down branch estimate (done at selectBranch with reduced costs)
- double upEstimate_

up branch estimate

- bool simulate_

  are we currently in strong branching?

6.11.1 Detailed Description

"Spatial" branching object.

Branching can also be performed on continuous variables.

Definition at line 37 of file CouenneBranchingObject.hpp.

6.11.2 Member Function Documentation

6.11.2.1 virtual double Couenne::CouenneBranchingObject::branch (OsiSolverInterface * *solver* = NULL) [virtual]

Execute the actions required to branch, as specified by the current state of the branching object, and advance the object's state.

Returns change in guessed objective on next branch

Implements **OsiTwoWayBranchingObject**.

Reimplemented in Couenne::CouenneOrbitBranchingObj, and Couenne::CouenneComplBranchingObject.

6.11.3 Member Data Documentation

6.11.3.1 expression* Couenne::CouenneBranchingObject::variable_ [protected]

The index of the variable this branching object refers to.

If the corresponding CouenneObject was created on w=f(x,y), it is either x or y, chosen previously with a call to getFix↩ Var() expression *reference_;

Definition at line 111 of file CouenneBranchingObject.hpp.

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

- CouenneBranchingObject.hpp

6.12 Couenne::CouenneBTPerfIndicator Class Reference

Collaboration diagram for Couenne::CouenneBTPerfIndicator:

Public Member Functions

- CouenneBTPerfIndicator (CouenneProblem *p, const std::string &name)

  Should stats be printed at the end? Copied from problem_ -> Jnlst () -> ProduceOutput (ERROR, BOUNDTIGHTENING)

- void addToTimer (double time) const

  add to timer

Protected Attributes

- double nFixed_

  Whose performance is this?
- double boundRatio_

  number of fixed variables
- double shrunkInf_

  average bound width shrinkage
- double shrunkDoubleInf_

  average # bounds that went from infinite to finite (counts twice if [-inf,inf] to [a,b]
- double nProvedInfeas_

  average # bounds that went from doubly infinite to infinite
- double weightSum_

  average # proofs of infeasibility
- double * oldLB_

  total weight (used to give an average indicator at the end of Couenne)
- double * oldUB_

  old lower bounds (initial, i.e. before BT)
- double totalTime_

  old upper bounds
- int nRuns_

  CPU time spent on this.
- CouenneProblem * problem_

  number of runs
- bool stats_

  Couenne problem info.

6.12.1 Detailed Description

Definition at line 23 of file CouenneBTPerfIndicator.hpp.

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

- CouenneBTPerfIndicator.hpp

6.13 Couenne::CouenneChooseStrong Class Reference

Inheritance diagram for Couenne::CouenneChooseStrong:

Collaboration diagram for Couenne::CouenneChooseStrong:

Public Member Functions

- CouenneChooseStrong (Bonmin::BabSetupBase &b, CouenneProblem *problem, **JnlstPtr** jnlst)

  Constructor from solver (so we can set up arrays etc)
- CouenneChooseStrong (const CouenneChooseStrong &)

  Copy constructor.

- CouenneChooseStrong & operator= (const CouenneChooseStrong &rhs)

  Assignment operator.
- virtual **OsiChooseVariable** * clone () const

  Clone.
- virtual ~CouenneChooseStrong ()

  Destructor.
- virtual int setupList (**OsiBranchingInformation** *info, bool initialize)

  Sets up strong list and clears all if initialize is true.
- virtual int doStrongBranching (**OsiSolverInterface** *OsiSolver, **OsiBranchingInformation** *info, int numberTo↩ Do, int returnCriterion)

  This is a utility function which does strong branching on a list of objects and stores the results in OsiHotInfo.objects.
- virtual bool feasibleSolution (const **OsiBranchingInformation** *info, const double *solution, int numberObjects, const **OsiObject** **objects)

  Returns true if solution looks feasible against given objects.
- virtual int chooseVariable (**OsiSolverInterface** *solver, **OsiBranchingInformation** *info, bool fixVariables)

  choose object to branch based on earlier setup

Static Public Member Functions

- static void registerOptions (**Ipopt::SmartPtr**< Bonmin::RegisteredOptions > roptions)

  Add list of options to be read from file.

Protected Member Functions

- int simulateBranch (**OsiObject** *Object, **OsiBranchingInformation** *info, **OsiBranchingObject** *branch, **Osi**↩ **SolverInterface** *solver, Bonmin::HotInfo *result, int direction)

  does one side of the branching

Protected Attributes

- CouenneProblem * problem_

  Pointer to the associated MINLP problem.
- bool pseudoUpdateLP_

  should we update the pseudocost multiplier with the distance between the LP point and the solution of the resulting branches' LPs? If so, this only happens in strong branching
- bool estimateProduct_

  Normally, a convex combination of the min/max lower bounds' estimates is taken to select a branching variable, as in the original definition of strong branching.
- **JnlstPtr** jnlst_

  pointer to journalist for detailed information
- double branchtime_

  total time spent in strong branching

6.13.1 Detailed Description

Definition at line 23 of file CouenneChooseStrong.hpp.

6.13.2 Member Function Documentation

6.13.2.1 virtual int Couenne::CouenneChooseStrong::setupList (OsiBranchingInformation * *info,* bool *initialize*) [virtual]

Sets up strong list and clears all if initialize is true.

Returns number of infeasibilities.

6.13.2.2 virtual int Couenne::CouenneChooseStrong::doStrongBranching (OsiSolverInterface * *OsiSolver,* OsiBranchingInformation * *info,* int *numberToDo,* int *returnCriterion*) [virtual]

This is a utility function which does strong branching on a list of objects and stores the results in OsiHotInfo.objects.

On entry the object sequence is stored in the **OsiHotInfo** object and maybe more. It returns - -1 - one branch was infeasible both ways 0 - all inspected - nothing can be fixed 1 - all inspected - some can be fixed (returnCriterion==0) 2 - may be returning early - one can be fixed (last one done) (returnCriterion==1) 3 - returning because max time

6.13.3 Member Data Documentation

6.13.3.1 bool Couenne::CouenneChooseStrong::estimateProduct_ [protected]

Normally, a convex combination of the min/max lower bounds' estimates is taken to select a branching variable, as in the original definition of strong branching.

If this option is set to true, their product is taken instead:

(1e-6+min) * max

where the 1e-6 is used to ensure that even those with one subproblem with no improvement are compared.

Definition at line 135 of file CouenneChooseStrong.hpp.

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

- CouenneChooseStrong.hpp

6.14 Couenne::CouenneChooseVariable Class Reference

Choose a variable for branching.

`#include <CouenneChooseVariable.hpp>`

Inheritance diagram for Couenne::CouenneChooseVariable:

Collaboration diagram for Couenne::CouenneChooseVariable:

Public Member Functions

- CouenneChooseVariable ()

  Default Constructor.
- CouenneChooseVariable (const **OsiSolverInterface** *, CouenneProblem *, **JnlstPtr** jnlst)

  Constructor from solver (so we can set up arrays etc)
- CouenneChooseVariable (const CouenneChooseVariable &)

Copy constructor.

- CouenneChooseVariable & operator= (const CouenneChooseVariable &)

  Assignment operator.

- virtual **OsiChooseVariable** * clone () const

  Clone.

- virtual ~CouenneChooseVariable ()

  Destructor.

- virtual int setupList (**OsiBranchingInformation** *, bool)

  Sets up strong list and clears all if initialize is true.

- virtual bool feasibleSolution (const **OsiBranchingInformation** *info, const double *solution, int numberObjects, const **OsiObject** **objects)

  Returns true if solution looks feasible against given objects.

Static Public Member Functions

- static void registerOptions (**Ipopt::SmartPtr**< Bonmin::RegisteredOptions > roptions)

  Add list of options to be read from file.

Protected Attributes

- CouenneProblem * problem_

  Pointer to the associated MINLP problem.

- **JnlstPtr** jnlst_

  journalist for detailed debug information

6.14.1 Detailed Description

Choose a variable for branching.

Definition at line 27 of file CouenneChooseVariable.hpp.

6.14.2 Member Function Documentation

6.14.2.1 virtual int Couenne::CouenneChooseVariable::setupList (**OsiBranchingInformation** *, **bool**) [virtual]

Sets up strong list and clears all if initialize is true.

Returns number of infeasibilities. If returns -1 then has worked out node is infeasible!

Reimplemented from **OsiChooseVariable**.

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

- CouenneChooseVariable.hpp

6.15 Couenne::CouenneComplBranchingObject Class Reference

"Spatial" branching object for complementarity constraints.

`#include <CouenneComplBranchingObject.hpp>`

Inheritance diagram for Couenne::CouenneComplBranchingObject:

Collaboration diagram for Couenne::CouenneComplBranchingObject:

Public Member Functions

- CouenneComplBranchingObject (**OsiSolverInterface** ∗solver, const **OsiObject** ∗**originalObject**, **JnlstPtr** jnlst, CouenneCutGenerator ∗c, CouenneProblem ∗p, expression ∗var, expression ∗var2, int **way**, CouNumber brpoint, bool doFBBT, bool doConvCuts, int sign)

  Constructor.
- CouenneComplBranchingObject (const CouenneComplBranchingObject &src)

  Copy constructor.
- virtual **OsiBranchingObject** ∗ clone () const

  cloning method
- virtual double branch (**OsiSolverInterface** ∗solver=NULL)

  Execute the actions required to branch, as specified by the current state of the branching object, and advance the object's state.

Protected Attributes

- expression ∗ variable2_

  use CouenneBranchingObject::variable_ as the first variable to set to 0, and this one as the second
- int sign_

  -1 if object is for xi ∗ xj <= 0 +1 if object is for xi ∗ xj <= 0 0 if object is for xi ∗ xj = 0 (classical)

6.15.1 Detailed Description

"Spatial" branching object for complementarity constraints.

Branching on such an object x_1 x_2 = 0 is performed by setting either x_1=0 or x_2=0

Definition at line 24 of file CouenneComplBranchingObject.hpp.

6.15.2 Member Function Documentation

6.15.2.1 virtual double Couenne::CouenneComplBranchingObject::branch (**OsiSolverInterface** ∗ *solver* = `NULL`) `[virtual]`

Execute the actions required to branch, as specified by the current state of the branching object, and advance the object's state.

Returns change in guessed objective on next branch

Reimplemented from Couenne::CouenneBranchingObject.

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

- CouenneComplBranchingObject.hpp

6.16 Couenne::CouenneComplObject Class Reference

OsiObject for complementarity constraints $x_1 x_2 \geq, \leq, = 0$.

`#include <CouenneComplObject.hpp>`

Inheritance diagram for Couenne::CouenneComplObject:

Collaboration diagram for Couenne::CouenneComplObject:

Public Member Functions

- CouenneComplObject (CouenneCutGenerator *c, CouenneProblem *p, exprVar *ref, Bonmin::BabSetupBase *base, **JnlstPtr** jnlst, int sign)
  Constructor with information for branching point selection strategy.
- CouenneComplObject (exprVar *ref, Bonmin::BabSetupBase *base, **JnlstPtr** jnlst, int sign)
  Constructor with lesser information, used for infeasibility only.
- ~CouenneComplObject ()
  Destructor.
- CouenneComplObject (const CouenneComplObject &src)
  Copy constructor.
- virtual CouenneObject * clone () const
  Cloning method.
- virtual double infeasibility (const **OsiBranchingInformation** *info, int &way) const
  compute infeasibility of this variable, |w - f(x)| (where w is the auxiliary variable defined as w = f(x)
- virtual double checkInfeasibility (const **OsiBranchingInformation** *info) const
  compute infeasibility of this variable, |w - f(x)|, where w is the auxiliary variable defined as w = f(x)
- virtual **OsiBranchingObject** * createBranch (**OsiSolverInterface** *, const **OsiBranchingInformation** *, int way) const
  create CouenneBranchingObject or CouenneThreeWayBranchObj based on this object

Additional Inherited Members

6.16.1 Detailed Description

OsiObject for complementarity constraints $x_1 x_2 \geq, \leq, = 0$.

Associated with two variables x_1 and x_2, branches with either $x_1 = 0$ or $x_2 = 0$

Definition at line 22 of file CouenneComplObject.hpp.

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

- CouenneComplObject.hpp

6.17 Couenne::CouenneConstraint Class Reference

Class to represent nonlinear constraints.

`#include <CouenneProblemElem.hpp>`

Inheritance diagram for Couenne::CouenneConstraint:

Collaboration diagram for Couenne::CouenneConstraint:

Public Member Functions

- CouenneConstraint (expression *body=NULL, expression *lb=NULL, expression *ub=NULL)

  Constructor.
- virtual ~CouenneConstraint ()

  Destructor.
- CouenneConstraint (const CouenneConstraint &c, Domain *d=NULL)

  Copy constructor.
- virtual CouenneConstraint * clone (Domain *d=NULL) const

  Cloning method.
- virtual expression * Lb () const

  Expression of lower bound.
- virtual expression * Ub () const

  Expression of upper bound.
- virtual expression * Body () const

  Expression of body of constraint.
- virtual expression * Body (expression *newBody)

  Set body of constraint.
- virtual exprAux * standardize (CouenneProblem *)

  decompose body of constraint through auxiliary variables
- virtual void print (std::ostream &=std::cout)

  print constraint

Protected Attributes

- expression * body_

  Body of constraint.
- expression * lb_

  Lower bound (expression)
- expression * ub_

  Upper bound (expression)

6.17.1 Detailed Description

Class to represent nonlinear constraints.

It consists of an expression as the body and two range expressions as lower- and upper bounds.

A general constraint is defined as lb_ <= body_ <= ub_, where all three components are expressions, depending on variables, auxiliaries and bounds. If the constraint is 2 <= exp (x1+x2) <= 4, then:

body_ = exp (x1+x2), that is,

new exprExp (new exprSum (new exprVar (1), new exprVar (2))

while lb_ = new exprConst (2.) and ub_ = new exprConst (4.).

Definition at line 39 of file CouenneProblemElem.hpp.

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

- CouenneProblemElem.hpp

6.18 Couenne::CouenneCrossConv Class Reference

Cut Generator that uses relationships between auxiliaries.

`#include <CouenneCrossConv.hpp>`

Inheritance diagram for Couenne::CouenneCrossConv:

Collaboration diagram for Couenne::CouenneCrossConv:

Public Member Functions

- CouenneCrossConv (CouenneProblem *, **JnlstPtr**, const **Ipopt::SmartPtr**< **Ipopt::OptionsList** >)

  constructor
- CouenneCrossConv (const CouenneCrossConv &)

  copy constructor
- virtual ~CouenneCrossConv ()

  destructor
- virtual CouenneCrossConv * clone () const

  clone method (necessary for the abstract **CglCutGenerator** *class)*
- virtual void generateCuts (const **OsiSolverInterface** &, **OsiCuts** &, const **CglTreeInfo**=**CglTreeInfo**()) const

  the main **CglCutGenerator**
- virtual void setup ()

  Set up data structure to detect redundancies.

Static Public Member Functions

- static void registerOptions (**Ipopt::SmartPtr**< Bonmin::RegisteredOptions > roptions)

  Add list of options to be read from file.

Protected Attributes

- **JnlstPtr** jnlst_

  Journalist.
- CouenneProblem * problem_

  pointer to the CouenneProblem representation

6.18.1 Detailed Description

Cut Generator that uses relationships between auxiliaries.

Definition at line 139 of file CouenneCrossConv.hpp.

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

- CouenneCrossConv.hpp

6.19 Couenne::CouenneCutGenerator Class Reference

Cut Generator for linear convexifications.

`#include <CouenneCutGenerator.hpp>`

Inheritance diagram for Couenne::CouenneCutGenerator:

Collaboration diagram for Couenne::CouenneCutGenerator:

Public Member Functions

- CouenneCutGenerator (Bonmin::OsiTMINLPInterface *=NULL, Bonmin::BabSetupBase *base=NULL, CouenneProblem *=NULL, struct ASL *=NULL)
  constructor
- CouenneCutGenerator (const CouenneCutGenerator &)
  copy constructor
- ~CouenneCutGenerator ()
  destructor
- CouenneCutGenerator * clone () const
  clone method (necessary for the abstract **CglCutGenerator** *class)*
- CouenneProblem * Problem () const
  return pointer to symbolic problem
- void setProblem (CouenneProblem *p)
  return pointer to symbolic problem
- int getnvars () const
  total number of variables (original + auxiliary)
- bool isFirst () const
  has generateCuts been called yet?
- bool addViolated () const
  should we add the violated cuts only (true), or all of them (false)?
- enum conv_type ConvType () const
  get convexification type (see CouenneTypes.h)
- int nSamples () const
  get number of convexification samples
- void generateCuts (const **OsiSolverInterface** &, **OsiCuts** &, const **CglTreeInfo**=**CglTreeInfo**()) const
  the main **CglCutGenerator**
- int createCut (**OsiCuts** &, CouNumber, CouNumber, int, CouNumber, int=-1, CouNumber=0., int=-1, CouNumber=0., bool=false) const
  create cut and check violation. Insert and return status
- int createCut (**OsiCuts** &, CouNumber, int, int, CouNumber, int=-1, CouNumber=0., int=-1, CouNumber=0., bool=false) const
  create cut and check violation. Other version with only one bound
- void addEnvelope (**OsiCuts** &, int, unary_function, unary_function, int, int, CouNumber, CouNumber, CouNumber, t_chg_bounds *=NULL, bool=false) const
  Add general linear envelope to convex function, given its variables' indices, the (univariate) function and its first derivative.
- void addEnvelope (**OsiCuts** &, int, funtriplet *, int, int, CouNumber, CouNumber, CouNumber, t_chg_bounds *=NULL, bool=false) const
  Add general linear envelope to convex function, given its variables' indices, the (univariate) function and its first derivative.
- int addSegment (**OsiCuts** &, int, int, CouNumber, CouNumber, CouNumber, CouNumber, int) const

Add half-plane through (x1,y1) and (x2,y2) – resp.

- int addTangent (**OsiCuts** &, int, int, CouNumber, CouNumber, CouNumber, int) const

  add tangent at given poing (x,w) with given slope

- void setBabPtr (Bonmin::Bab *p)

  Method to set the Bab pointer.

- void getStats (int &nrc, int &ntc, double &st)

  Get statistics.

- bool & infeasNode () const

  Allow to get and set the infeasNode_ flag (used only in generateCuts())

- void genRowCuts (const **OsiSolverInterface** &, **OsiCuts** &cs, int, int *, t_chg_bounds *=NULL) const

  generate OsiRowCuts for current convexification

- void genColCuts (const **OsiSolverInterface** &, **OsiCuts** &, int, int *) const

  generate OsiColCuts for improved (implied and propagated) bounds

- void printLineInfo () const

  print node, depth, LB/UB/LP info

- **ConstJnlstPtr** Jnlst () const

  Provide Journalist.

- double & rootTime ()

  Time spent at root node.

- bool check_lp () const

  return check_lp flag (used in CouenneSolverInterface)

- bool enableLpImpliedBounds () const

  *returns value of enable_lp_implied_bounds_*

Static Public Member Functions

- static void registerOptions (**Ipopt::SmartPtr**< Bonmin::RegisteredOptions > roptions)

  Add list of options to be read from file.

Protected Attributes

- bool firstcall_

  True if no convexification cuts have been generated yet for this problem.

- bool addviolated_

  True if we should add the violated cuts only, false if all of them should be added.

- enum conv_type convtype_

  what kind of sampling should be performed?

- int nSamples_

  how many cuts should be added for each function?

- CouenneProblem * problem_

  pointer to symbolic repr. of constraint, variables, and bounds

- int nrootcuts_

  number of cuts generated at the first call

- int ntotalcuts_

  total number of cuts generated

- double septime_

separation time (includes generation of problem)

- double objValue_

  Record obj value at final point of CouenneConv.

- Bonmin::OsiTMINLPInterface * nlp_

  nonlinear solver interface as used within Bonmin (used at first Couenne pass of each b&b node

- Bonmin::Bab * BabPtr_

  pointer to the Bab object (used to retrieve the current primal bound through bestObj())

- bool infeasNode_

  signal infeasibility of current node (found through bound tightening)

- **JnlstPtr** jnlst_

  SmartPointer to the Journalist.

- double rootTime_

  Time spent at the root node.

- bool check_lp_

  Check all generated LPs through an independent call to **OsiClpSolverInterface::initialSolve()**

- bool enable_lp_implied_bounds_

  Take advantage of **OsiClpSolverInterface::tightenBounds** *(), known to have caused some problems some time ago.*

- int lastPrintLine

  Running count of printed info lines.

6.19.1 Detailed Description

Cut Generator for linear convexifications.

Definition at line 49 of file CouenneCutGenerator.hpp.

6.19.2 Member Function Documentation

6.19.2.1 int Couenne::CouenneCutGenerator::addSegment (OsiCuts & , int , int , CouNumber , CouNumber , CouNumber , CouNumber , int) const

Add half-plane through (x1,y1) and (x2,y2) – resp.

4th, 5th, 6th, and 7th argument

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

- CouenneCutGenerator.hpp

6.20 Couenne::CouenneDisjCuts Class Reference

Cut Generator for linear convexifications.

`#include <CouenneDisjCuts.hpp>`

Inheritance diagram for Couenne::CouenneDisjCuts:

Collaboration diagram for Couenne::CouenneDisjCuts:

Public Member Functions

- CouenneDisjCuts (Bonmin::OsiTMINLPInterface *minlp=NULL, Bonmin::BabSetupBase *base=NUL↩ L, CouenneCutGenerator *cg=NULL, **OsiChooseVariable** *bcv=NULL, bool is_strong=false, **JnlstPtr** journalist=NULL, const **Ipopt::SmartPtr**< **Ipopt::OptionsList** > options=NULL)

  constructor
- CouenneDisjCuts (const CouenneDisjCuts &)

  copy constructor
- ~CouenneDisjCuts ()

  destructor
- CouenneDisjCuts * clone () const

  clone method (necessary for the abstract **CglCutGenerator** *class)*
- CouenneCutGenerator * couenneCG () const

  return pointer to symbolic problem
- void generateCuts (const **OsiSolverInterface** &, **OsiCuts** &, const **CglTreeInfo**=**CglTreeInfo**()) const

  the main **CglCutGenerator**
- **ConstJnlstPtr** Jnlst () const

  Provide Journalist.
- int getDisjunctions (std::vector< std::pair< **OsiCuts** *, **OsiCuts** * > > &disjunctions, **OsiSolverInterface** &si, **OsiCuts** &cs, const **CglTreeInfo** &info) const

  get all disjunctions
- int separateWithDisjunction (**OsiCuts** *cuts, **OsiSolverInterface** &si, **OsiCuts** &cs, const **CglTreeInfo** &info) const

  separate couenne cuts on both sides of single disjunction
- int generateDisjCuts (std::vector< std::pair< **OsiCuts** *, **OsiCuts** * > > &disjs, **OsiSolverInterface** &si, **Osi**↩ **Cuts** &cs, const **CglTreeInfo** &info) const

  generate one disjunctive cut from one CGLP
- int checkDisjSide (**OsiSolverInterface** &si, **OsiCuts** *cuts) const

  check if (column!) cuts compatible with solver interface
- int getBoxUnion (**OsiSolverInterface** &si, **OsiCuts** *left, **OsiCuts** *right, **CoinPackedVector** &lower, **Coin**↩ **PackedVector** &upper) const

  compute smallest box containing both left and right boxes.

Static Public Member Functions

- static void registerOptions (**Ipopt::SmartPtr**< Bonmin::RegisteredOptions > roptions)

  Add list of options to be read from file.

Protected Member Functions

- **OsiCuts** * getSingleDisjunction (**OsiSolverInterface** &si) const

  create single osicolcut disjunction
- void mergeBoxes (int dir, **CoinPackedVector** &left, **CoinPackedVector** &right, **CoinPackedVector** merged) const

  utility to merge vectors into one
- void applyColCuts (**OsiSolverInterface** &si, **OsiCuts** *cuts) const

  our own applyColCuts
- void applyColCuts (**OsiSolverInterface** &si, **OsiColCut** ***cut**) const

our own applyColCut, single cut

- int OsiCuts2MatrVec (**OsiSolverInterface** *cglp, **OsiCuts** *cuts, int displRow, int displRhs) const

  add CGLP columns to solver interface; return number of columns added (for later removal)

Protected Attributes

- CouenneCutGenerator * couenneCG_

  pointer to symbolic repr. of constraint, variables, and bounds

- int nrootcuts_

  number of cuts generated at the first call

- int ntotalcuts_

  total number of cuts generated

- double septime_

  separation time (includes generation of problem)

- double objValue_

  Record obj value at final point of CouenneConv.

- Bonmin::OsiTMINLPInterface * minlp_

  nonlinear solver interface as used within Bonmin (used at first Couenne pass of each b&b node)

- **OsiChooseVariable** * branchingMethod_

  Branching scheme (if strong, we can use SB candidates)

- bool isBranchingStrong_

  Is branchMethod_ referred to a strong branching scheme?

- **JnlstPtr** jnlst_

  SmartPointer to the Journalist.

- int numDisjunctions_

  Number of disjunction to consider at each separation.

- double initDisjPercentage_

  Initial percentage of objects to use for generating cuts, in [0,1].

- int initDisjNumber_

  Initial number of objects to use for generating cuts.

- int depthLevelling_

  Depth of the BB tree where start decreasing number of objects.

- int depthStopSeparate_

  Depth of the BB tree where stop separation.

- bool activeRows_

  only include active rows in CGLP

- bool activeCols_

  only include active columns in CGLP

- bool addPreviousCut_

  add previous disj cut to current CGLP?

- double cpuTime_

  maximum CPU time

6.20.1 Detailed Description

Cut Generator for linear convexifications.

Definition at line 34 of file CouenneDisjCuts.hpp.

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

- CouenneDisjCuts.hpp

6.21 Couenne::CouenneExprMatrix Class Reference

Collaboration diagram for Couenne::CouenneExprMatrix:

Classes

- struct compare_pair_ind

Public Member Functions

- CouenneSparseVector & operator∗ (const CouenneSparseVector &factor) const
  matrix ∗ vector
- CouenneExprMatrix & operator∗ (const CouenneExprMatrix &post) const
  matrix ∗ matrix

Protected Attributes

- std::set< std::pair< int, CouenneSparseVector ∗ >, compare_pair_ind > row_
  row major
- std::set< std::pair< int, CouenneSparseVector ∗ >, compare_pair_ind > col_
  col major
- std::vector< expression ∗ > varIndices_
  if used in sdp cuts, contains indices of x_i used in X_ij = x_i ∗ x_j

6.21.1 Detailed Description

Definition at line 104 of file CouenneMatrix.hpp.

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

- CouenneMatrix.hpp

6.22 Couenne::CouenneFeasPump Class Reference

An implementation of the Feasibility pump that uses linearization and **Ipopt** to find the two sequences of points.

`#include <CouenneFeasPump.hpp>`

Inheritance diagram for Couenne::CouenneFeasPump:

Collaboration diagram for Couenne::CouenneFeasPump:

Public Member Functions

- CouenneFeasPump (CouenneProblem *couenne=NULL, CouenneCutGenerator *cg=NULL, **Ipopt::SmartPtr**< **Ipopt::OptionsList** > options=NULL)

  Constructor with (optional) MINLP pointer.
- CouenneFeasPump (const CouenneFeasPump &other)

  Copy constructor.
- virtual ~CouenneFeasPump ()

  Destructor.
- virtual **CbcHeuristic** * clone () const

  Clone.
- CouenneFeasPump & operator= (const CouenneFeasPump &rhs)

  Assignment operator.
- virtual void resetModel (**CbcModel** *model)

  Does nothing, but necessary as **CbcHeuristic** *declares it pure virtual.*
- virtual int solution (double &objectiveValue, double *newSolution)

  Run heuristic, return 1 if a better solution than the one passed is found and 0 otherwise.
- void setNumberSolvePerLevel (int value)

  set number of nlp's solved for each given level of the tree
- virtual CouNumber solveMILP (const CouNumber *nSol, CouNumber *&iSol, int niter, int *nsuciter)

  find integer (possibly NLP-infeasible) point isol closest (according to the l-1 norm of the hessian) to the current NLP-feasible (but fractional) solution nsol
- virtual CouNumber solveNLP (const CouNumber *nSol, CouNumber *&iSol)

  obtain solution to NLP
- expression * updateNLPObj (const double *)

  set new expression as the NLP objective function using argument as point to minimize distance from.
- bool fixIntVariables (const double *sol)

  admits a (possibly fractional) solution and fixes the integer components in the nonlinear problem for later re-solve.
- double findSolution (const double *nSol, double *&sol, int niter, int *nsuciter)

  find feasible solution (called by solveMILP ())
- void init_MILP ()

  initialize all solvers at the first call, where the initial MILP is built
- void initIpoptApp ()

  Common code for initializing non-smartptr ipopt application.
- CouenneProblem * Problem () const

  return pointer to problem
- enum fpCompDistIntType compDistInt () const

  return type of MILP solved
- double multDistNLP () const

  Return Weights in computing distance, in both MILP and NLP (must sum up to 1 for MILP and for NLP):
- double multHessNLP () const

  weight of Hessian in NLP
- double multObjFNLP () const

  weight of objective in NLP

- double multDistMILP () const
  weight of distance in MILP
- double multHessMILP () const
  weight of Hessian in MILP
- double multObjFMILP () const
  weight of objective in MILP
- CouenneTNLP * nlp () const
  return NLP
- int & nCalls ()
  return number of calls (can be changed)
- int milpPhase (double *nSol, double *iSol)
  MILP phase of the FP.
- int nlpPhase (double *iSol, double *nSol)
  NLP phase of the FP.

Static Public Member Functions

- static void registerOptions (**Ipopt::SmartPtr**< Bonmin::RegisteredOptions >)
  initialize options to be read later

6.22.1 Detailed Description

An implementation of the Feasibility pump that uses linearization and **Ipopt** to find the two sequences of points.

Definition at line 57 of file CouenneFeasPump.hpp.

6.22.2 Member Function Documentation

6.22.2.1 virtual int Couenne::CouenneFeasPump::solution (double & *objectiveValue,* double * *newSolution*) [virtual]

Run heuristic, return 1 if a better solution than the one passed is found and 0 otherwise.

objectiveValue Best known solution in input and value of solution found in output

newSolution Solution found by heuristic.

Implements **CbcHeuristic**.

6.22.2.2 expression* Couenne::CouenneFeasPump::updateNLPObj (const double *)

set new expression as the NLP objective function using argument as point to minimize distance from.

Return new objective function

6.22.2.3 bool Couenne::CouenneFeasPump::fixIntVariables (const double * *sol*)

admits a (possibly fractional) solution and fixes the integer components in the nonlinear problem for later re-solve.

Returns false if restriction infeasible, true otherwise

6.22.2.4 double Couenne::CouenneFeasPump::multDistNLP ( ) const [inline]

Return Weights in computing distance, in both MILP and NLP (must sum up to 1 for MILP and for NLP):

weight of distance in NLP

Definition at line 145 of file CouenneFeasPump.hpp.

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

- CouenneFeasPump.hpp

6.23 Couenne::CouenneFixPoint Class Reference

Cut Generator for FBBT fixpoint.

`#include <CouenneFixPoint.hpp>`

Inheritance diagram for Couenne::CouenneFixPoint:

Collaboration diagram for Couenne::CouenneFixPoint:

Public Member Functions

- CouenneFixPoint (CouenneProblem *, const **Ipopt::SmartPtr**< **Ipopt::OptionsList** >)
  constructor
- CouenneFixPoint (const CouenneFixPoint &)
  copy constructor
- ~CouenneFixPoint ()
  destructor
- CouenneFixPoint * clone () const
  clone method (necessary for the abstract **CglCutGenerator** *class)*
- void generateCuts (const **OsiSolverInterface** &, **OsiCuts** &, const **CglTreeInfo**=**CglTreeInfo**()) const
  the main **CglCutGenerator**

Static Public Member Functions

- static void registerOptions (**Ipopt::SmartPtr**< Bonmin::RegisteredOptions > roptions)
  Add list of options to be read from file.

Protected Member Functions

- void createRow (int, int, int, **OsiSolverInterface** *, const int *, const double *, const double, const int, bool, int, int) const
  Create a single cut.

Protected Attributes

- bool extendedModel_
  should we use an extended model or a more compact one?

- CouenneProblem * problem_
  pointer to the CouenneProblem representation
- bool firstCall_
  Is this the first call?
- double CPUtime_
  CPU time.
- int nTightened_
  Number of bounds tightened.
- CouenneBTPerfIndicator perfIndicator_
  Performance indicator.

6.23.1 Detailed Description

Cut Generator for FBBT fixpoint.

Definition at line 30 of file CouenneFixPoint.hpp.

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

- CouenneFixPoint.hpp

6.24 Couenne::CouenneFPpool Class Reference

Pool of solutions.

`#include <CouenneFPpool.hpp>`

Collaboration diagram for Couenne::CouenneFPpool:

Public Member Functions

- CouenneFPpool (CouenneProblem *p, enum what_to_compare c)
  simple constructor (empty pool)
- CouenneFPpool (const CouenneFPpool &src)
  copy constructor
- CouenneFPpool & operator= (const CouenneFPpool &src)
  assignment
- std::set< CouenneFPsolution, compareSol > & Set ()
  return the main object in this class
- CouenneProblem * Problem ()
  return the problem pointer
- void findClosestAndReplace (double *&sol, const double *nSol, int nvars)
  finds, in pool, solution x closest to sol; removes it from the pool and overwrites it to sol

Protected Attributes

- std::set< CouenneFPsolution, compareSol > set_
  Pool.
- CouenneProblem * problem_
  Problem pointer.

6.24.1 Detailed Description

Pool of solutions.

Definition at line 91 of file CouenneFPpool.hpp.

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

- CouenneFPpool.hpp

6.25 Couenne::CouenneFPsolution Class Reference

Class containing a solution with infeasibility evaluation.

`#include <CouenneFPpool.hpp>`

Collaboration diagram for Couenne::CouenneFPsolution:

Public Member Functions

- CouenneFPsolution (CouenneProblem *p, CouNumber *x, bool copied=false)
  CouenneProblem-aware constructor.
- CouenneFPsolution (const CouenneFPsolution &src)
  copy constructor
- CouenneFPsolution & operator= (const CouenneFPsolution &src)
  assignment
- ~CouenneFPsolution ()
  destructor
- const int n () const
  returns size
- const double * x () const
  returns vector
- bool compare (const CouenneFPsolution &other, enum what_to_compare comparedTerm) const
  basic comparison procedure – what to compare depends on user's choice

Protected Attributes

- CouNumber * x_
  solution
- int n_
  number of variables (for independence from CouenneProblem)
- int nNLinf_
  number of NL infeasibilities
- int nlinf_
  number of integer infeasibilities
- CouNumber objVal_
  objective function value
- CouNumber maxNLinf_
  maximum NL infeasibility

- CouNumber maxlinf_

  maximum integer infeasibility
- bool copied_

  This is a temporary copy, not really a solution holder.
- CouenneProblem * problem_

  holds pointer to problem to check integrality in comparison of integer variables

6.25.1 Detailed Description

Class containing a solution with infeasibility evaluation.

Definition at line 32 of file CouenneFPpool.hpp.

6.25.2 Member Data Documentation

6.25.2.1 bool Couenne::CouenneFPsolution::copied_ [protected]

This is a temporary copy, not really a solution holder.

As a result, all the above members are meaningless for copied solutions

Definition at line 48 of file CouenneFPpool.hpp.

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

- CouenneFPpool.hpp

6.26 Couenne::CouenneInfo Class Reference

Bonmin class for passing info between components of branch-and-cuts.

`#include <BonCouenneInfo.hpp>`

Inheritance diagram for Couenne::CouenneInfo:

Collaboration diagram for Couenne::CouenneInfo:

Classes

- class NlpSolution

  Class for storing an Nlp Solution.

Public Member Functions

- CouenneInfo (int type)

  Default constructor.
- CouenneInfo (const **OsiBabSolver** &other)

  Constructor from **OsiBabSolver**.
- CouenneInfo (const CouenneInfo &other)

  Copy constructor.

- virtual ~CouenneInfo ()

  Destructor.
- virtual **OsiAuxInfo** * clone () const

  Virtual copy constructor.
- const std::list< **Ipopt::SmartPtr**< const NlpSolution > > & NlpSolutions () const

  List of all stored NLP solutions.
- void addSolution (**Ipopt::SmartPtr**< const NlpSolution > newSol)

  Add a new NLP solution.

6.26.1 Detailed Description

Bonmin class for passing info between components of branch-and-cuts.

Definition at line 22 of file BonCouenneInfo.hpp.

6.26.2 Constructor & Destructor Documentation

6.26.2.1 Couenne::CouenneInfo::CouenneInfo (int *type*)

Default constructor.

6.26.2.2 Couenne::CouenneInfo::CouenneInfo (const OsiBabSolver & *other*)

Constructor from **OsiBabSolver**.

6.26.2.3 Couenne::CouenneInfo::CouenneInfo (const CouenneInfo & *other*)

Copy constructor.

6.26.2.4 virtual Couenne::CouenneInfo::~CouenneInfo ( ) [virtual]

Destructor.

6.26.3 Member Function Documentation

6.26.3.1 virtual OsiAuxInfo* Couenne::CouenneInfo::clone ( ) const [virtual]

Virtual copy constructor.

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

- BonCouenneInfo.hpp

6.27 Couenne::CouenneInterface Class Reference

Inheritance diagram for Couenne::CouenneInterface:

Collaboration diagram for Couenne::CouenneInterface:

Public Member Functions

- CouenneInterface ()

  Default constructor.
- CouenneInterface (const CouenneInterface &other)

  Copy constructor.
- virtual CouenneInterface * clone (bool CopyData)

  virutal copy constructor.
- virtual ~CouenneInterface ()

  Destructor.

Overloaded methods to build outer approximations

- bool have_nlp_solution_

  true if we got an integer feasible solution from initial solve
- virtual void extractLinearRelaxation (**OsiSolverInterface** &si, CouenneCutGenerator &couenneCg, bool getObj=1, bool solveNlp=1)

  Extract a linear relaxation of the MINLP.
- virtual void setAppDefaultOptions (**Ipopt::SmartPtr**< **Ipopt::OptionsList** > **Options**)

  To set some application specific defaults.
- bool haveNlpSolution ()

  *return value of have_nlp_solution_*

6.27.1 Detailed Description

Definition at line 33 of file BonCouenneInterface.hpp.

6.27.2 Constructor & Destructor Documentation

6.27.2.1 Couenne::CouenneInterface::CouenneInterface ( )

Default constructor.

6.27.2.2 Couenne::CouenneInterface::CouenneInterface (const CouenneInterface & *other*)

Copy constructor.

6.27.2.3 virtual Couenne::CouenneInterface::~CouenneInterface ( ) [virtual]

Destructor.

6.27.3 Member Function Documentation

6.27.3.1 virtual CouenneInterface* Couenne::CouenneInterface::clone (bool *CopyData*) [virtual]

virutal copy constructor.

6.27.3.2 virtual void Couenne::CouenneInterface::extractLinearRelaxation (OsiSolverInterface & *si*, CouenneCutGenerator & *couenneCg*, bool *getObj* = 1, bool *solveNlp* = 1) [virtual]

Extract a linear relaxation of the MINLP.

Solve the continuous relaxation and takes first-order outer-approximation constraints at the optimum. The put everything in an **OsiSolverInterface**.

6.27.3.3 virtual void Couenne::CouenneInterface::setAppDefaultOptions (Ipopt::SmartPtr< Ipopt::OptionsList > *Options*) [virtual]

To set some application specific defaults.

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

- BonCouenneInterface.hpp

6.28 Couenne::CouenneIterativeRounding Class Reference

An iterative rounding heuristic, tailored for nonconvex MINLPs.

`#include <CouenneIterativeRounding.hpp>`

Inheritance diagram for Couenne::CouenneIterativeRounding:

Collaboration diagram for Couenne::CouenneIterativeRounding:

Public Member Functions

- CouenneIterativeRounding ()

  Default constructor.
- CouenneIterativeRounding (Bonmin::OsiTMINLPInterface *nlp, **OsiSolverInterface** *cinlp, CouenneProblem *couenne, **Ipopt::SmartPtr**< **Ipopt::OptionsList** > options)

  Constructor with model and Couenne problems.
- CouenneIterativeRounding (const CouenneIterativeRounding &other)

  Copy constructor.
- virtual ~CouenneIterativeRounding ()

  Destructor.
- virtual **CbcHeuristic** * clone () const

  Clone.
- CouenneIterativeRounding & operator= (const CouenneIterativeRounding &rhs)

  Assignment operator.
- void setNlp (Bonmin::OsiTMINLPInterface *nlp, **OsiSolverInterface** *cinlp)

  Set the minlp solver.
- void setCouenneProblem (CouenneProblem *couenne)

  Set the couenne problem to use.
- void resetModel (**CbcModel** *model)

  Does nothing.
- int solution (double &objectiveValue, double *newSolution)

  Run heuristic, return 1 if a better solution than the one passed is found and 0 otherwise.

- void setMaxRoundingIter (int value)

  Set maximum number of iterations for each rounding phase.
- void setMaxFirPoints (int value)

  Set maximum number of points that we try to round in F-IR.
- void setMaxTime (double value)

  Set maximum CPU time for the heuristic at each node.
- void setMaxTimeFirstCall (double value)

  Set maximum CPU time for the heuristic at the root node only.
- void setOmega (double value)

  Set the value for omega, the multiplicative factor for the minimum log-barrier parameter mu used by F-IR whenever we need to generate a new NLP feasible point (in the interior of the feasible region)
- void setBaseLbRhs (int value)

  Set the base value for the rhs of the local branching constraint in the I-IR heuristic.
- void setAggressiveness (int value)

  Set aggressiveness of heuristic.

Static Public Member Functions

- static void registerOptions (**Ipopt::SmartPtr**< Bonmin::RegisteredOptions >)

  initialize options to be read later

6.28.1 Detailed Description

An iterative rounding heuristic, tailored for nonconvex MINLPs.

It solves a sequence of MILPs and NLPs for a given number of iterations, or until a better solution is found.

Definition at line 36 of file CouenneIterativeRounding.hpp.

6.28.2 Constructor & Destructor Documentation

6.28.2.1 Couenne::CouenneIterativeRounding::CouenneIterativeRounding ( )

Default constructor.

6.28.2.2 Couenne::CouenneIterativeRounding::CouenneIterativeRounding (Bonmin::OsiTMINLPInterface * *nlp*, OsiSolverInterface * *cinlp*, CouenneProblem * *couenne*, Ipopt::SmartPtr< Ipopt::OptionsList > *options*)

Constructor with model and Couenne problems.

6.28.2.3 Couenne::CouenneIterativeRounding::CouenneIterativeRounding (const CouenneIterativeRounding & *other*)

Copy constructor.

6.28.3 Member Function Documentation

6.28.3.1 virtual CbcHeuristic* Couenne::CouenneIterativeRounding::clone ( ) const [virtual]

Clone.

Implements **CbcHeuristic**.

6.28.3.2 void Couenne::CouenneIterativeRounding::setNlp (Bonmin::OsiTMINLPInterface * *nlp,* OsiSolverInterface * *cinlp*)

Set the minlp solver.

6.28.3.3 void Couenne::CouenneIterativeRounding::setCouenneProblem (CouenneProblem * *couenne*) [inline]

Set the couenne problem to use.

Definition at line 62 of file CouenneIterativeRounding.hpp.

6.28.3.4 void Couenne::CouenneIterativeRounding::resetModel (CbcModel * *model*) [inline], [virtual]

Does nothing.

Implements **CbcHeuristic**.

Definition at line 67 of file CouenneIterativeRounding.hpp.

6.28.3.5 int Couenne::CouenneIterativeRounding::solution (double & *objectiveValue,* double * *newSolution*) [virtual]

Run heuristic, return 1 if a better solution than the one passed is found and 0 otherwise.

objectiveValue Best known solution in input and value of solution found in output newSolution Solution found by heuristic.

Implements **CbcHeuristic**.

6.28.3.6 void Couenne::CouenneIterativeRounding::setBaseLbRhs (int *value*) [inline]

Set the base value for the rhs of the local branching constraint in the I-IR heuristic.

The actual rhs is then computed depending on current variable bounds

Definition at line 109 of file CouenneIterativeRounding.hpp.

6.28.3.7 void Couenne::CouenneIterativeRounding::setAggressiveness (int *value*)

Set aggressiveness of heuristic.

Three levels, that sets various parameters accordingly.

The levels are: 0: maxRoundingIter = 5, maxTimeFirstCall = 300, maxFirPoints = 5, maxTime = 60 1: maxRoundingIter = 10, maxTimeFirstCall = 300, maxFirPoints = 5, maxTime = 120 2: maxRoundingIter = 20, maxTimeFirstCall = 1000, maxFirPoints = 5, maxTime = 300

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

- CouenneIterativeRounding.hpp

6.29 Couenne::CouenneMINLPInterface Class Reference

This is class provides an Osi interface for a Mixed Integer Linear Program expressed as a TMINLP (so that we can use it for example as the continuous solver in Cbc).

`#include <CouenneMINLPInterface.hpp>`

Inheritance diagram for Couenne::CouenneMINLPInterface:

Collaboration diagram for Couenne::CouenneMINLPInterface:

Public Member Functions

- void setObj (int index, expression *newObj)

  REMOVE — backward compatibility sets objective[index] at newObj.
- void setInitSol (const CouNumber *sol)

  sets the initial solution for the NLP solver
- CouNumber solve (CouNumber *solution)

  solves and returns the optimal objective function and the solution
- CouenneProblem * problem () const

  return pointer to Couenne problem
- **Ipopt::OptionsList** * options () const

  return pointer to options

6.29.1 Detailed Description

This is class provides an Osi interface for a Mixed Integer Linear Program expressed as a TMINLP (so that we can use it for example as the continuous solver in Cbc).

Definition at line 59 of file CouenneMINLPInterface.hpp.

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

- CouenneMINLPInterface.hpp

6.30 Couenne::CouenneMultiVarProbe Class Reference

Inheritance diagram for Couenne::CouenneMultiVarProbe:

Collaboration diagram for Couenne::CouenneMultiVarProbe:

Public Member Functions

- CouenneMultiVarProbe (CouenneSetup *couenne, const **Ipopt::SmartPtr**< **Ipopt::OptionsList** > options)

  Constructor.
- CouenneMultiVarProbe (const CouenneMultiVarProbe &rhs)

  Copy constructor.
- ~CouenneMultiVarProbe ()

  Destructor.
- CouenneMultiVarProbe * clone () const

Clone method (necessary for the abstract **CglCutGenerator** *class)*

- void generateCuts (const **OsiSolverInterface** &solver, **OsiCuts** &cuts, const **CglTreeInfo**=**CglTreeInfo**()) const

  The main **CglCutGenerator***; not implemented yet.*

Protected Attributes

- CouenneSetup * couenne_

  Pointer to the CouenneProblem representation.
- int numCols_

  Number of columns (want to have this handy)
- double maxTime_

  Maximum time to probe one variable.

6.30.1 Detailed Description

Definition at line 25 of file CouenneMultiVarProbe.hpp.

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

- CouenneMultiVarProbe.hpp

6.31 Couenne::CouenneObject Class Reference

OsiObject for auxiliary variables $w=f(x)$.

`#include <CouenneObject.hpp>`

Inheritance diagram for Couenne::CouenneObject:

Collaboration diagram for Couenne::CouenneObject:

Public Types

- enum pseudocostMult

  type of up/down estimate to return for pseudocosts
- enum branch_obj

  type of object (for branching variable selection)
- enum brSelStrat

  strategy names

Public Member Functions

- CouenneObject ()

  empty constructor (for unused objects)
- CouenneObject (CouenneCutGenerator *cutgen, CouenneProblem *p, exprVar *ref, Bonmin::BabSetupBase *base, **JnlstPtr** jnlst)

  Constructor with information for branching point selection strategy.
- CouenneObject (exprVar *ref, Bonmin::BabSetupBase *base, **JnlstPtr** jnlst)

Constructor with lesser information, used for infeasibility only.

- ~CouenneObject ()

  Destructor.

- CouenneObject (const CouenneObject &src)

  Copy constructor.

- virtual CouenneObject * clone () const

  Cloning method.

- void setParameters (Bonmin::BabSetupBase *base)

  set object parameters by reading from command line

- virtual double infeasibility (const **OsiBranchingInformation** *info, int &way) const

  compute infeasibility of this variable, |w - f(x)| (where w is the auxiliary variable defined as w = f(x)

- virtual double checkInfeasibility (const **OsiBranchingInformation** *info) const

  compute infeasibility of this variable, |w - f(x)|, where w is the auxiliary variable defined as w = f(x)

- virtual double feasibleRegion (**OsiSolverInterface** *, const **OsiBranchingInformation** *) const

  fix (one of the) arguments of reference auxiliary variable

- virtual **OsiBranchingObject** * createBranch (**OsiSolverInterface** *, const **OsiBranchingInformation** *, int) const

  create CouenneBranchingObject or CouenneThreeWayBranchObj based on this object

- exprVar * Reference () const

  return reference auxiliary variable

- enum brSelStrat Strategy () const

  return branching point selection strategy

- CouNumber getBrPoint (funtriplet *ft, CouNumber x0, CouNumber l, CouNumber u, const **OsiBranching↩ Information** *info=NULL) const

  pick branching point based on current strategy

- CouNumber midInterval (CouNumber x, CouNumber l, CouNumber u, const **OsiBranchingInformation** *info=NULL) const

  returns a point "inside enough" a given interval, or x if it already is.

- virtual double downEstimate () const

  Return "down" estimate (for non-convex, distance old <–> new LP point)

- virtual double upEstimate () const

  Return "up" estimate (for non-convex, distance old <–> new LP point)

- void setEstimate (double est, int direction)

  set up/down estimate (0 for down, 1 for up).

- void setEstimates (const **OsiBranchingInformation** *info, CouNumber *infeasibility, CouNumber *brpt) const

  set up/down estimates based on branching information

- virtual bool isCuttable () const

  are we on the bad or good side of the expression?

- virtual double intInfeasibility (double value, double lb, double ub) const

  integer infeasibility: min {value - floor(value), ceil(value) - value}

- CouNumber lp_clamp () const

  Defines safe interval percentage for using LP point as a branching point.

- virtual int columnNumber () const

  Returns the column index.

Protected Attributes

- CouenneCutGenerator * cutGen_

  pointer to cut generator (not necessary, can be NULL)
- CouenneProblem * problem_

  pointer to Couenne problem
- exprVar * reference_

  The (auxiliary) variable this branching object refers to.
- enum brSelStrat strategy_

  Branching point selection strategy.
- **JnlstPtr** jnlst_

  SmartPointer to the Journalist.
- CouNumber alpha_

  Combination parameter for the mid-point branching point selection strategy.
- CouNumber lp_clamp_

  Defines safe interval percentage for using LP point as a branching point.
- CouNumber feas_tolerance_

  feasibility tolerance (equal to that of CouenneProblem)
- bool doFBBT_

  shall we do Feasibility based Bound Tightening (FBBT) at branching?
- bool doConvCuts_

  shall we add convexification cuts at branching?
- double downEstimate_

  down estimate (to be used in pseudocost)
- double upEstimate_

  up estimate (to be used in pseudocost)
- enum pseudocostMult pseudoMultType_

  multiplier type for pseudocost

6.31.1 Detailed Description

OsiObject for auxiliary variables $w=f(x)$.

Associated with a multi-variate function $f(x)$ and a related infeasibility $|w-f(x)|$, creates branches to help restoring feasibility

Definition at line 57 of file CouenneObject.hpp.

6.31.2 Member Function Documentation

6.31.2.1 CouNumber Couenne::CouenneObject::midInterval (CouNumber *x*, CouNumber *l*, CouNumber *u*, const OsiBranchingInformation * *info* = `NULL`) const

returns a point "inside enough" a given interval, or x if it already is.

Modify alpha_ using gap provided by info

6.31.2.2 void Couenne::CouenneObject::setEstimate (double *est,* int *direction*) [inline]

set up/down estimate (0 for down, 1 for up).

This happens in CouenneChooseStrong, where a new LP point is available and we can measure distance from old LP point. This is the denominator we use in pseudocost

Definition at line 150 of file CouenneObject.hpp.

6.31.3 Member Data Documentation

6.31.3.1 exprVar* Couenne::CouenneObject::reference_ [protected]

The (auxiliary) variable this branching object refers to.

If the expression is w=f(x,y), this is w, as opposed to CouenneBranchingObject, where it would be either x or y.

Definition at line 188 of file CouenneObject.hpp.

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

- CouenneObject.hpp

6.32 Couenne::CouenneObjective Class Reference

Objective function.

`#include <CouenneProblemElem.hpp>`

Collaboration diagram for Couenne::CouenneObjective:

Public Member Functions

- CouenneObjective (expression *body)

  constructor
- ~CouenneObjective ()

  destructor
- CouenneObjective (const CouenneObjective &o, Domain *d=NULL)

  copy constructor
- CouenneObjective * clone (Domain *d=NULL) const

  cloning method
- expression * Body () const

  get body
- expression * Body (expression *newBody)

  Set body.
- exprAux * standardize (CouenneProblem *p)

  Get standard form of this objective function.
- void print (std::ostream &out=std::cout)

  Print to iostream.

Protected Attributes

- expression * body_

  expression to optimize

6.32.1 Detailed Description

Objective function.

It consists of an expression only. We only assume minimization problems (proper sign changes are applied upon reading)

Definition at line 109 of file CouenneProblemElem.hpp.

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

- CouenneProblemElem.hpp

6.33 Couenne::CouenneOrbitBranchingObj Class Reference

"Spatial" branching object.

`#include <CouenneOrbitBranchingObj.hpp>`

Inheritance diagram for Couenne::CouenneOrbitBranchingObj:

Collaboration diagram for Couenne::CouenneOrbitBranchingObj:

Public Member Functions

- CouenneOrbitBranchingObj (**OsiSolverInterface** *solver, const **OsiObject** ***originalObject**, **JnlstPtr** jnlst, CouenneCutGenerator *c, CouenneProblem *p, expression *var, int **way**, CouNumber brpoint, bool doFBBT, bool doConvCuts)

  Constructor.
- CouenneOrbitBranchingObj (const CouenneOrbitBranchingObj &src)

  Copy constructor.
- virtual **OsiBranchingObject** * clone () const

  cloning method
- virtual double branch (**OsiSolverInterface** *solver=NULL)

  Execute the actions required to branch, as specified by the current state of the branching object, and advance the object's state.
- virtual bool boundBranch () const

  does this branching object only change variable bounds?
- void setSimulate (bool s)

  set simulate_ field below

Additional Inherited Members

6.33.1 Detailed Description

"Spatial" branching object.

Branching can also be performed on continuous variables.

Definition at line 36 of file CouenneOrbitBranchingObj.hpp.

6.33.2 Member Function Documentation

6.33.2.1 virtual double Couenne::CouenneOrbitBranchingObj::branch (OsiSolverInterface * *solver* = NULL) [virtual]

Execute the actions required to branch, as specified by the current state of the branching object, and advance the object's state.

Returns change in guessed objective on next branch

Reimplemented from Couenne::CouenneBranchingObject.

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

- CouenneOrbitBranchingObj.hpp

6.34 Couenne::CouenneOSInterface Class Reference

Inheritance diagram for Couenne::CouenneOSInterface:

Collaboration diagram for Couenne::CouenneOSInterface:

Public Member Functions

- CouenneProblem * getCouenneProblem ()

  Should return the problem to solve in algebraic form.
- **Ipopt::SmartPtr**< Bonmin::TMINLP > getTMINLP ()

  Should return the problem to solve as TMINLP.
- bool writeSolution (Bonmin::Bab &bab)

  Called after B&B finished.

6.34.1 Detailed Description

Definition at line 36 of file CouenneOSInterface.hpp.

6.34.2 Member Function Documentation

6.34.2.1 CouenneProblem* Couenne::CouenneOSInterface::getCouenneProblem ( ) [virtual]

Should return the problem to solve in algebraic form.

NOTE: Couenne is (currently) going to modify this problem!

Implements Couenne::CouenneUserInterface.

6.34.2.2 bool Couenne::CouenneOSInterface::writeSolution (Bonmin::Bab & *bab*) [virtual]

Called after B&B finished.

Should write solution information.

Reimplemented from Couenne::CouenneUserInterface.

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

- CouenneOSInterface.hpp

6.35 Couenne::CouenneProblem Class Reference

Class for MINLP problems with symbolic information.

`#include <CouenneProblem.hpp>`

Collaboration diagram for Couenne::CouenneProblem:

Public Types

- enum multiSep
  Type of multilinear separation.

Public Member Functions

- CouenneProblem (ASL *=NULL, Bonmin::BabSetupBase *base=NULL, **JnlstPtr** jnlst=NULL)
  Constructor.
- CouenneProblem (const CouenneProblem &)
  Copy constructor.
- ~CouenneProblem ()
  Destructor.
- void initOptions (**Ipopt::SmartPtr**< **Ipopt::OptionsList** > options)
  initializes parameters like doOBBT
- CouenneProblem * clone () const
  Clone method (for use within CouenneCutGenerator::clone)
- int nObjs () const
  Get number of objectives.
- int nCons () const
  Get number of constraints.
- int nOrigCons () const
  Get number of original constraints.
- int nOrigVars () const
  Number of orig. variables.
- int nDefVars () const
  Number of def'd variables.
- int nOrigIntVars () const
  Number of original integers.
- int nIntVars () const

Number of integer variables.

- int nVars () const

  Total number of variables.

- void setupSymmetry ()

  empty if no NTY, symmetry data structure setup otherwise

- int evalOrder (int i) const

  get evaluation order index

- int * evalVector ()

  get evaluation order vector (numbering_)

- CouenneConstraint * Con (int i) const

  i-th constraint

- CouenneObjective * Obj (int i) const

  i-th objective

- exprVar * Var (int i) const

  Return pointer to i-th variable.

- std::vector< exprVar * > & Variables ()

  Return vector of variables (symbolic representation)

- std::set< exprAux *, compExpr > *& AuxSet ()

  Return pointer to set for comparisons.

- DepGraph * getDepGraph ()

  Return pointer to dependence graph.

- Domain * domain () const

  return current point & bounds

- CouNumber & X (int i) const

  x_i

- CouNumber & Lb (int i) const

  lower bound on x_i

- CouNumber & Ub (int i) const

  upper bound on x_i

- CouNumber * X () const

  Return vector of variables.

- CouNumber * Lb () const

  Return vector of lower bounds.

- CouNumber * Ub () const

  Return vector of upper bounds.

- CouNumber *& bestSol () const

  Best known solution (read from file)

- CouNumber bestObj () const

  Objective of best known solution.

- bool *& Commuted ()

  Get vector of commuted variables.

- void addObjective (expression *, const std::string &="min")

  Add (non linear) objective function.

- void addEQConstraint (expression *, expression *=NULL)

  Add equality constraint $h(x) = b$.

- void addGEConstraint (expression *, expression *=NULL)

  Add $\geq$ *constraint,* $h(x) \geq b$.

- void addLEConstraint (expression *, expression *=NULL)

  Add $\leq$ constraint, $h(x) \leq b$.
- void addRNGConstraint (expression *, expression *=NULL, expression *=NULL)

  Add range constraint, $a \leq h(x) \leq b$.
- void setObjective (int indObj=0, expression *=NULL, const std::string &="min")

  Add (non linear) objective function.
- expression * addVariable (bool isint=false, Domain *d=NULL)

  Add original variable.
- exprAux * addAuxiliary (expression *)

  Add auxiliary variable and associate it with expression given as argument (used in standardization)
- void reformulate (CouenneCutGenerator *=NULL)

  preprocess problem in order to extract linear relaxations etc.
- bool standardize ()

  Break problem's nonlinear constraints in simple expressions to be convexified later.
- void print (std::ostream &=std::cout)

  Display current representation of problem: objective, linear and nonlinear constraints, and auxiliary variables.
- bool doFBBT () const

  shall we do Feasibility Based Bound Tightening?
- bool doRCBT () const

  shall we do reduced cost Bound Tightening?
- bool doOBBT () const

  shall we do Optimality Based Bound Tightening?
- bool doABT () const

  shall we do Aggressive Bound Tightening?
- int logObbtLev () const

  How often shall we do OBBT?
- int logAbtLev () const

  How often shall we do ABT?
- void writeAMPL (const std::string &fname, bool aux)

  Write nonlinear problem to a .mod file (with lots of defined variables)
- void writeGAMS (const std::string &fname)

  Write nonlinear problem to a .gms file.
- void writeLP (const std::string &fname)

  Write nonlinear problem to a .lp file.
- void initAuxs () const

  Initialize auxiliary variables and their bounds from original variables.
- void getAuxs (CouNumber *) const

  Get auxiliary variables from original variables.
- bool boundTightening (t_chg_bounds *, const **CglTreeInfo** info, Bonmin::BabInfo *=NULL) const

  tighten bounds using propagation, implied bounds and reduced costs
- bool btCore (t_chg_bounds *chg_bds) const

  core of the bound tightening procedure
- int obbt (const CouenneCutGenerator *cg, const **OsiSolverInterface** &csi, **OsiCuts** &cs, const **CglTreeInfo** &info, Bonmin::BabInfo *babInfo, t_chg_bounds *chg_bds)

  Optimality Based Bound Tightening.
- bool aggressiveBT (Bonmin::OsiTMINLPInterface *nlp, t_chg_bounds *, const **CglTreeInfo** &info, Bonmin::Bab↩ Info *=NULL) const

aggressive bound tightening.

- int redCostBT (const **OsiSolverInterface** *psi, t_chg_bounds *chg_bds) const

  procedure to strengthen variable bounds.

- int tightenBounds (t_chg_bounds *) const

  "Forward" bound tightening, that is, propagate bound of variable x in an expression $w = f(x)$ to the bounds of w.

- int impliedBounds (t_chg_bounds *) const

  "Backward" bound tightening, aka implied bounds.

- void fillQuadIndices ()

  Look for quadratic terms to be used with SDP cuts.

- void fillObjCoeff (double *&)

  Fill vector with coefficients of objective function.

- void auxiliarize (exprVar *, exprVar *=NULL)

  Replace all occurrences of original variable with new aux given as argument.

- void setCutOff (CouNumber cutoff, const CouNumber *sol=NULL) const

  Set cutoff.

- void resetCutOff (CouNumber value=COUENNE_INFINITY) const

  Reset cutoff.

- CouNumber getCutOff () const

  Get cutoff.

- CouNumber * getCutOffSol () const

  Get cutoff solution.

- void installCutOff () const

  Make cutoff known to the problem.

- **ConstJnlstPtr** Jnlst () const

  Provide Journalist.

- bool checkNLP (const double *solution, double &obj, bool recompute=false) const

  Check if solution is MINLP feasible.

- int getIntegerCandidate (const double *xFrac, double *xInt, double *lb, double *ub) const

  generate integer NLP point Y starting from fractional solution using bound tightening

- bool readOptimum (std::string *fname=NULL)

  Read best known solution from file given in argument.

- exprAux * linStandardize (bool addAux, CouNumber c0, LinMap &lmap, QuadMap &qmap)

  standardization of linear exprOp's

- int splitAux (CouNumber, expression *, expression *&, bool *, enum expression::auxSign &)

  split a constraint w - f(x) = c into w's index (it is returned) and rest = f(x) + c

- void indcoe2vector (int *indexL, CouNumber *coeff, std::vector< std::pair< exprVar *, CouNumber > > &lcoeff)

  translates pair (indices, coefficients) into vector with pointers to variables

- void indcoe2vector (int *indexI, int *indexJ, CouNumber *coeff, std::vector< quadElem > &qcoeff)

  translates triplet (indicesI, indicesJ, coefficients) into vector with pointers to variables

- void decomposeTerm (expression *term, CouNumber initCoe, CouNumber &c0, LinMap &lmap, QuadMap &qmap)

  *given (expression *) element of sum, returns (coe,ind0,ind1) depending on element:*

- const std::string & problemName () const

  return problem name

- const std::vector< std::set< int > > & Dependence () const

  return inverse dependence structure

- const std::vector< CouenneObject * > & Objects () const

return object vector

- int findSOS (**CbcModel** *CbcModelPtr, **OsiSolverInterface** *solver, **OsiObject** **objects)

  find SOS constraints in problem
- void setMaxCpuTime (double time)

  set maximum CPU time
- double getMaxCpuTime () const

  return maximum CPU time
- void setBase (Bonmin::BabSetupBase *base)

  save CouenneBase
- void createUnusedOriginals ()

  Some originals may be unused due to their zero multiplicity (that happens when they are duplicates).
- void restoreUnusedOriginals (CouNumber *=NULL) const

  Some originals may be unused due to their zero multiplicity (that happens when they are duplicates).
- int * unusedOriginalsIndices ()

  return indices of neglected redundant variables
- int nUnusedOriginals ()

  number of unused originals
- enum multiSep MultilinSep () const

  return type of separator for multilinear terms
- bool fbbtReachedIterLimit () const

  true if latest call to FBBT terminated due to iteration limit reached
- bool orbitalBranching () const

  return true if orbital branching activated
- void setCheckAuxBounds (bool value)

  set the value for checkAuxBounds.
- bool checkAuxBounds () const

  return true if bounds of auxiliary variables have to be satisfied whenever a solution is tested for MINLP feasibiliry
- enum TrilinDecompType getTrilinDecompType ()

  return type of decomposition of quadrilinear terms
- Bonmin::BabSetupBase * bonBase () const

  options
- double constObjVal () const

  returns constant objective value if it contains no variables
- CouenneSdpCuts * getSdpCutGen ()

  Returns pointer to sdp cut generator.
- CouenneRecordBestSol * getRecordBestSol () const

  returns recorded best solution
- double getFeasTol ()

  returns feasibility tolerance
- double checkObj (const CouNumber *sol, const double &precision) const

  Recompute objective value for sol.
- bool checkInt (const CouNumber *sol, const int from, const int upto, const std::vector< int > listInt, const bool origVarOnly, const bool stopAtFirstViol, const double precision, double &maxViol) const

  check integrality of vars in sol with index between from and upto (original vars only if origVarOnly == true); return true if all integer vars are within precision of an integer value
- bool checkBounds (const CouNumber *sol, const bool stopAtFirstViol, const double precision, double &maxViol) const

Check bounds; returns true iff feasible for given precision.

- bool checkAux (const CouNumber *sol, const bool stopAtFirstViol, const double precision, double &maxViol) const

  returns true iff value of all auxiliaries are within bounds

- bool checkCons (const CouNumber *sol, const bool stopAtFirstViol, const double precision, double &maxViol) const

  returns true iff value of all auxiliaries are within bounds

- bool checkNLP2 (const double *solution, const double obj, const bool careAboutObj, const bool stopAtFirstViol, const bool checkAll, const double precision) const

  Return true if either solution or recomputed_solution obtained using getAuxs() from the original variables in solution is feasible within precision (the solution with minimum violation is then stored in recBSol->modSol, as well as its value and violation); return false otherwise.

- bool checkNLP0 (const double *solution, double &obj, bool recompute_obj=false, const bool careAboutObj=false, const bool stopAtFirstViol=true, const bool checkAll=false, const double precision=-1) const

  And finally a method to get both.

- std::vector< CouenneConstraint * > * ConstraintClass (const char *str)

  return particular constraint class.

Static Public Member Functions

- static void registerOptions (**Ipopt::SmartPtr**< Bonmin::RegisteredOptions > roptions)

  Add list of options to be read from file.

Protected Member Functions

- int fake_tighten (char direction, int index, const double *X, CouNumber *olb, CouNumber *oub, t_chg_bounds *chg_bds, t_chg_bounds *f_chg) const

  single fake tightening.

- int obbtInner (**OsiSolverInterface** *, **OsiCuts** &, t_chg_bounds *, Bonmin::BabInfo *) const

  Optimality Based Bound Tightening – inner loop.

- void analyzeSparsity (CouNumber, LinMap &, QuadMap &)

  analyze sparsity of potential exprQuad/exprGroup and change linear/quadratic maps accordingly, if necessary by adding new auxiliary variables and including them in the linear map

- void flattenMul (expression *mul, CouNumber &coe, std::map< int, CouNumber > &indices)

  re-organizes multiplication and stores indices (and exponents) of its variables

- void realign ()

  clear all spurious variables pointers not referring to the variables_ vector

- void fillDependence (Bonmin::BabSetupBase *base, CouenneCutGenerator *=NULL)

  fill dependence_ structure

- void fillIntegerRank () const

  fill freeIntegers_ array

- int testIntFix (int index, CouNumber xFrac, enum fixType *fixed, CouNumber *xInt, CouNumber *dualL, CouNumber *dualR, CouNumber *olb, CouNumber *oub, bool patient) const

  Test fixing of an integer variable (used in getIntegerCandidate())

Protected Attributes

- std::string problemName_

  problem name
- std::vector< exprVar * > variables_

  Variables (original, auxiliary, and defined)
- std::vector< CouenneObjective * > objectives_

  Objectives.
- std::vector< CouenneConstraint * > constraints_

  Constraints.
- std::vector< expression * > commonexprs_

  AMPL's common expressions (read from AMPL through structures cexps and cexps1)
- Domain domain_

  current point and bounds;
- std::set< exprAux *, compExpr > * auxSet_

  Expression map for comparison in standardization and to count occurrences of an auxiliary.
- int curnvars_

  Number of elements in the x_, lb_, ub_ arrays.
- int nIntVars_

  Number of discrete variables.
- CouNumber * optimum_

  Best solution known to be loaded from file – for testing purposes.
- CouNumber bestObj_

  Best known objective function.
- bool * commuted_

  Variables that have commuted to auxiliary.
- int * numbering_

  numbering of variables.
- int ndefined_

  Number of "defined variables" (aka "common expressions")
- DepGraph * graph_

  Dependence (acyclic) graph: shows dependence of all auxiliary variables on one another and on original variables.
- int nOrigVars_

  Number of original variables.
- int nOrigCons_

  Number of original constraints (disregarding those that turned into auxiliary variable definition)
- int nOrigIntVars_

  Number of original integer variables.
- GlobalCutOff * pcutoff_

  Pointer to a global cutoff object.
- bool created_pcutoff_

  flag indicating if this class is creator of global cutoff object
- bool doFBBT_

  do Feasibility-based bound tightening
- bool doRCBT_

  do reduced cost bound tightening
- bool doOBBT_

do Optimality-based bound tightening

- bool doABT_

  do Aggressive bound tightening

- int logObbtLev_

  frequency of Optimality-based bound tightening

- int logAbtLev_

  frequency of Aggressive bound tightening

- **JnlstPtr** jnlst_

  SmartPointer to the Journalist.

- CouNumber opt_window_

  window around known optimum (for testing purposes)

- bool useQuadratic_

  Use quadratic expressions?

- CouNumber feas_tolerance_

  feasibility tolerance (to be used in checkNLP)

- std::vector< std::set< int > > dependence_

  inverse dependence structure: for each variable x give set of auxiliary variables (or better, their indices) whose expression depends on x

- std::vector< CouenneObject * > objects_

  vector of pointer to CouenneObjects.

- int * integerRank_

  each element is true if variable is integer and, if auxiliary, depends on no integer

- std::vector< int > numberInRank_

  numberInRank_ [i] is the number of integer variables in rank i

- double maxCpuTime_

  maximum cpu time

- Bonmin::BabSetupBase * bonBase_

  options

- ASL * asl_

  AMPL structure pointer (temporary — looking forward to embedding into OS...)

- int * unusedOriginalsIndices_

  some originals may be unused due to their zero multiplicity (that happens when they are duplicates).

- int nUnusedOriginals_

  number of unused originals

- enum multiSep multilinSep_

  Type of Multilinear separation.

- int max_fbbt_iter_

  number of FBBT iterations

- bool fbbtReachedIterLimit_

  true if FBBT exited for iteration limits as opposed to inability to further tighten bounds

- bool orbitalBranching_

  use orbital branching?

- bool checkAuxBounds_

  check bounds on auxiliary variables when verifying MINLP feasibility of a solution.

- enum TrilinDecompType trilinDecompType_

  return type of decomposition of quadrilinear terms

- double constObjVal_

constant value of the objective if no variable is declared in it

- CouenneBTPerfIndicator * perfIndicator_

  Performance indicator for FBBT – to be moved away from CouenneProblem when we do it with FBBT.

- std::map< const char *, std::vector< CouenneConstraint * > *, less_than_str > ConstraintClass_

  Return particular constraint class.

- CouenneSdpCuts * sdpCutGen_

  Temporary pointer to SDP cut generator.

6.35.1 Detailed Description

Class for MINLP problems with symbolic information.

It is read from an AMPL .nl file and contains variables, AMPL's "defined variables" (aka common expressions), objective(s), and constraints in the form of expression's. Changes throughout the program occur in standardization.

Definition at line 169 of file CouenneProblem.hpp.

6.35.2 Member Function Documentation

6.35.2.1 expression* Couenne::CouenneProblem::addVariable (bool *isint* = `false`, Domain * *d* = `NULL`)

Add original variable.

Parameters

isint	if true, this variable is integer, otherwise it is continuous

6.35.2.2 bool Couenne::CouenneProblem::standardize ( )

Break problem's nonlinear constraints in simple expressions to be convexified later.

Return true if problem looks feasible, false if proven infeasible.

6.35.2.3 void Couenne::CouenneProblem::print (std::ostream & = `std::cout`)

Display current representation of problem: objective, linear and nonlinear constraints, and auxiliary variables.

6.35.2.4 void Couenne::CouenneProblem::writeAMPL (const std::string & *fname*, bool *aux*)

Write nonlinear problem to a .mod file (with lots of defined variables)

Parameters

fname	Name of the .mod file to be written
aux	controls the use of auxiliaries. If true, a problem is written with auxiliary variables written with their associated expression, i.e. $w_i = h_i(x, y, w)$ and bounds $l_i \leq w_i \leq u_i$, while if false these constraints are written in the form $l_i \leq h_i(x, y) \leq u_i$.

Note: if used before standardization, writes original AMPL formulation

6.35.2.5 void Couenne::CouenneProblem::writeGAMS (const std::string & *fname*)

Write nonlinear problem to a .gms file.

Parameters

fname	Name of the .gams file to be written.

6.35.2.6 void Couenne::CouenneProblem::writeLP (const std::string & *fname*)

Write nonlinear problem to a .lp file.

Note: only works with MIQCQPs (and MISOCPs in the future)

Parameters

fname	Name of the .lp file to be written

6.35.2.7 bool Couenne::CouenneProblem::aggressiveBT (Bonmin::OsiTMINLPInterface * *nlp,* t_chg_bounds * *,* const CglTreeInfo & *info,* Bonmin::BabInfo * = NULL) const

aggressive bound tightening.

Fake bounds in order to cut portions of the solution space by fathoming on bounds/infeasibility

6.35.2.8 int Couenne::CouenneProblem::redCostBT (const OsiSolverInterface * *psi,* t_chg_bounds * *chg_bds*) const

procedure to strengthen variable bounds.

Return false if problem turns out to be infeasible with given bounds, true otherwise.

6.35.2.9 int Couenne::CouenneProblem::tightenBounds (t_chg_bounds *) const

"Forward" bound tightening, that is, propagate bound of variable x in an expression $w = f(x)$ to the bounds of w.

6.35.2.10 void Couenne::CouenneProblem::decomposeTerm (expression * *term,* CouNumber *initCoe,* CouNumber & *c0,* LinMap & *lmap,* QuadMap & *qmap*)

given (expression *) element of sum, returns (coe,ind0,ind1) depending on element:

1) a * x_i ^ 2 —> (a,i,?) return COU_EXPRPOW 2) a * x_i —> (a,i,?) return COU_EXPRVAR 3) a * x_i * x_j —> (a,i,j) return COU_EXPRMUL 4) a —> (a,?,?) return COU_EXPRCONST

x_i and/or x_j may come from standardizing other (linear or quadratic operator) sub-expressions

6.35.2.11 void Couenne::CouenneProblem::createUnusedOriginals ( )

Some originals may be unused due to their zero multiplicity (that happens when they are duplicates).

This procedure creates a structure for quickly checking and restoring their value after solving.

6.35.2.12 void Couenne::CouenneProblem::restoreUnusedOriginals (CouNumber * = NULL) const

Some originals may be unused due to their zero multiplicity (that happens when they are duplicates).

This procedure restores their value after solving

6.35.2.13 void Couenne::CouenneProblem::setCheckAuxBounds (bool *value*) [inline]

set the value for checkAuxBounds.

When true, all MINLP feasible solutions will additionally be tested for feasibility with respect to auxiliary variable bounds. This is normally not needed.

Definition at line 718 of file CouenneProblem.hpp.

6.35.2.14 int Couenne::CouenneProblem::fake_tighten (char *direction,* int *index,* const double * *X,* CouNumber * *olb,* CouNumber * *oub,* t_chg_bounds * *chg_bds,* t_chg_bounds * *f_chg*) const [protected]

single fake tightening.

Return

-1 if infeasible 0 if no improvement +1 if improved

Parameters

direction	0: left, 1: right
index	index of the variable tested
X	point round which tightening is done
olb	cur. lower bound
oub	cur. upper bound

6.35.2.15 bool Couenne::CouenneProblem::checkNLP2 (const double * *solution,* const double *obj,* const bool *careAboutObj,* const bool *stopAtFirstViol,* const bool *checkAll,* const double *precision*) const

Return true if either solution or recomputed_solution obtained using getAuxs() from the original variables in solution is feasible within precision (the solution with minimum violation is then stored in recBSol->modSol, as well as its value and violation); return false otherwise.

If stopAtFirstViol == true, recBSol->modSol is meaningless upon return. If stopAtFirstViol == false, recBSol->modSol contains the solution with minimum violation, although this violation might be larger than precision. This is useful for cases where the current solution must be considered valid (e.g., because Cbc is going to accept it anyway), although it violates precision requirements. Value of obj matters only if careAboutObj == true; the code then tries to balance violation of constraints and value of objective. if checkAll = false, check only integrality/bounds for original vars and constraints; consider only recomputed_sol if checkAll == true, check also integrality/bounds on auxs; consider both recomputed_sol and solution if careAboutObj is set to true, then stopAtFirstViol must be set to false too.

6.35.2.16 std::vector<CouenneConstraint *>* Couenne::CouenneProblem::ConstraintClass (const char * *str*) [inline]

return particular constraint class.

Classes:

1) "convex": convex constraints; 2) "PSDcon": constraints of the form X 0 3) "normal": regular constraints

Definition at line 896 of file CouenneProblem.hpp.

6.35.3 Member Data Documentation

6.35.3.1 int* Couenne::CouenneProblem::numbering_ [protected]

numbering of variables.

No variable xi with associated pi(i) greater than pi(j) should be evaluated before variable xj

Definition at line 225 of file CouenneProblem.hpp.

6.35.3.2 DepGraph* Couenne::CouenneProblem::graph_ [protected]

Dependence (acyclic) graph: shows dependence of all auxiliary variables on one another and on original variables.

Used to create a numbering of all variables for evaluation and bound tightening (reverse order for implied bounds)

Definition at line 234 of file CouenneProblem.hpp.

6.35.3.3 std::vector<CouenneObject *> Couenne::CouenneProblem::objects_ [protected]

vector of pointer to CouenneObjects.

Used by CouenneVarObjects when finding all objects related to (having as argument) a single variable

Definition at line 280 of file CouenneProblem.hpp.

6.35.3.4 int* Couenne::CouenneProblem::unusedOriginalsIndices_ [protected]

some originals may be unused due to their zero multiplicity (that happens when they are duplicates).

This array keeps track of their indices and is sorted by evaluation order

Definition at line 301 of file CouenneProblem.hpp.

6.35.3.5 bool Couenne::CouenneProblem::checkAuxBounds_ [protected]

check bounds on auxiliary variables when verifying MINLP feasibility of a solution.

Usually this is not needed, unless some manipulation on auxiliary variables is done before Branch-and-Bound

Definition at line 329 of file CouenneProblem.hpp.

6.35.3.6 std::map<const char *, std::vector <CouenneConstraint *> *, less_than_str> Couenne::CouenneProblem::ConstraintClass_ [protected]

Return particular constraint class.

Classes:

1) "convex": convex constraints; 2) "PSDcon": constraints of the form X 0 3) "normal": regular constraints

Definition at line 346 of file CouenneProblem.hpp.

6.35.3.7 CouenneSdpCuts* Couenne::CouenneProblem::sdpCutGen_ [protected]

Temporary pointer to SDP cut generator.

A little dirty as it is generated DURING standardization, but necessary to avoid meddling with different spaces

Definition at line 351 of file CouenneProblem.hpp.

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

- CouenneProblem.hpp

6.36 Couenne::CouennePSDcon Class Reference

Class to represent positive semidefinite constraints ////////////////////.

`#include <CouennePSDcon.hpp>`

Inheritance diagram for Couenne::CouennePSDcon:

Collaboration diagram for Couenne::CouennePSDcon:

Public Member Functions

- CouennePSDcon (CouenneExprMatrix ∗X)

  Constructor.
- ∼CouennePSDcon ()

  Destructor.
- CouennePSDcon (const CouennePSDcon &c, Domain ∗d=NULL)

  Copy constructor.
- CouennePSDcon & operator= (const CouennePSDcon &c)

  Assignment operator.
- CouenneConstraint ∗ clone (Domain ∗d=NULL) const

  Cloning method.
- CouenneExprMatrix ∗ getX () const

  return X
- exprAux ∗ standardize (CouenneProblem ∗)

  Decompose body of constraint through auxiliary variables.
- void print (std::ostream &=std::cout)

  Print constraint.

Protected Attributes

- CouenneExprMatrix ∗ X_

  contains indices of matrix X 0

6.36.1 Detailed Description

Class to represent positive semidefinite constraints ////////////////////.

Definition at line 24 of file CouennePSDcon.hpp.

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

- CouennePSDcon.hpp

6.37 Couenne::CouenneRecordBestSol Class Reference

Collaboration diagram for Couenne::CouenneRecordBestSol:

Public Member Functions

- CouenneRecordBestSol ()
  Constructor.
- CouenneRecordBestSol (const CouenneRecordBestSol &other)
  Copy constructor.
- ~CouenneRecordBestSol ()
  Destructor.

6.37.1 Detailed Description

Definition at line 19 of file CouenneRecordBestSol.hpp.

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

- CouenneRecordBestSol.hpp

6.38 Couenne::CouenneScalar Class Reference

Collaboration diagram for Couenne::CouenneScalar:

Protected Attributes

- int index_
  index of element in vector
- expression * elem_
  element

6.38.1 Detailed Description

Definition at line 25 of file CouenneMatrix.hpp.

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

- CouenneMatrix.hpp

6.39 Couenne::CouenneSdpCuts Class Reference

These are cuts of the form.

`#include <CouenneSdpCuts.hpp>`

Inheritance diagram for Couenne::CouenneSdpCuts:

Collaboration diagram for Couenne::CouenneSdpCuts:

Public Member Functions

- CouenneSdpCuts (CouenneProblem *, **JnlstPtr**, const **Ipopt::SmartPtr**< **Ipopt::OptionsList** >)
  Constructor.
- ~CouenneSdpCuts ()
  Destructor.
- CouenneSdpCuts & operator= (const CouenneSdpCuts &)
  Assignment.
- CouenneSdpCuts (const CouenneSdpCuts &)
  Copy constructor.
- virtual **CglCutGenerator** * clone () const
  Cloning constructor.
- virtual void generateCuts (const **OsiSolverInterface** &, **OsiCuts** &, const **CglTreeInfo**=**CglTreeInfo**()) const
  The main **CglCutGenerator**.

Static Public Member Functions

- static void registerOptions (**Ipopt::SmartPtr**< Bonmin::RegisteredOptions > roptions)
  Add list of options to be read from file.

Protected Attributes

- CouenneProblem * problem_
  pointer to problem info
- bool doNotUse_
  after construction, true if there are enough product terms to justify application.
- std::vector< CouenneExprMatrix * > minors_
  minors on which to apply cuts
- int numEigVec_
  number of eigenvectors to be used (default: n)
- bool onlyNegEV_
  only use negative eigenvalues (default: yes)
- bool useSparsity_
  Sparsify eigenvalues before writing inequality (default: no)
- bool fillMissingTerms_
  If minor not fully dense, create fictitious auxiliary variables that will be used in sdp cuts only (tighter than sdp cuts without)

6.39.1 Detailed Description

These are cuts of the form.

a' X a >= 0

where X is a matrix constrained to be PSD.

Typical application is in problems with products forming a matrix of auxiliary variables X0 = (x_ij)_{i,j in N}, and x_ij is the auxiliary variable for x_i * x_j. After reformulation, matrices like X0 arise naturally and can be used to separate cuts that help strengthen the lower bound. See Sherali and Fraticelli for the base idea, and Qualizza, Belotti and Margot for an efficient rework and its implementation. Andrea Qualizza's code has been made open source and is used here (thanks Andrea!).

Definition at line 43 of file CouenneSdpCuts.hpp.

6.39.2 Member Data Documentation

6.39.2.1 bool Couenne::CouenneSdpCuts::doNotUse_ [protected]

after construction, true if there are enough product terms to justify application.

If not, do not add this cut generator

Definition at line 49 of file CouenneSdpCuts.hpp.

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

- CouenneSdpCuts.hpp

6.40 Couenne::CouenneSetup Class Reference

Inheritance diagram for Couenne::CouenneSetup:

Collaboration diagram for Couenne::CouenneSetup:

Public Member Functions

- CouenneSetup ()

  Default constructor.
- CouenneSetup (const CouenneSetup &other)

  Copy constructor.
- virtual Bonmin::BabSetupBase * clone () const

  virtual copy constructor.
- virtual ~CouenneSetup ()

  destructor
- bool InitializeCouenne (char **argv=NULL, CouenneProblem *couenneProb=NULL, **Ipopt::SmartPtr**< Bonmin::TMINLP > tminlp=NULL, CouenneInterface *ci=NULL, Bonmin::Bab *bb=NULL)

  Initialize from command line arguments.
- virtual void registerOptions ()

  register the options
- virtual void readOptionsFile ()

  Get the basic options if don't already have them.
- CouenneCutGenerator * couennePtr () const

  return pointer to cut generator (used to get pointer to problem)
- bool displayStats ()

  true if one wants to display statistics at the end of program
- void addMilpCutGenerators ()

  add cut generators
- void setDoubleParameter (const DoubleParameter &p, const double val)

  modify parameter (used for MaxTime)
- double getDoubleParameter (const DoubleParameter &p) const

  modify parameter (used for MaxTime)

Static Public Member Functions

- static void registerAllOptions (**Ipopt::SmartPtr**< Bonmin::RegisteredOptions > roptions)

  Register all Couenne options.

6.40.1 Detailed Description

Definition at line 43 of file BonCouenneSetup.hpp.

6.40.2 Constructor & Destructor Documentation

6.40.2.1 Couenne::CouenneSetup::CouenneSetup (const CouenneSetup & *other*) [inline]

Copy constructor.

Definition at line 55 of file BonCouenneSetup.hpp.

6.40.3 Member Function Documentation

6.40.3.1 virtual Bonmin::BabSetupBase* Couenne::CouenneSetup::clone ( ) const [inline], [virtual]

virtual copy constructor.

Definition at line 62 of file BonCouenneSetup.hpp.

6.40.3.2 bool Couenne::CouenneSetup::InitializeCouenne (char ** *argv* = NULL, CouenneProblem * *couenneProb* = NULL, Ipopt::SmartPtr< Bonmin::TMINLP > *tminlp* = NULL, CouenneInterface * *ci* = NULL, Bonmin::Bab * *bb* = NULL)

Initialize from command line arguments.

6.40.3.3 static void Couenne::CouenneSetup::registerAllOptions (Ipopt::SmartPtr< Bonmin::RegisteredOptions > *roptions*) [static]

Register all Couenne options.

6.40.3.4 virtual void Couenne::CouenneSetup::readOptionsFile ( ) [inline], [virtual]

Get the basic options if don't already have them.

Definition at line 81 of file BonCouenneSetup.hpp.

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

- BonCouenneSetup.hpp

6.41 Couenne::CouenneSolverInterface< T > Class Template Reference

Solver interface class with a pointer to a Couenne cut generator.

`#include <CouenneSolverInterface.hpp>`

Inheritance diagram for Couenne::CouenneSolverInterface< T >:

Collaboration diagram for Couenne::CouenneSolverInterface< T >:

Public Member Functions

- CouenneSolverInterface (CouenneCutGenerator *cg=NULL)

  Constructor.
- CouenneSolverInterface (const CouenneSolverInterface &src)

  Copy constructor.
- ~CouenneSolverInterface ()

  Destructor.
- virtual **OsiSolverInterface** * clone (bool copyData=true) const

  Clone.
- virtual bool isProvenPrimalInfeasible () const

  we need to overwrite this since we might have internal knowledge
- virtual bool isProvenOptimal () const

  we need to overwrite this since we might have internal knowledge
- CouenneCutGenerator * CutGen ()

  Return cut generator pointer.
- void setCutGenPtr (CouenneCutGenerator *cg)

  Set cut generator pointer after setup, to avoid changes in the pointer due to cut generator cloning (it happens twice in the algorithm)
- virtual void initialSolve ()

  Solve initial LP relaxation.
- virtual void resolve ()

  Resolve an LP relaxation after problem modification.
- virtual void resolve_nobt ()

  Resolve an LP without applying bound tightening beforehand.
- virtual int tightenBounds (int lightweight)

  Tighten bounds on all variables (including continuous).
- bool isProvenDualInfeasible () const

  *set doingResolve_*
- virtual double getObjValue () const

  Get the objective function value.

Methods for strong branching.

- virtual void markHotStart ()

  Create a hot start snapshot of the optimization process.
- virtual void solveFromHotStart ()

  Optimize starting from the hot start snapshot.
- virtual void unmarkHotStart ()

  Delete the hot start snapshot.

Protected Member Functions

- virtual int tightenBoundsCLP (int lightweight)

  Copy of the Clp version — not light version.
- virtual int tightenBoundsCLP_Light (int lightweight)

  Copy of the Clp version — light version.

Protected Attributes

- CouenneCutGenerator * cutgen_

  The pointer to the Couenne cut generator.
- bool knowInfeasible_

  Flag indicating that infeasibility was detected during solveFromHotStart.
- bool knowOptimal_

  Flag indicating that optimality was detected during solveFromHotStart.
- bool knowDualInfeasible_

  Flag indicating this problem's continuous relaxation is unbounded.

6.41.1 Detailed Description

template<class T>class Couenne::CouenneSolverInterface< T >

Solver interface class with a pointer to a Couenne cut generator.

Its main purposes are:

1) to apply bound tightening before re-solving 2) to replace **OsiSolverInterface::isInteger** () with problem_ -> [expression] -> isInteger () 3) to use NLP solution at branching

Definition at line 21 of file CouenneChooseStrong.hpp.

6.41.2 Member Function Documentation

6.41.2.1 template<class T > bool Couenne::CouenneSolverInterface< T >::isProvenDualInfeasible ( ) const

set doingResolve_

is this problem unbounded?

6.41.2.2 template<class T > virtual double Couenne::CouenneSolverInterface< T >::getObjValue ( ) const [virtual]

Get the objective function value.

Modified due to possible constant objectives passed to Couenne

6.41.3 Member Data Documentation

6.41.3.1 template<class T > CouenneCutGenerator* Couenne::CouenneSolverInterface< T >::cutgen_ [protected]

The pointer to the Couenne cut generator.

Gives us a lot of information, for instance the nlp solver pointer, and the chance to do bound tightening before resolve ().

Definition at line 116 of file CouenneSolverInterface.hpp.

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

- CouenneChooseStrong.hpp
- CouenneSolverInterface.hpp

6.42 Couenne::CouenneSOSBranchingObject Class Reference

Inheritance diagram for Couenne::CouenneSOSBranchingObject:

Collaboration diagram for Couenne::CouenneSOSBranchingObject:

Public Member Functions

- virtual **OsiBranchingObject** * clone () const

  Clone.
- virtual double branch (**OsiSolverInterface** *solver)

  Does next branch and updates state.

Protected Attributes

- CouenneProblem * problem_

  pointer to Couenne problem
- exprVar * reference_

  The (auxiliary) variable this branching object refers to.
- **JnlstPtr** jnlst_

  SmartPointer to the Journalist.
- bool doFBBT_

  shall we do Feasibility based Bound Tightening (FBBT) at branching?
- bool doConvCuts_

  shall we add convexification cuts at branching?

6.42.1 Detailed Description

Definition at line 27 of file CouenneSOSObject.hpp.

6.42.2 Member Data Documentation

6.42.2.1 exprVar* Couenne::CouenneSOSBranchingObject::reference_ [protected]

The (auxiliary) variable this branching object refers to.

If the expression is w=f(x,y), this is w, as opposed to CouenneBranchingObject, where it would be either x or y.

Definition at line 37 of file CouenneSOSObject.hpp.

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

- CouenneSOSObject.hpp

6.43 Couenne::CouenneSOSObject Class Reference

Inheritance diagram for Couenne::CouenneSOSObject:

Collaboration diagram for Couenne::CouenneSOSObject:

Public Member Functions

- CouenneSOSObject (const CouenneSOSObject &src)

  Copy constructor.
- virtual **OsiObject** * clone () const

  Cloning method.
- **OsiBranchingObject** * createBranch (**OsiSolverInterface** *si, const **OsiBranchingInformation** *info, int way) const

  create branching objects

Protected Attributes

- CouenneProblem * problem_

  pointer to Couenne problem
- exprVar * reference_

  The (auxiliary) variable this branching object refers to.
- **JnlstPtr** jnlst_

  SmartPointer to the Journalist.
- bool doFBBT_

  shall we do Feasibility based Bound Tightening (FBBT) at branching?
- bool doConvCuts_

  shall we add convexification cuts at branching?

6.43.1 Detailed Description

Definition at line 95 of file CouenneSOSObject.hpp.

6.43.2 Member Data Documentation

6.43.2.1 exprVar* Couenne::CouenneSOSObject::reference_ [protected]

The (auxiliary) variable this branching object refers to.

If the expression is w=f(x,y), this is w, as opposed to CouenneBranchingObject, where it would be either x or y.

Definition at line 105 of file CouenneSOSObject.hpp.

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

- CouenneSOSObject.hpp

6.44 Couenne::CouenneSparseBndVec< T > Class Template Reference

Public Member Functions

- CouenneSparseBndVec (unsigned int size)

  Constructor.
- CouenneSparseBndVec (CouenneSparseBndVec &src)

  Copy constructor.
- ~CouenneSparseBndVec ()

  Destructor.
- void reset ()

  Reset (eeeeasy!)
- T & operator[ ] (register unsigned int index)

  Access – the only chance for garbage to be returned (and for valgrind to complain) is when object[ind] is READ without making sure it has been written.
- T * data ()

  Return data in DENSE format – use with care.
- unsigned int * indices ()

  Return indices in DENSE format – for use with data()
- unsigned int nElements ()

  Return current size.
- void resize (unsigned int newsize)

  Resize.

6.44.1 Detailed Description

template<class T>class Couenne::CouenneSparseBndVec< T >

Definition at line 16 of file CouenneSparseBndVec.hpp.

6.44.2 Constructor & Destructor Documentation

6.44.2.1 template<class T > Couenne::CouenneSparseBndVec< T >::CouenneSparseBndVec (CouenneSparseBndVec< T > & *src*) [inline]

Copy constructor.

assert: src.sInd [ind] == i

Definition at line 62 of file CouenneSparseBndVec.hpp.

6.44.3 Member Function Documentation

6.44.3.1 template<class T > T& Couenne::CouenneSparseBndVec< T >::operator[ ] (register unsigned int *index*) [inline]

Access – the only chance for garbage to be returned (and for valgrind to complain) is when object[ind] is READ without making sure it has been written.

This should not happen to the end user as read operations are only performed on the dense structure, after this object has been populated.

Definition at line 91 of file CouenneSparseBndVec.hpp.

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

- CouenneSparseBndVec.hpp

6.45 Couenne::CouenneSparseMatrix Class Reference

Class for sparse Matrixs (used in modifying distances in FP)

`#include <CouenneSparseMatrix.hpp>`

Public Member Functions

- CouenneSparseMatrix ()

  Constructor.
- CouenneSparseMatrix (const CouenneSparseMatrix &)

  Copy constructor.
- CouenneSparseMatrix & operator= (const CouenneSparseMatrix &rhs)

  Assignment.
- CouenneSparseMatrix * clone ()

  Clone.
- virtual ~CouenneSparseMatrix ()

  Destructor.
- int & num ()

  Get methods.
- double *& val ()

  values
- int *& col ()

  column indices
- int *& row ()

  row indices

6.45.1 Detailed Description

Class for sparse Matrixs (used in modifying distances in FP)

Definition at line 17 of file CouenneSparseMatrix.hpp.

6.45.2 Member Function Documentation

6.45.2.1 int& Couenne::CouenneSparseMatrix::num ( ) [inline]

Get methods.

number of elements

Definition at line 37 of file CouenneSparseMatrix.hpp.

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

- CouenneSparseMatrix.hpp

6.46 Couenne::CouenneSparseVector Class Reference

Collaboration diagram for Couenne::CouenneSparseVector:

Classes

- struct compare_scalars

Public Member Functions

- const std::set< CouenneScalar *, compare_scalars > & getElements ()
  returns elements of vector as (ordered) set
- double operator* (const CouenneSparseVector &factor) const
  *vector * vector (dot product)*
- CouenneSparseVector & operator* (const CouenneExprMatrix &post) const
  *vector * matrix*
- double multiply_thres (const CouenneSparseVector &v2, double thres) const
  stops multiplication if above threshold

6.46.1 Detailed Description

Definition at line 66 of file CouenneMatrix.hpp.

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

- CouenneMatrix.hpp

6.47 Couenne::CouenneThreeWayBranchObj Class Reference

Spatial, three-way branching object.

`#include <CouenneThreeWayBranchObj.hpp>`

Inheritance diagram for Couenne::CouenneThreeWayBranchObj:

Collaboration diagram for Couenne::CouenneThreeWayBranchObj:

Public Member Functions

- CouenneThreeWayBranchObj (**JnlstPtr** jnlst, expression *, CouNumber, CouNumber, int=THREE_CENTER)

  Constructor.
- CouenneThreeWayBranchObj (const CouenneThreeWayBranchObj &src)

  Copy constructor.
- virtual **OsiBranchingObject** * clone () const

  Cloning method.
- virtual double branch (**OsiSolverInterface** *solver=NULL)

  Execute the actions required to branch, as specified by the current state of the branching object, and advance the object's state.

Protected Attributes

- expression * brVar_

  The variable this branching object refers to.
- CouNumber lcrop_

  left divider
- CouNumber rcrop_

  right divider
- int firstBranch_

  First branch to be performed: 0 is left, 1 is central, 2 is right.
- **JnlstPtr** jnlst_

  True if the associated variable is integer.

6.47.1 Detailed Description

Spatial, three-way branching object.

Branching is performed on continuous variables but a better convexification is sought around the current point by dividing the interval in three parts

Definition at line 28 of file CouenneThreeWayBranchObj.hpp.

6.47.2 Member Function Documentation

6.47.2.1 virtual double Couenne::CouenneThreeWayBranchObj::branch (**OsiSolverInterface** * *solver* = NULL) [virtual]

Execute the actions required to branch, as specified by the current state of the branching object, and advance the object's state.

Returns change in guessed objective on next (what does "next" mean here?) branch

Implements **OsiBranchingObject**.

6.47.3 Member Data Documentation

6.47.3.1 expression* Couenne::CouenneThreeWayBranchObj::brVar_ [protected]

The variable this branching object refers to.

If the corresponding CouenneObject was created on w=f(x,y), it is either x or y.

Definition at line 67 of file CouenneThreeWayBranchObj.hpp.

6.47.3.2 JnlstPtr Couenne::CouenneThreeWayBranchObj::jnlst_ [protected]

True if the associated variable is integer.

SmartPointer to the Journalist

Definition at line 79 of file CouenneThreeWayBranchObj.hpp.

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

- CouenneThreeWayBranchObj.hpp

6.48 Couenne::CouenneTNLP Class Reference

Class for handling NLPs using CouenneProblem.

`#include <CouenneTNLP.hpp>`

Inheritance diagram for Couenne::CouenneTNLP:

Collaboration diagram for Couenne::CouenneTNLP:

Public Member Functions

- CouenneTNLP ()
  Empty constructor.
- CouenneTNLP (CouenneProblem *)
  Constructor.
- CouenneTNLP (const CouenneTNLP &)
  Copy constructor.
- CouenneTNLP & operator= (const CouenneTNLP &rhs)
  Assignment.
- CouenneTNLP * clone ()
  Clone.
- virtual ~CouenneTNLP ()
  Destructor.
- void setInitSol (const double *sol)
  set initial solution
- CouNumber * getSolution ()
  returns best solution (if it exists)
- CouNumber getSolValue ()
  returns value of the best solution

- virtual bool get_nlp_info (Ipopt::Index &n, Ipopt::Index &m, Ipopt::Index &nnz_jac_g, Ipopt::Index &nnz_h_lag, enum **Ipopt::TNLP::IndexStyleEnum** &index_style)

  return the number of variables and constraints, and the number of non-zeros in the jacobian and the hessian.
- virtual bool get_bounds_info (Ipopt::Index n, Ipopt::Number ∗x_l, Ipopt::Number ∗x_u, Ipopt::Index m, Ipopt::Number ∗g_l, Ipopt::Number ∗g_u)

  return the information about the bound on the variables and constraints.
- virtual bool get_variables_linearity (Ipopt::Index n, **Ipopt::TNLP::LinearityType** ∗var_types)

  return the variables linearity (TNLP::Linear or TNLP::NonLinear).
- virtual bool get_constraints_linearity (Ipopt::Index m, **Ipopt::TNLP::LinearityType** ∗const_types)

  return the constraint linearity.
- virtual bool get_starting_point (Ipopt::Index n, bool init_x, Ipopt::Number ∗x, bool init_z, Ipopt::Number ∗z_L, Ipopt::Number ∗z_U, Ipopt::Index m, bool init_lambda, Ipopt::Number ∗lambda)

  return the starting point.
- virtual bool eval_f (Ipopt::Index n, const Ipopt::Number ∗x, bool new_x, Ipopt::Number &obj_value)

  return the value of the objective function
- virtual bool eval_grad_f (Ipopt::Index n, const Ipopt::Number ∗x, bool new_x, Ipopt::Number ∗grad_f)

  return the vector of the gradient of the objective w.r.t. x
- virtual bool eval_g (Ipopt::Index n, const Ipopt::Number ∗x, bool new_x, Ipopt::Index m, Ipopt::Number ∗g)

  return the vector of constraint values
- virtual bool eval_jac_g (Ipopt::Index n, const Ipopt::Number ∗x, bool new_x, Ipopt::Index m, Ipopt::Index nele_jac, Ipopt::Index ∗iRow, Ipopt::Index ∗jCol, Ipopt::Number ∗values)

  return the jacobian of the constraints.
- virtual bool eval_h (Ipopt::Index n, const Ipopt::Number ∗x, bool new_x, Ipopt::Number obj_factor, Ipopt::Index m, const Ipopt::Number ∗lambda, bool new_lambda, Ipopt::Index nele_hess, Ipopt::Index ∗iRow, Ipopt::Index ∗jCol, Ipopt::Number ∗values)

  return the hessian of the lagrangian.
- virtual void finalize_solution (Ipopt::SolverReturn status, Ipopt::Index n, const Ipopt::Number ∗x, const Ipopt::Number ∗z_L, const Ipopt::Number ∗z_U, Ipopt::Index m, const Ipopt::Number ∗g, const Ipopt::Number ∗lambda, Ipopt::Number obj_value, const **Ipopt::IpoptData** ∗ip_data, **Ipopt::IpoptCalculatedQuantities** ∗ip_cq)

  This method is called when the algorithm is complete so the TNLP can store/write the solution.
- virtual bool intermediate_callback (Ipopt::AlgorithmMode mode, Ipopt::Index iter, Ipopt::Number obj_value, Ipopt::Number inf_pr, Ipopt::Number inf_du, Ipopt::Number mu, Ipopt::Number d_norm, Ipopt::Number regularization_size, Ipopt::Number alpha_du, Ipopt::Number alpha_pr, Ipopt::Index ls_trials, const **Ipopt::IpoptData** ∗ip_data, **Ipopt::IpoptCalculatedQuantities** ∗ip_cq)

  Intermediate Callback method for the user.

Methods for quasi-Newton approximation. If the second

derivatives are approximated by **Ipopt**, it is better to do this only in the space of nonlinear variables.

The following methods are call by **Ipopt** if the quasi-Newton approximation is selected. If -1 is returned as number of nonlinear variables, **Ipopt** assumes that all variables are nonlinear. Otherwise, it calls get_list_of_nonlinear_variables with an array into which the indices of the nonlinear variables should be written - the array has the lengths num_nonlin_vars, which is identical with the return value of get_number_of_nonlinear_variables (). It is assumed that the indices are counted starting with 1 in the FORTRAN_STYLE, and 0 for the C_STYLE.

- virtual Ipopt::Index **get_number_of_nonlinear_variables** ()
- virtual bool get_list_of_nonlinear_variables (Ipopt::Index num_nonlin_vars, Ipopt::Index ∗pos_nonlin_vars)

  get real list
- virtual void setObjective (expression ∗newObj)

Change objective function and modify gradient expressions accordingly.

- CouenneSparseMatrix *& optHessian ()

  Get methods.

- bool & getSaveOptHessian ()

  *set and get saveOptHessian_*

6.48.1 Detailed Description

Class for handling NLPs using CouenneProblem.

Definition at line 27 of file CouenneTNLP.hpp.

6.48.2 Member Function Documentation

6.48.2.1 virtual bool Couenne::CouenneTNLP::get_nlp_info (Ipopt::Index & *n*, Ipopt::Index & *m*, Ipopt::Index & *nnz_jac_g*, Ipopt::Index & *nnz_h_lag*, enum **Ipopt::TNLP::IndexStyleEnum** & *index_style*) [virtual]

return the number of variables and constraints, and the number of non-zeros in the jacobian and the hessian.

The index_style parameter lets you specify C or Fortran style indexing for the sparse matrix iRow and jCol parameters. C_STYLE is 0-based, and FORTRAN_STYLE is 1-based.

6.48.2.2 virtual bool Couenne::CouenneTNLP::get_bounds_info (Ipopt::Index *n*, Ipopt::Number * *x_l*, Ipopt::Number * *x_u*, Ipopt::Index *m*, Ipopt::Number * *g_l*, Ipopt::Number * *g_u*) [virtual]

return the information about the bound on the variables and constraints.

The value that indicates that a bound does not exist is specified in the parameters nlp_lower_bound_inf and nlp_↩ upper_bound_inf. By default, nlp_lower_bound_inf is -1e19 and nlp_upper_bound_inf is 1e19. (see TNLPAdapter)

6.48.2.3 virtual bool Couenne::CouenneTNLP::get_variables_linearity (Ipopt::Index *n*, **Ipopt::TNLP::LinearityType** * *var_types*) [virtual]

return the variables linearity (TNLP::Linear or TNLP::NonLinear).

The var_types array should be allocated with length at least n. (default implementation just return false and does not fill the array).

6.48.2.4 virtual bool Couenne::CouenneTNLP::get_constraints_linearity (Ipopt::Index *m*, **Ipopt::TNLP::LinearityType** * *const_types*) [virtual]

return the constraint linearity.

array should be alocated with length at least n. (default implementation just return false and does not fill the array).

6.48.2.5 virtual bool Couenne::CouenneTNLP::get_starting_point (Ipopt::Index *n*, bool *init_x*, Ipopt::Number * *x*, bool *init_z*, Ipopt::Number * *z_L*, Ipopt::Number * *z_U*, Ipopt::Index *m*, bool *init_lambda*, Ipopt::Number * *lambda*) [virtual]

return the starting point.

The bool variables indicate whether the algorithm wants you to initialize x, z_L/z_u, and lambda, respectively. If, for some reason, the algorithm wants you to initialize these and you cannot, return false, which will cause **Ipopt** to stop. You will have to run **Ipopt** with different options then.

6.48.2.6 virtual bool Couenne::CouenneTNLP::eval_jac_g (Ipopt::Index *n,* const Ipopt::Number * *x,* bool *new_x,* Ipopt::Index *m,* Ipopt::Index *nele_jac,* Ipopt::Index * *iRow,* Ipopt::Index * *jCol,* Ipopt::Number * *values*) [virtual]

return the jacobian of the constraints.

The vectors iRow and jCol only need to be set once. The first call is used to set the structure only (iRow and jCol will be non-NULL, and values will be NULL) For subsequent calls, iRow and jCol will be NULL.

6.48.2.7 virtual bool Couenne::CouenneTNLP::eval_h (Ipopt::Index *n,* const Ipopt::Number * *x,* bool *new_x,* Ipopt::Number *obj_factor,* Ipopt::Index *m,* const Ipopt::Number * *lambda,* bool *new_lambda,* Ipopt::Index *nele_hess,* Ipopt::Index * *iRow,* Ipopt::Index * *jCol,* Ipopt::Number * *values*) [virtual]

return the hessian of the lagrangian.

The vectors iRow and jCol only need to be set once (during the first call). The first call is used to set the structure only (iRow and jCol will be non-NULL, and values will be NULL) For subsequent calls, iRow and jCol will be NULL. This matrix is symmetric - specify the lower diagonal only. A default implementation is provided, in case the user wants to se quasi-Newton approximations to estimate the second derivatives and doesn't not neet to implement this method.

6.48.2.8 virtual bool Couenne::CouenneTNLP::intermediate_callback (Ipopt::AlgorithmMode *mode,* Ipopt::Index *iter,* Ipopt::Number *obj_value,* Ipopt::Number *inf_pr,* Ipopt::Number *inf_du,* Ipopt::Number *mu,* Ipopt::Number *d_norm,* Ipopt::Number *regularization_size,* Ipopt::Number *alpha_du,* Ipopt::Number *alpha_pr,* Ipopt::Index *ls_trials,* const **Ipopt::IpoptData** * *ip_data,* **Ipopt::IpoptCalculatedQuantities** * *ip_cq*) [virtual]

Intermediate Callback method for the user.

Providing dummy default implementation. For details see IntermediateCallBack in **IpNLP.hpp**.

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

- CouenneTNLP.hpp

6.49 Couenne::CouenneTwoImplied Class Reference

Cut Generator for implied bounds derived from pairs of linear (in)equalities.

`#include <CouenneTwoImplied.hpp>`

Inheritance diagram for Couenne::CouenneTwoImplied:

Collaboration diagram for Couenne::CouenneTwoImplied:

Public Member Functions

- CouenneTwoImplied (CouenneProblem *, **JnlstPtr**, const **Ipopt::SmartPtr**< **Ipopt::OptionsList** >)

  constructor
- CouenneTwoImplied (const CouenneTwoImplied &)

  copy constructor

- ~CouenneTwoImplied ()

  destructor
- CouenneTwoImplied * clone () const

  clone method (necessary for the abstract **CglCutGenerator** *class)*
- void generateCuts (const **OsiSolverInterface** &, **OsiCuts** &, const **CglTreeInfo**=**CglTreeInfo**()) const

  the main **CglCutGenerator**

Static Public Member Functions

- static void registerOptions (**Ipopt::SmartPtr**< Bonmin::RegisteredOptions > roptions)

  Add list of options to be read from file.

Protected Attributes

- CouenneProblem * problem_

  pointer to problem data structure (used for post-BT)
- **JnlstPtr** jnlst_

  Journalist.
- int nMaxTrials_

  maximum number of trials in every call
- double totalTime_

  Total CPU time spent separating cuts.
- double totalInitTime_

  CPU time spent columning the row formulation.
- bool firstCall_

  first call indicator
- int depthLevelling_

  Depth of the BB tree where to start decreasing chance of running this.
- int depthStopSeparate_

  Depth of the BB tree where stop separation.

6.49.1 Detailed Description

Cut Generator for implied bounds derived from pairs of linear (in)equalities.

Implied bounds usually work on a SINGLE inequality of the form

$\ell_j \leq \sum_{i \in N_+} a_{ji} x_i + \sum_{i \in N_-} a_{ji} x_i \leq u_j$

where $a_{ji} > 0$ for $i \in N_+$ and $a_{ji} < 0$ for $i \in N_-$, and allow one to infer better bounds $[x_i^L, x_i^U]$ on all variables with nonzero coefficients:

(1) $x_i^L \geq (\ell_j - \sum_{i \in N_+} a_{ji} x_i^U - \sum_{i \in N_-} a_{ji} x_i^L)/a_{ji} \qquad \forall i \in N_+$

(2) $x_i^U \leq (u_j - \sum_{i \in N_+} a_{ji} x_i^L - \sum_{i \in N_-} a_{ji} x_i^U)/a_{ji} \qquad \forall i \in N_+$

(3) $x_i^L \geq (u_j - \sum_{i \in N_+} a_{ji} x_i^L - \sum_{i \in N_-} a_{ji} x_i^U)/a_{ji} \qquad \forall i \in N_-$

(4) $x_i^U \leq (\ell_j - \sum_{i \in N_+} a_{ji} x_i^U - \sum_{i \in N_-} a_{ji} x_i^L)/a_{ji} \qquad \forall i \in N_+$

Consider now two inequalities:

$\ell_h \leq \sum_{i \in N^1_+} a_{hi} x_i + \sum_{i \in N^1_-} a_{hi} x_i \leq u_h$

$\ell_k \leq \sum_{i \in N^2_+} a_{ki} x_i + \sum_{i \in N^2_-} a_{ki} x_i \leq u_k$

and their CONVEX combination using α and $1 - \alpha$, where $\alpha \in [0, 1]$:

$\ell' \leq \sum_{i \in N} b_i x_i \leq u'$

with $N = N^1_+ \cup N^1_- \cup N^2_+ \cup N^2_-$, $\ell' = \alpha \ell_h + (1 - \alpha)\ell_k$, and $u' = \alpha u_h + (1 - \alpha) u_k$. As an example where this might be useful, consider

$x + y \geq 2$

$x - y \geq 1$

with $x \in [0, 4]$ and $y \in [0, 1]$. (This is similar to an example given in Tawarmalani and Sahinidis to explain FBBT != OBBT, I believe.) The sum of the two above inequalities gives $x \geq 1.5$, while using only the implied bounds on the single inequalities gives $x \geq 1$.

The key consideration here is that the b_i coefficients, ℓ' , and u' are functions of α , which determines which, among (1)-(4), to apply. In general,

if $b_i > 0$ then

$x^L_i \geq (l' - \sum_{j \in N'_+} b_j x^U_j - \sum_{j \in N'_-} b_j x^L_j)/b_i$,

$x^U_i \leq (u' - \sum_{j \in N'_+} b_j x^L_j - \sum_{j \in N'_-} b_j x^U_j)/b_i$;

if $b_i < 0$ then

$x^L_i \geq (l' - \sum_{j \in N'_+} b_j x^U_j - \sum_{j \in N'_-} b_j x^L_j)/b_i$,

$x^U_i \leq (u' - \sum_{j \in N'_+} b_j x^L_j - \sum_{j \in N'_-} b_j x^U_j)/b_i$.

Each lower/upper bound is therefore a piecewise rational function of α , given that b_i and the content of N'_+ and N'_- depend on α . These functions are continuous (easy to prove) but not differentiable at some points of $[0, 1]$.

The purpose of this procedure is to find the maximum of the lower bounding function and the minimum of the upper bounding function.

Divide the interval $[0, 1]$ into at most $m + 1$ intervals (where m is the number of coefficients not identically zero, or the number of b_i that are nonzero for at least one value of α). The limits c_i of the subintervals are the zeros of each coefficient, i.e. the values of α such that $\alpha a_{ki} + (1 - \alpha) a_{hi} = 0$, or $c_i = \frac{-a_{hi}}{a_{ki} - a_{hi}}$.

Sorting these values gives us something to do on every interval $[c_j, c_{j+1}]$ when computing a new value of x^L_i and x^U_i , which I'll denote L_i and U_i in the following.

0) if $c_j = c_i$ then

- compute $VL = \lim_{c_j \to \alpha} L_i(\alpha)$
- if $= +\infty$, infeasible else compute derivative DL (should be $+\infty$)

1) else

- compute $VL = \lim_{\alpha \to c_j} L_i(\alpha)$ (can be retrieved from previous interval as $L_i(\alpha)$ is continuous)
- compute $DL = \lim_{\alpha \to c_j} dL_i(\alpha)$

update x^L with VL if necessary.

2) if $c_{j+1} = c_i$ then

- compute $VR = \lim_{\alpha \to c_{j+1}} L_i(\alpha)$

- if = $+\infty$, infeasible else compute derivative DR (should be $-\infty$)

3) else

- compute $VR = \lim_{\alpha \to c_{j+1}} L_i(\alpha)$
- compute $DR = \lim_{\alpha \to c_{j+1}} dL_i(\alpha)$

update x^L with VR if necessary.

if $DL > 0$ and $DR < 0$, there might be a maximum in between, otherwise continue to next interval

compute internal maximum VI, update x^L with VI if necessary.

Apply a similar procedure for the upper bound

This should be applied for any h, k, i , therefore we might have a lot to do. First, select possible pairs (h, k) among those for which there exists at least one variable that satisfies neither of the following conditions:

a) same sign coefficient, constraints (h, k) both $\geq$ or both $\leq$

b) opposite sign coefficient, constraints (h, k) $(\leq, \geq)$ or $(\geq, \leq)$

as in those cases, no c_i would be in $[0, 1]$

Definition at line 174 of file CouenneTwoImplied.hpp.

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

- CouenneTwoImplied.hpp

6.50 Couenne::CouenneUserInterface Class Reference

Inheritance diagram for Couenne::CouenneUserInterface:

Collaboration diagram for Couenne::CouenneUserInterface:

Public Member Functions

- virtual bool setupJournals ()

  Setup journals for printing.
- virtual CouenneProblem $*$ getCouenneProblem ()=0

  Should return the problem to solve in algebraic form.
- virtual **Ipopt::SmartPtr**$<$ Bonmin::TMINLP $>$ getTMINLP ()=0

  Should return the problem to solve as TMINLP.
- virtual bool addBabPlugins (Bonmin::Bab &bab)

  Called after B&B object is setup.
- virtual bool writeSolution (Bonmin::Bab &bab)

  Called after B&B finished.

6.50.1 Detailed Description

Definition at line 32 of file CouenneUserInterface.hpp.

6.50.2 Member Function Documentation

6.50.2.1 virtual bool Couenne::CouenneUserInterface::setupJournals ( ) [inline], [virtual]

Setup journals for printing.

Default is to have one journal that prints to stdout.

Definition at line 47 of file CouenneUserInterface.hpp.

6.50.2.2 virtual CouenneProblem* Couenne::CouenneUserInterface::getCouenneProblem ( ) [pure virtual]

Should return the problem to solve in algebraic form.

NOTE: Couenne is (currently) going to modify this problem!

Implemented in Couenne::CouenneOSInterface, and Couenne::CouenneAmplInterface.

6.50.2.3 virtual bool Couenne::CouenneUserInterface::addBabPlugins (Bonmin::Bab & *bab*) [inline], [virtual]

Called after B&B object is setup.

User should add plugins like cut generators, bound tighteners, or heuristics here.

Definition at line 65 of file CouenneUserInterface.hpp.

6.50.2.4 virtual bool Couenne::CouenneUserInterface::writeSolution (Bonmin::Bab & *bab*) [inline], [virtual]

Called after B&B finished.

Should write solution information.

Reimplemented in Couenne::CouenneOSInterface, and Couenne::CouenneAmplInterface.

Definition at line 79 of file CouenneUserInterface.hpp.

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

- CouenneUserInterface.hpp

6.51 Couenne::CouenneVarObject Class Reference

OsiObject for variables in a MINLP.

`#include <CouenneVarObject.hpp>`

Inheritance diagram for Couenne::CouenneVarObject:

Collaboration diagram for Couenne::CouenneVarObject:

Public Member Functions

- CouenneVarObject (CouenneCutGenerator *c, CouenneProblem *p, exprVar *ref, Bonmin::BabSetupBase *base, **JnlstPtr** jnlst, int varSelection)

  Constructor with information for branching point selection strategy.
- CouenneVarObject (const CouenneVarObject &src)

Copy constructor.

- ~CouenneVarObject ()

  Destructor.

- virtual CouenneObject * clone () const

  Cloning method.

- virtual double infeasibility (const **OsiBranchingInformation** *info, int &way) const

  compute infeasibility of this variable x as the sum/min/max of all infeasibilities of auxiliaries w whose defining function depends on x |w - f(x)|

- virtual double checkInfeasibility (const **OsiBranchingInformation** *info) const

  compute infeasibility of this variable, |w - f(x)|, where w is the auxiliary variable defined as w = f(x)

- virtual **OsiBranchingObject** * createBranch (**OsiSolverInterface** *, const **OsiBranchingInformation** *, int) const

  create CouenneBranchingObject or CouenneThreeWayBranchObj based on this object

- virtual double feasibleRegion (**OsiSolverInterface** *, const **OsiBranchingInformation** *) const

  fix nonlinear coordinates of current integer-nonlinear feasible solution

- virtual bool isCuttable () const

  are we on the bad or good side of the expression?

Protected Member Functions

- CouNumber computeBranchingPoint (const **OsiBranchingInformation** *info, int &bestWay, const CouenneObject *&criticalObject) const

  Method computing the branching point.

Protected Attributes

- int varSelection_

  branching scheme used.

Additional Inherited Members

6.51.1 Detailed Description

OsiObject for variables in a MINLP.

Definition at line 22 of file CouenneVarObject.hpp.

6.51.2 Member Function Documentation

6.51.2.1 virtual double Couenne::CouenneVarObject::infeasibility (const OsiBranchingInformation * *info*, int & *way*) const [virtual]

compute infeasibility of this variable x as the sum/min/max of all infeasibilities of auxiliaries w whose defining function depends on x |w - f(x)|

TODO: suggest way

Reimplemented from Couenne::CouenneObject.

Reimplemented in Couenne::CouenneVTObject.

6.51.3 Member Data Documentation

6.51.3.1 int Couenne::CouenneVarObject::varSelection_ [protected]

branching scheme used.

Experimental: still figuring out why plain LP branching doesn't work with strong/reliability branching

Definition at line 73 of file CouenneVarObject.hpp.

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

- CouenneVarObject.hpp

6.52 Couenne::CouenneVTObject Class Reference

OsiObject for violation transfer on variables in a MINLP.

`#include <CouenneVTObject.hpp>`

Inheritance diagram for Couenne::CouenneVTObject:

Collaboration diagram for Couenne::CouenneVTObject:

Public Member Functions

- CouenneVTObject (CouenneCutGenerator *c, CouenneProblem *p, exprVar *ref, Bonmin::BabSetupBase *base, **JnlstPtr** jnlst, int varSelection)

  Constructor with information for branching point selection strategy.
- CouenneVTObject (const CouenneVTObject &src)

  Copy constructor.
- ~CouenneVTObject ()

  Destructor.
- virtual CouenneObject * clone () const

  Cloning method.
- virtual double infeasibility (const **OsiBranchingInformation** *info, int &way) const

  compute infeasibility of this variable x as the sum/min/max of all infeasibilities of auxiliaries w whose defining function depends on x |w - f(x)|

Additional Inherited Members

6.52.1 Detailed Description

OsiObject for violation transfer on variables in a MINLP.

Definition at line 19 of file CouenneVTObject.hpp.

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

- CouenneVTObject.hpp

6.53 Couenne::CouExpr Class Reference

6.53.1 Detailed Description

Definition at line 17 of file CouExpr.hpp.

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

- CouExpr.hpp

6.54 Couenne::DepGraph Class Reference

Dependence graph.

`#include <CouenneDepGraph.hpp>`

Collaboration diagram for Couenne::DepGraph:

Public Member Functions

- DepGraph ()
  constructor
- ~DepGraph ()
  destructor
- std::set< DepNode *, compNode > & Vertices ()
  return vertex set
- int & Counter ()
  node index counter
- void insert (exprVar *)
  insert new variable if new
- void insert (exprAux *)
  insert new auxiliary if new
- void erase (exprVar *)
  delete element
- bool depends (int, int, bool=false)
  does w depend on x?
- void createOrder ()
  assign numbering to all nodes of graph
- void print (bool descend=false)
  debugging procedure
- DepNode * lookup (int index)
  search for node in vertex set
- bool checkCycles ()
  check for dependence cycles in graph
- void replaceIndex (int oldVar, int newVar)
  replace, throughout the whole graph, the index of a variable with another in the entire graph.

Protected Attributes

- std::set< DepNode *, compNode > vertices_
  set of variable nodes
- int counter_
  counter to assign numbering to all nodes

6.54.1 Detailed Description

Dependence graph.

Shows dependence of auxiliary variable on other (auxiliary and/or original) variables

Definition at line 115 of file CouenneDepGraph.hpp.

6.54.2 Member Function Documentation

6.54.2.1 void Couenne::DepGraph::replaceIndex (int *oldVar,* int *newVar*)

replace, throughout the whole graph, the index of a variable with another in the entire graph.

Used when redundant constraints w := x are discovered

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

- CouenneDepGraph.hpp

6.55 Couenne::DepNode Class Reference

vertex of a dependence graph.

`#include <CouenneDepGraph.hpp>`

Collaboration diagram for Couenne::DepNode:

Public Types

- enum dep_color
  color used in DFS for checking cycles

Public Member Functions

- DepNode (int ind)
  fictitious constructor: only fill in index (such object is used in find() and then discarded)
- ~DepNode ()
  destructor
- int Index () const
  return index of this variable
- int Order () const
  return index of this variable

- std::set< DepNode *, compNode > * DepList () const

  return all variables it depends on
- bool depends (int xi, bool=false, std::set< DepNode *, compNode > *already_visited=NULL) const

  does this variable depend on variable with index xi?
- void createOrder (DepGraph *)

  assign numbering to all nodes of graph
- void print (int=0, bool descend=false) const

  debugging procedure
- enum dep_color & color ()

  return or set color of a node
- std::set< DepNode *, compNode > * depList ()

  index nodes on which this one depends (forward star in dependence graph)
- void replaceIndex (DepNode *oldVarNode, DepNode *newVarNode)

  replace the index of a variable with another in the entire graph.

Protected Attributes

- int index_

  index of variable associated with node
- std::set< DepNode *, compNode > * depList_

  index nodes on which this one depends (forward star in dependence graph)
- int order_

  order in which this variable should be updated, evaluated, etc.
- enum dep_color color_

  color used in DFS for checking cycles

6.55.1 Detailed Description

vertex of a dependence graph.

Contains variable and its forward star (all variables it depends on)

Definition at line 33 of file CouenneDepGraph.hpp.

6.55.2 Member Function Documentation

6.55.2.1 void Couenne::DepNode::replaceIndex (DepNode * *oldVarNode,* DepNode * *newVarNode*)

replace the index of a variable with another in the entire graph.

Used when redundant constraints w := x are discovered

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

- CouenneDepGraph.hpp

6.56 Couenne::Domain Class Reference

Define a dynamic point+bounds, with a way to save and restore previous points+bounds through a LIFO structure.

`#include <CouenneDomain.hpp>`

Collaboration diagram for Couenne::Domain:

Public Member Functions

- Domain ()
  basic constructor
- Domain (const Domain &src)
  copy constructor
- ~Domain ()
  destructor
- void push (int dim, CouNumber *x, CouNumber *lb, CouNumber *ub, bool copy=true)
  save current point and start using another
- void push (int dim, const CouNumber *x, const CouNumber *lb, const CouNumber *ub, bool copy=true)
  save current point and start using another
- void push (const **OsiSolverInterface** *si, **OsiCuts** *cs=NULL, bool copy=true)
  save current point and start using another – retrieve information from solver interface and from previous column cuts
- void push (const DomainPoint &dp, bool copy=true)
  save current point and start using another
- void pop ()
  restore previous point
- DomainPoint * current ()
  return current point
- CouNumber & x (register int index)
  current variable
- CouNumber & lb (register int index)
  current lower bound
- CouNumber & ub (register int index)
  current upper bound
- CouNumber * x ()
  return current variable vector
- CouNumber * lb ()
  return current lower bound vector
- CouNumber * ub ()
  return current upper bound vector

Protected Attributes

- DomainPoint * point_
  current point
- std::stack< DomainPoint * > domStack_
  stack of saved points

6.56.1 Detailed Description

Define a dynamic point+bounds, with a way to save and restore previous points+bounds through a LIFO structure.

Definition at line 104 of file CouenneDomain.hpp.

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

- CouenneDomain.hpp

6.57 Couenne::DomainPoint Class Reference

Define a point in the solution space and the bounds around it.

`#include <CouenneDomain.hpp>`

Public Member Functions

- DomainPoint (int dim, CouNumber *x, CouNumber *lb, CouNumber *ub, bool copy=true)
  constructor
- DomainPoint (int dim=0, const CouNumber *x=NULL, const CouNumber *lb=NULL, const CouNumber *ub=NULL, bool copy=true)
  constructor
- ~DomainPoint ()
  destructor
- DomainPoint (const DomainPoint &src)
  copy constructor
- void resize (int newdim)
  resize domain point (for extending into higher space)
- int size () const
  return current size
- int Dimension ()
  *return dimension_*
- CouNumber & x (register int index)
  return current variable
- CouNumber & lb (register int index)
  return current lower bound
- CouNumber & ub (register int index)
  return current upper bound
- CouNumber * x ()
  return current variable vector
- CouNumber * lb ()
  return current lower bound vector
- CouNumber * ub ()
  return current upper bound vector
- DomainPoint & operator= (const DomainPoint &src)
  assignment operator
- bool & isNlp ()
  true if this point is the nlp solution

Protected Attributes

- int dimension_

  dimension of point
- CouNumber * x_

  current value of variables
- CouNumber * lb_

  lower bound
- CouNumber * ub_

  upper bound
- bool copied_

  true if data has been copied (so we own it, and have to delete it upon destruction)
- bool isNlp_

  true if this point comes from an NLP solver (and is thus nlp feasible)

6.57.1 Detailed Description

Define a point in the solution space and the bounds around it.

Definition at line 30 of file CouenneDomain.hpp.

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

- CouenneDomain.hpp

6.58 Couenne::exprAbs Class Reference

class for $|f(x)|$

`#include <CouenneExprAbs.hpp>`

Inheritance diagram for Couenne::exprAbs:

Collaboration diagram for Couenne::exprAbs:

Public Member Functions

- exprAbs (expression *al)

  Constructor.
- unary_function F ()

  The operator's function.
- expression * clone (Domain *d=NULL) const

  cloning method
- std::string printOp () const

  output
- CouNumber gradientNorm (const double *x)

  return l_2 norm of gradient at given point
- expression * differentiate (int index)

  differentiation
- virtual void getBounds (expression *&, expression *&)

Get lower and upper bound of an expression (if any)

- virtual void getBounds (CouNumber &lb, CouNumber &ub)

  Get value of lower and upper bound of an expression (if any)

- void generateCuts (expression *w, **OsiCuts** &cs, const CouenneCutGenerator *cg, t_chg_bounds *=NULL, int=-1, CouNumber=-COUENNE_INFINITY, CouNumber=COUENNE_INFINITY)

  *generate equality between *this and *w*

- enum expr_type code ()

  code for comparisons

- bool isInteger ()

  is this expression integer?

- bool impliedBound (int, CouNumber *, CouNumber *, t_chg_bounds *, enum auxSign=expression::AUX_EQ)

  implied bound processing

- virtual CouNumber selectBranch (const CouenneObject *obj, const **OsiBranchingInformation** *info, expression *&var, double *&brpts, double *&brDist, int &way)

  set up branching object by evaluating many branching points for each expression's arguments

- virtual void closestFeasible (expression *varind, expression *vardep, CouNumber &left, CouNumber &right) const

  closest feasible points in function in both directions

- virtual bool isCuttable (CouenneProblem *problem, int index) const

  can this expression be further linearized or are we on its concave ("bad") side

Additional Inherited Members

6.58.1 Detailed Description

class for $|f(x)|$

Definition at line 23 of file CouenneExprAbs.hpp.

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

- CouenneExprAbs.hpp

6.59 Couenne::exprAux Class Reference

Auxiliary variable.

`#include <CouenneExprAux.hpp>`

Inheritance diagram for Couenne::exprAux:

Collaboration diagram for Couenne::exprAux:

Public Types

- enum intType

  integrality type of an auxiliary variable: unset, continuous, integer

Public Member Functions

- enum nodeType Type () const

  Node type.
- exprAux (expression *, int, int, intType=Unset, Domain *=NULL, enum auxSign=expression::AUX_EQ)

  Constructor.
- exprAux (expression *, Domain *=NULL, enum auxSign=expression::AUX_EQ)

  Constructor to be used with standardize ([...], false)
- virtual ~exprAux ()

  Destructor.
- exprAux (const exprAux &, Domain *d=NULL)

  Copy constructor.
- virtual exprVar * clone (Domain *d=NULL) const

  Cloning method.
- expression * Lb ()

  get lower bound expression
- expression * Ub ()

  get upper bound expression
- virtual void print (std::ostream &=std::cout, bool=false) const

  Print expression.
- expression * Image () const

  The expression associated with this auxiliary variable.
- void Image (expression *image)

  Sets expression associated with this auxiliary variable.
- CouNumber operator() ()

  Null function for evaluating the expression.
- int DepList (std::set< int > &deplist, enum dig_type type=ORIG_ONLY)

  fill in the set with all indices of variables appearing in the expression
- expression * simplify ()

  simplify
- int Linearity ()

  Get a measure of "how linear" the expression is (see CouenneTypes.h)
- void crossBounds ()

  Get lower and upper bound of an expression (if any)
- void generateCuts (**OsiCuts** &, const CouenneCutGenerator *, t_chg_bounds *=NULL, int=-1, CouNumber=-COUENNE_INFINITY, CouNumber=COUENNE_INFINITY)

  generate cuts for expression associated with this auxiliary
- virtual int rank ()

  used in rank-based branching variable choice
- virtual bool isDefinedInteger ()

  is this expression defined as integer?
- virtual bool isInteger ()

  is this expression integer?
- virtual void setInteger (bool value)

  Set this variable as integer.
- void increaseMult ()

  Tell this variable appears once more.

- void decreaseMult ()

  Tell this variable appears once less (standardized within exprSum, for instance)

- void zeroMult ()

  Disable this auxiliary variable.

- int Multiplicity ()

  How many times this variable appears.

- void linkDomain (Domain *d)

  link this variable to a domain

- bool & top_level ()

  *return top_level_*

- CouenneObject * properObject (CouenneCutGenerator *c, CouenneProblem *p, Bonmin::BabSetupBase *base, **JnlstPtr** jnlst)

  return proper object to handle expression associated with this variable (NULL if this is not an auxiliary)

- virtual enum auxSign sign () const

  return its sign in the definition constraint

Protected Attributes

- expression * image_

  The expression associated with this auxiliary variable.

- expression * lb_

  lower bound, a function of the associated expression and the bounds on the variables in the expression

- expression * ub_

  upper bound, a function of the associated expression and the bounds on the variables in the expression

- int rank_

  used in rank-based branching variable choice: original variables have rank 1; auxiliary w=f(x) has rank r(w) = r(x)+1; finally, auxiliary w=f(x1,x2...,xk) has rank r(w) = 1+max{r(xi):i=1..k}.

- int multiplicity_

  number of appearances of this aux in the formulation.

- enum intType integer_

  is this variable integer?

- bool top_level_

  True if this variable replaces the lhs of a constraint, i.e., if it is a top level variable in the DAG of the problem.

- enum auxSign sign_

  "sign" of the defining constraint

6.59.1 Detailed Description

Auxiliary variable.

It is associated with an expression which depends, in general, on original and/or other auxiliary variables. It is used for AMPL's defined variables (aka common expressions) and to reformulate nonlinear constraints/objectives.

Definition at line 31 of file CouenneExprAux.hpp.

6.59.2 Member Function Documentation

6.59.2.1 void Couenne::exprAux::crossBounds ( ) [virtual]

Get lower and upper bound of an expression (if any)

set bounds depending on both branching rules and propagated bounds. To be used after standardization

Reimplemented from Couenne::exprVar.

6.59.3 Member Data Documentation

6.59.3.1 int Couenne::exprAux::rank_ [protected]

used in rank-based branching variable choice: original variables have rank 1; auxiliary w=f(x) has rank r(w) = r(x)+1; finally, auxiliary w=f(x1,x2...,xk) has rank r(w) = 1+max{r(xi):i=1..k}.

Definition at line 54 of file CouenneExprAux.hpp.

6.59.3.2 int Couenne::exprAux::multiplicity_ [protected]

number of appearances of this aux in the formulation.

The more times it occurs in the formulation, the more implication its branching has on other variables

Definition at line 59 of file CouenneExprAux.hpp.

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

- CouenneExprAux.hpp

6.60 Couenne::exprBinProd Class Reference

class for $\prod_{i=1}^{n} f_i(x)$ with $f_i(x)$ all binary

`#include <CouenneExprBinProd.hpp>`

Inheritance diagram for Couenne::exprBinProd:

Collaboration diagram for Couenne::exprBinProd:

Public Member Functions

- exprBinProd (expression **, int)
  Constructor.
- exprBinProd (expression *, expression *)
  Constructor with two arguments.
- CouNumber gradientNorm (const double *x)
  return l-2 norm of gradient at given point
- expression * differentiate (int index)
  differentiation
- expression * simplify ()
  simplification

- virtual int Linearity ()

  get a measure of "how linear" the expression is:
- virtual void getBounds (expression *&, expression *&)

  Get lower and upper bound of an expression (if any)
- virtual void getBounds (CouNumber &lb, CouNumber &ub)

  Get value of lower and upper bound of an expression (if any)
- virtual exprAux * standardize (CouenneProblem *p, bool addAux=true)

  reduce expression in standard form, creating additional aux variables (and constraints)
- void generateCuts (expression *w, **OsiCuts** &cs, const CouenneCutGenerator *cg, t_chg_bounds *=NULL, int=-1, CouNumber=-COUENNE_INFINITY, CouNumber=COUENNE_INFINITY)

  *generate equality between *this and *w*
- virtual enum expr_type code ()

  code for comparison
- bool impliedBound (int, CouNumber *, CouNumber *, t_chg_bounds *, enum Couenne::expression::auxSign=Couenne::expression::AUX_EQ)

  implied bound processing
- virtual CouNumber selectBranch (const CouenneObject *obj, const **OsiBranchingInformation** *info, expression *&var, double *&brpts, double *&brDist, int &way)

  set up branching object by evaluating many branching points for each expression's arguments
- virtual void closestFeasible (expression *varind, expression *vardep, CouNumber &left, CouNumber &right) const

  compute y^{lv} and y^{uv} for Violation Transfer algorithm

Protected Member Functions

- CouNumber balancedMul (const **OsiBranchingInformation** *info, int index, int wind)

  balanced strategy for branching point selection in products
- virtual bool isCuttable (CouenneProblem *problem, int index) const

  can this expression be further linearized or are we on its concave ("bad") side

Additional Inherited Members

6.60.1 Detailed Description

class for $\prod_{i=1}^{n} f_i(x)$ with $f_i(x)$ all binary

Definition at line 21 of file CouenneExprBinProd.hpp.

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

- CouenneExprBinProd.hpp

6.61 Couenne::exprCeil Class Reference

class ceiling, $\lceil f(x) \rceil$

`#include <CouenneExprCeil.hpp>`

Inheritance diagram for Couenne::exprCeil:

Collaboration diagram for Couenne::exprCeil:

Public Member Functions

- exprCeil (expression *arg)

  constructor, destructor
- expression * clone (Domain *d=NULL) const

  cloning method
- unary_function F ()

  the operator itself (e.g. sin, log...)
- std::string printOp () const

  print operator
- CouNumber gradientNorm (const double *x)

  return l-2 norm of gradient at given point
- expression * differentiate (int index)

  obtain derivative of expression
- void getBounds (expression *&, expression *&)

  Get lower and upper bound of an expression (if any)
- void getBounds (CouNumber &lb, CouNumber &ub)

  Get value of lower and upper bound of an expression.
- void generateCuts (expression *w, **OsiCuts** &cs, const CouenneCutGenerator *cg, t_chg_bounds *=NULL, int=-1, CouNumber=-COUENNE_INFINITY, CouNumber=COUENNE_INFINITY)

  *generate equality between *this and *w*
- virtual enum expr_type code ()

  code for comparisons
- bool impliedBound (int index, CouNumber *l, CouNumber *u, t_chg_bounds *chg, enum auxSign=expression::AUX_EQ)

  implied bound processing
- virtual CouNumber selectBranch (const CouenneObject *obj, const **OsiBranchingInformation** *info, expression *&var, double *&brpts, double *&brDist, int &way)

  Set up branching object by evaluating many branching points for each expression's arguments.
- virtual void closestFeasible (expression *varind, expression *vardep, CouNumber &left, CouNumber &right) const

  closest feasible points in function in both directions
- virtual bool isCuttable (CouenneProblem *problem, int index) const

  can this expression be further linearized or are we on its concave ("bad") side?

Additional Inherited Members

6.61.1 Detailed Description

class ceiling, $\lceil f(x) \rceil$

Definition at line 20 of file CouenneExprCeil.hpp.

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

- CouenneExprCeil.hpp

6.62 Couenne::exprClone Class Reference

expression clone (points to another expression)

`#include <CouenneExprClone.hpp>`

Inheritance diagram for Couenne::exprClone:

Collaboration diagram for Couenne::exprClone:

Public Member Functions

- exprClone (expression *copy)

  Constructor.
- exprClone (const exprClone &e, Domain *d=NULL)

  copy constructor
- expression * clone (Domain *d=NULL) const

  cloning method
- virtual ~exprClone ()

  Destructor.
- virtual void print (std::ostream &out=std::cout, bool descend=false) const

  Printing.
- CouNumber Value () const

  value
- CouNumber operator() ()

  null function for evaluating the expression

Additional Inherited Members

6.62.1 Detailed Description

expression clone (points to another expression)

Definition at line 24 of file CouenneExprClone.hpp.

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

- CouenneExprClone.hpp

6.63 Couenne::exprConst Class Reference

constant-type operator

`#include <CouenneExprConst.hpp>`

Inheritance diagram for Couenne::exprConst:

Collaboration diagram for Couenne::exprConst:

Public Member Functions

- enum nodeType Type () const

  node type
- CouNumber Value () const

  value of expression
- exprConst (CouNumber value)

  Constructor.
- exprConst (const exprConst &e, Domain *d=NULL)

  Copy constructor.
- virtual expression * clone (Domain *d=NULL) const

  Cloning method.
- void print (std::ostream &out=std::cout, bool=false) const

  I/O.
- CouNumber operator() ()

  return constant's value
- expression * differentiate (int)

  differentiation
- int dependsOn (int *ind, int n, enum dig_type type=STOP_AT_AUX)

  dependence on variable set
- int Linearity ()

  get a measure of "how linear" the expression is (see CouenneTypes.h)
- void getBounds (expression *&lower, expression *&upper)

  Get lower and upper bound of an expression (if any)
- void getBounds (CouNumber &lower, CouNumber &upper)

  Get value of lower and upper bound of an expression (if any)
- void generateCuts (expression *, **OsiCuts** &, const CouenneCutGenerator *, t_chg_bounds *=NULL, int=-1, CouNumber=-COUENNE_INFINITY, CouNumber=COUENNE_INFINITY)

  generate convexification cut for constraint w = this
- virtual enum expr_type code ()

  code for comparisons
- virtual bool isInteger ()

  is this expression integer?
- virtual int rank ()

  used in rank-based branching variable choice

Additional Inherited Members

6.63.1 Detailed Description

constant-type operator

Definition at line 23 of file CouenneExprConst.hpp.

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

- CouenneExprConst.hpp

6.64 Couenne::exprCopy Class Reference

Inheritance diagram for Couenne::exprCopy:

Collaboration diagram for Couenne::exprCopy:

Public Member Functions

- enum nodeType Type () const

  node type
- exprCopy (expression *copy)

  Empty constructor - used in cloning method of exprClone.
- exprCopy (const exprCopy &e, Domain *d=NULL)

  Copy constructor.
- virtual ~exprCopy ()

  Destructor – CAUTION: this is the only destructive destructor, exprClone and exprStore do not destroy anything.
- virtual expression * clone (Domain *d=NULL) const

  Cloning method.
- const expression * Original () const

  If this is an exprClone of a exprClone of an expr???, point to the original expr??? instead of an exprClone – improves computing efficiency.
- bool isaCopy () const

  return true if this is a copy of something, i.e.
- expression * Copy () const

  return copy of this expression (only makes sense in exprCopy)
- expression * Image () const

  return pointer to corresponding expression (for auxiliary variables only)
- int Index () const

  Get variable index in problem.
- int nArgs () const

  Return number of arguments (when applicable, that is, with N-ary functions)
- expression ** ArgList () const

  return arglist (when applicable, that is, with N-ary functions)
- void ArgList (expression **al)

  set arglist (used in deleting nodes without deleting children)
- expression * Argument () const

  return argument (when applicable, i.e., with univariate functions)
- expression ** ArgPtr ()

  return pointer to argument (when applicable, i.e., with univariate functions)
- virtual void print (std::ostream &out=std::cout, bool descend=false) const

  I/O.
- virtual CouNumber Value () const

  value
- virtual CouNumber operator() ()

  null function for evaluating the expression
- CouNumber gradientNorm (const double *x)

  return l-2 norm of gradient at given point
- expression * differentiate (int index)

differentiation

- int DepList (std::set< int > &deplist, enum dig_type type=ORIG_ONLY)

  fill in the set with all indices of variables appearing in the expression
- expression * simplify ()

  simplify expression (useful for derivatives)
- int Linearity ()

  get a measure of "how linear" the expression is (see CouenneTypes.h)
- bool isInteger ()

  is this expression integer?
- virtual bool isDefinedInteger ()

  is this expression DEFINED as integer?
- void getBounds (expression *&lower, expression *&upper)

  Get lower and upper bound of an expression (if any)
- void getBounds (CouNumber &lower, CouNumber &upper)

  Get value of lower and upper bound of an expression (if any)
- exprAux * standardize (CouenneProblem *p, bool addAux=true)

  Create standard formulation of this expression.
- void generateCuts (expression *w, **OsiCuts** &cs, const CouenneCutGenerator *cg, t_chg_bounds *chg=NULL, int wind=-1, CouNumber lb=-COUENNE_INFINITY, CouNumber ub=COUENNE_INFINITY)

  generate convexification cut for constraint w = this
- enum expr_type code ()

  code for comparisons
- enum convexity convexity () const

  either CONVEX, CONCAVE, AFFINE, or NONCONVEX
- int compare (expression &e)

  compare this with other expression
- int rank ()

  used in rank-based branching variable choice
- bool impliedBound (int wind, CouNumber *l, CouNumber *u, t_chg_bounds *chg)

  implied bound processing
- int Multiplicity ()

  multiplicity of a variable: how many times this variable occurs in expressions throughout the problem
- CouNumber selectBranch (const CouenneObject *obj, const **OsiBranchingInformation** *info, expression *&var, double *&brpts, double *&brDist, int &way)

  Set up branching object by evaluating many branching points for each expression's arguments.
- void replace (exprVar *, exprVar *)

  replace occurrence of a variable with another variable
- void fillDepSet (std::set< DepNode *, compNode > *dep, DepGraph *g)

  fill in dependence structure
- void realign (const CouenneProblem *p)

  redirect variables to proper variable vector
- bool isBijective () const

  indicating if function is monotonically increasing
- CouNumber inverse (expression *vardep) const

  compute the inverse function
- void closestFeasible (expression *varind, expression *vardep, CouNumber &left, CouNumber &right) const

  closest feasible points in function in both directions
- bool isCuttable (CouenneProblem *problem, int index) const

  can this expression be further linearized or are we on its concave ("bad") side

Protected Attributes

- expression * copy_

  the expression this object is a (reference) copy of
- CouNumber value_

  saved value to be used by exprStore expressions

Additional Inherited Members

6.64.1 Detailed Description

Definition at line 25 of file CouenneExprCopy.hpp.

6.64.2 Constructor & Destructor Documentation

6.64.2.1 Couenne::exprCopy::exprCopy (expression * *copy*) [inline]

Empty constructor - used in cloning method of exprClone.

Constructor

Definition at line 45 of file CouenneExprCopy.hpp.

6.64.3 Member Function Documentation

6.64.3.1 bool Couenne::exprCopy::isaCopy ( ) const [inline], [virtual]

return true if this is a copy of something, i.e.

if it is an exprCopy or derivates

Reimplemented from Couenne::expression.

Definition at line 72 of file CouenneExprCopy.hpp.

6.64.3.2 CouNumber Couenne::exprCopy::selectBranch (const CouenneObject * *obj,* const OsiBranchingInformation * *info,* expression *& *var,* double *& *brpts,* double *& *brDist,* int & *way*) [inline], [virtual]

Set up branching object by evaluating many branching points for each expression's arguments.

Return estimated improvement in objective function

Reimplemented from Couenne::expression.

Definition at line 199 of file CouenneExprCopy.hpp.

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

- CouenneExprCopy.hpp

6.65 Couenne::exprCos Class Reference

class cosine, $\cos f(x)$

`#include <CouenneExprCos.hpp>`

Inheritance diagram for Couenne::exprCos:

Collaboration diagram for Couenne::exprCos:

Public Member Functions

- exprCos (expression *al)
  constructor, destructor
- expression * clone (Domain *d=NULL) const
  cloning method
- unary_function F ()
  the operator itself (e.g. sin, log...)
- std::string printOp () const
  print operator
- CouNumber gradientNorm (const double *x)
  return l-2 norm of gradient at given point
- expression * differentiate (int index)
  obtain derivative of expression
- void getBounds (expression *&, expression *&)
  Get lower and upper bound of an expression (if any)
- void getBounds (CouNumber &lb, CouNumber &ub)
  Get value of lower and upper bound of an expression.
- void generateCuts (expression *w, **OsiCuts** &cs, const CouenneCutGenerator *cg, t_chg_bounds *=NULL, int=-1, CouNumber=-COUENNE_INFINITY, CouNumber=COUENNE_INFINITY)
  *generate equality between *this and *w*
- virtual enum expr_type code ()
  code for comparisons
- bool impliedBound (int index, CouNumber *l, CouNumber *u, t_chg_bounds *chg, enum auxSign=expression↩::AUX_EQ)
  implied bound processing
- virtual CouNumber selectBranch (const CouenneObject *obj, const **OsiBranchingInformation** *info, expression *&var, double *&brpts, double *&brDist, int &way)
  Set up branching object by evaluating many branching points for each expression's arguments.
- virtual void closestFeasible (expression *varind, expression *vardep, CouNumber &left, CouNumber &right) const
  closest feasible points in function in both directions
- virtual bool isCuttable (CouenneProblem *problem, int index) const
  can this expression be further linearized or are we on its concave ("bad") side

Additional Inherited Members

6.65.1 Detailed Description

class cosine, $\cos f(x)$

Definition at line 20 of file CouenneExprCos.hpp.

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

- CouenneExprCos.hpp

6.66 Couenne::exprDiv Class Reference

class for divisions, $\frac{f(x)}{g(x)}$

`#include <CouenneExprDiv.hpp>`

Inheritance diagram for Couenne::exprDiv:

Collaboration diagram for Couenne::exprDiv:

Public Member Functions

- exprDiv (expression **al, int n=2)

  Constructor.
- exprDiv (expression *arg0, expression *arg1)

  Constructor with two arguments given explicitly.
- expression * clone (Domain *d=NULL) const

  Cloning method.
- std::string printOp () const

  Print operator.
- CouNumber operator() ()

  Function for the evaluation of the expression.
- CouNumber gradientNorm (const double *x)

  return l-2 norm of gradient at given point
- expression * differentiate (int index)

  Differentiation.
- expression * simplify ()

  Simplification.
- int Linearity ()

  Get a measure of "how linear" the expression is (see CouenneTypes.h)
- void getBounds (expression *&lb, expression *&ub)

  Get lower and upper bound of an expression (if any)
- void getBounds (CouNumber &lb, CouNumber &ub)

  Get value of lower and upper bound of an expression (if any)
- exprAux * standardize (CouenneProblem *p, bool addAux=true)

  Reduce expression in standard form, creating additional aux variables (and constraints)
- void generateCuts (expression *w, **OsiCuts** &cs, const CouenneCutGenerator *cg, t_chg_bounds *=NULL, int=-1, CouNumber=-COUENNE_INFINITY, CouNumber=COUENNE_INFINITY)

  *Generate equality between *this and *w.*
- virtual enum expr_type code ()

  Code for comparisons.
- bool isInteger ()

  is this expression integer?
- bool impliedBound (int, CouNumber *, CouNumber *, t_chg_bounds *, enum auxSign=expression::AUX_EQ)

  Implied bound processing.
- virtual CouNumber selectBranch (const CouenneObject *obj, const **OsiBranchingInformation** *info, expression *&var, double *&brpts, double *&brDist, int &way)

  Set up branching object by evaluating many branching points for each expression's arguments.

- virtual void closestFeasible (expression *varind, expression *vardep, CouNumber &left, CouNumber &right) const

  compute y^{lv} and y^{uv} for Violation Transfer algorithm
- virtual bool isCuttable (CouenneProblem *problem, int index) const

  can this expression be further linearized or are we on its concave ("bad") side

Additional Inherited Members

6.66.1 Detailed Description

class for divisions, $\frac{f(x)}{g(x)}$

Definition at line 24 of file CouenneExprDiv.hpp.

6.66.2 Member Function Documentation

6.66.2.1 CouNumber Couenne::exprDiv::operator() ( ) [inline], [virtual]

Function for the evaluation of the expression.

Compute division.

Implements Couenne::expression.

Definition at line 115 of file CouenneExprDiv.hpp.

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

- CouenneExprDiv.hpp

6.67 Couenne::expression Class Reference

Expression base class.

`#include <CouenneExpression.hpp>`

Inheritance diagram for Couenne::expression:

Public Types

- enum auxSign

  "sign" of the constraint defining an auxiliary.

Public Member Functions

- expression ()

  Constructor.
- expression (const expression &e, Domain *d=NULL)

  Copy constructor.
- virtual ~expression ()

Destructor.

- virtual expression * clone (Domain *d=NULL) const

  Cloning method.

- virtual int Index () const

  Return index of variable (only valid for exprVar and exprAux)

- virtual int nArgs () const

  return number of arguments (when applicable, that is, with N-ary functions)

- virtual expression ** ArgList () const

  return arglist (when applicable, that is, with N-ary functions)

- virtual void ArgList (expression **al)

  set arglist (used in deleting nodes without deleting children)

- virtual expression * Argument () const

  return argument (when applicable, i.e., with univariate functions)

- virtual expression ** ArgPtr ()

  return pointer to argument (when applicable, i.e., with univariate functions)

- virtual enum nodeType Type () const

  node type

- virtual expression * Image () const

  return pointer to corresponding expression (for auxiliary variables only)

- virtual void Image (expression *image)

  set expression associated with this auxiliary variable (for compatibility with exprAux)

- virtual CouNumber Value () const

  value (empty)

- virtual const expression * Original () const

  If this is an exprClone of a exprClone of an expr???, point to the original expr??? instead of an exprClone – improve computing efficiency.

- virtual void print (std::ostream &s=std::cout, bool=false) const

  print expression to iostream

- virtual CouNumber operator() ()=0

  null function for evaluating the expression

- virtual CouNumber gradientNorm (const double *x)

  return l-2 norm of gradient at given point

- virtual expression * differentiate (int)

  differentiation

- virtual int dependsOn (int *ind, int n, enum dig_type type=STOP_AT_AUX)

  dependence on variable set: return cardinality of subset of the set of indices in first argument which occur in expression.

- int dependsOn (int singleton, enum dig_type type=STOP_AT_AUX)

  version with one index only

- virtual int DepList (std::set< int > &deplist, enum dig_type type=ORIG_ONLY)

  fill std::set with indices of variables on which this expression depends.

- virtual expression * simplify ()

  simplify expression (useful for derivatives)

- virtual int Linearity ()

  get a measure of "how linear" the expression is (see CouenneTypes.h)

- virtual bool isDefinedInteger ()

  is this expression defined as an integer?

- virtual bool isInteger ()

is this expression integer?

- virtual void getBounds (expression *&, expression *&)

  Get lower and upper bound of an expression (if any)

- virtual void getBounds (CouNumber &, CouNumber &)

  Get lower and upper bound of an expression (if any) – real values.

- virtual exprAux * standardize (CouenneProblem *p, bool addAux=true)

  Create standard form of this expression, by:

- virtual void generateCuts (expression *w, **OsiCuts** &cs, const CouenneCutGenerator *cg, t_chg_bounds *chg=NULL, int wind=-1, CouNumber lb=-COUENNE_INFINITY, CouNumber ub=COUENNE_INFINITY)

  generate convexification cut for constraint w = this

- virtual enum expr_type code ()

  return integer for comparing expressions (used to recognize common expression)

- virtual enum convexity convexity () const

  either CONVEX, CONCAVE, AFFINE, or NONCONVEX

- virtual int compare (expression &)

  compare expressions

- virtual int compare (exprCopy &)

  compare copies of expressions

- virtual int rank ()

  used in rank-based branching variable choice: original variables have rank 1; auxiliary w=f(x) has rank r(w) = r(x)+1; finally, auxiliary w=f(x1,x2...,xk) has rank r(w) = 1+max{r(xi):i=1..k}.

- virtual bool impliedBound (int, CouNumber *, CouNumber *, t_chg_bounds *, enum auxSign=expression::AUX_EQ)

  does a backward implied bound processing on every expression, including exprSums although already done by Clp (useful when repeated within Couenne).

- virtual int Multiplicity ()

  multiplicity of a variable

- virtual CouNumber selectBranch (const CouenneObject *obj, const **OsiBranchingInformation** *info, expression *&var, double *&brpts, double *&brDist, int &way)

  set up branching object by evaluating many branching points for each expression's arguments.

- virtual void replace (exprVar *, exprVar *)

  replace expression with another

- virtual void fillDepSet (std::set< DepNode *, compNode > *, DepGraph *)

  update dependence set with index of variables on which this expression depends

- virtual void linkDomain (Domain *d)

  empty function to update domain pointer

- virtual void realign (const CouenneProblem *p)

  empty function to redirect variables to proper variable vector

- virtual bool isBijective () const

  indicating if function is monotonically increasing

- virtual CouNumber inverse (expression *vardep) const

  compute the inverse function

- virtual void closestFeasible (expression *varind, expression *vardep, CouNumber &left, CouNumber &right) const

  closest feasible points in function in both directions

- virtual bool isCuttable (CouenneProblem *problem, int index) const

  can this expression be further linearized or are we on its concave ("bad") side

- virtual bool isaCopy () const

return true if this is a copy of something (i.e. an exprCopy)

- virtual expression * Copy () const

  return copy of this expression (only makes sense in exprCopy)

6.67.1 Detailed Description

Expression base class.

An empty expression class with no type or operator() from which all other expression classes (for constants, variables, and operators) are derived.

Definition at line 48 of file CouenneExpression.hpp.

6.67.2 Member Enumeration Documentation

6.67.2.1 enum Couenne::expression::auxSign

"sign" of the constraint defining an auxiliary.

If the auxiliary is defined as $w \leq f(x)$, then it is LEQ. It is EQ and GEQ, respectively, if it is defined with $=$ and $\geq$.

Definition at line 55 of file CouenneExpression.hpp.

6.67.3 Constructor & Destructor Documentation

6.67.3.1 Couenne::expression::expression (const expression & *e*, Domain * *d* = NULL) [inline]

Copy constructor.

Pass pointer to variable vector when generating new problem, whose set of variables is equivalent but may be changed or whose value is independent.

Definition at line 63 of file CouenneExpression.hpp.

6.67.4 Member Function Documentation

6.67.4.1 virtual const expression* Couenne::expression::Original ( ) const [inline], [virtual]

If this is an exprClone of a exprClone of an expr???, point to the original expr??? instead of an exprClone – improve computing efficiency.

Only overloaded for exprClones/exprCopy, of course.

Reimplemented in Couenne::exprCopy.

Definition at line 114 of file CouenneExpression.hpp.

6.67.4.2 virtual void Couenne::expression::print (std::ostream & *s* = std::cout, bool = false) const [inline], [virtual]

print expression to iostream

descend into auxiliaries' image?

Reimplemented in Couenne::exprUpperBound, Couenne::exprCopy, Couenne::exprQuad, Couenne::exprAux, Couenne::exprVar, Couenne::exprUBQuad, Couenne::exprOp, Couenne::exprUnary, Couenne::exprGroup, Couenne::exprLowerBound, Couenne::exprClone, Couenne::exprConst, Couenne::exprInv, Couenne::exprLBQuad, Couenne::exprStore, Couenne::exprOpp, and Couenne::exprIVar.

Definition at line 118 of file CouenneExpression.hpp.

6.67.4.3 virtual int Couenne::expression::dependsOn (int * *ind,* int *n,* enum dig_type *type* = STOP_AT_AUX) [virtual]

dependence on variable set: return cardinality of subset of the set of indices in first argument which occur in expression.

Reimplemented in Couenne::exprUpperBound, Couenne::exprLowerBound, and Couenne::exprConst.

6.67.4.4 virtual int Couenne::expression::DepList (std::set< int > & *deplist,* enum dig_type *type* = ORIG_ONLY) [inline], [virtual]

fill std::set with indices of variables on which this expression depends.

Also deal with expressions that have no variable pointers (exprGroup, exprQuad)

Reimplemented in Couenne::exprQuad, Couenne::exprCopy, Couenne::exprAux, Couenne::exprVar, Couenne::exprOp, Couenne::exprUnary, and Couenne::exprGroup.

Definition at line 143 of file CouenneExpression.hpp.

6.67.4.5 virtual exprAux* Couenne::expression::standardize (CouenneProblem * *p,* bool *addAux* = true) [inline], [virtual]

Create standard form of this expression, by:

- creating auxiliary w variables and corresponding expressions
- returning linear counterpart as new constraint (to replace current one)

For the base exprOp class we only do the first part (for argument list components only), and the calling class (Sum, Sub, Mul, Pow, and the like) will do the part for its own object

addAux is true if a new auxiliary variable should be added associated with the standardized expression

Reimplemented in Couenne::exprCopy, Couenne::exprOp, Couenne::exprUnary, Couenne::exprOpp, Couenne::exprPow, Couenne::exprDiv, Couenne::exprSub, Couenne::exprMul, Couenne::exprSum, Couenne::exprEvenPow, Couenne::exprOddPow, Couenne::exprBinProd, and Couenne::exprMultiLin.

Definition at line 181 of file CouenneExpression.hpp.

6.67.4.6 virtual int Couenne::expression::rank ( ) [inline], [virtual]

used in rank-based branching variable choice: original variables have rank 1; auxiliary w=f(x) has rank r(w) = r(x)+1; finally, auxiliary w=f(x1,x2...,xk) has rank r(w) = 1+max{r(xi):i=1..k}.

Reimplemented in Couenne::exprQuad, Couenne::exprCopy, Couenne::exprVar, Couenne::exprAux, Couenne::exprOp, Couenne::exprUnary, Couenne::exprGroup, and Couenne::exprConst.

Definition at line 209 of file CouenneExpression.hpp.

6.67.4.7 virtual bool Couenne::expression::impliedBound (int , CouNumber * , CouNumber * , t_chg_bounds * , enum *auxSign* = expression::AUX_EQ) [inline],[virtual]

does a backward implied bound processing on every expression, including exprSums although already done by Clp (useful when repeated within Couenne).

Parameters are the index of the (auxiliary) variable in question and the current lower/upper bound. The method returns true if there has been a change on any bound on the variables on which the expression depends.

Reimplemented in Couenne::exprQuad, Couenne::exprVar, Couenne::exprPow, Couenne::exprSum, Couenne::exprSin, Couenne::exprDiv, Couenne::exprOpp, Couenne::exprInv, Couenne::exprSub, Couenne::exprEvenPow, Couenne::exprOddPow, Couenne::exprAbs, Couenne::exprCeil, Couenne::exprCos, Couenne::exprFloor, Couenne::exprExp, and Couenne::exprLog.

Definition at line 218 of file CouenneExpression.hpp.

6.67.4.8 virtual CouNumber Couenne::expression::selectBranch (const CouenneObject * *obj*, const OsiBranchingInformation * *info*, expression *& *var*, double *& *brpts*, double *& *brDist*, int & *way*) [inline],[virtual]

set up branching object by evaluating many branching points for each expression's arguments.

Return estimated improvement in objective function

Reimplemented in Couenne::exprQuad, Couenne::exprCopy, Couenne::exprSin, Couenne::exprPow, Couenne::exprDiv, Couenne::exprInv, Couenne::exprCos, Couenne::exprMul, Couenne::exprEvenPow, Couenne::exprOddPow, Couenne::exprCeil, Couenne::exprFloor, Couenne::exprAbs, Couenne::exprBinProd, Couenne::exprExp, Couenne::exprMultiLin, Couenne::exprLog, and Couenne::exprTrilinear.

Definition at line 228 of file CouenneExpression.hpp.

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

- CouenneExpression.hpp

6.68 Couenne::exprEvenPow Class Reference

Power of an expression (binary operator) with even exponent, $f(x)^k$ with $k \in \mathbb{Z}$ constant even.

`#include <CouenneExprEvenPow.hpp>`

Inheritance diagram for Couenne::exprEvenPow:

Collaboration diagram for Couenne::exprEvenPow:

Public Member Functions

- exprEvenPow (expression **al, int n=2)
  Constructor.
- exprEvenPow (expression *arg0, expression *arg1)
  Constructor with only two arguments.
- expression * clone (Domain *d=NULL) const
  cloning method
- CouNumber operator() ()
  function for the evaluation of the expression

- void getBounds (expression *&, expression *&)

  Get lower and upper bound of an expression (if any)
- void getBounds (CouNumber &lb, CouNumber &ub)

  Get value of lower and upper bound of an expression (if any)
- exprAux * standardize (CouenneProblem *p, bool addAux=true)

  reduce expression in standard form, creating additional aux variables (and constraints)
- void generateCuts (expression *w, **OsiCuts** &cs, const CouenneCutGenerator *cg, t_chg_bounds *=NULL, int=-1, CouNumber=-COUENNE_INFINITY, CouNumber=COUENNE_INFINITY)

  *generate equality between *this and *w*
- expression * getFixVar ()

  return an index to the variable's argument that is better fixed in a branching rule for solving a nonconvexity gap
- virtual enum expr_type code ()

  code for comparison
- bool impliedBound (int, CouNumber *, CouNumber *, t_chg_bounds *, enum auxSign=expression::AUX_EQ)

  implied bound processing
- virtual CouNumber selectBranch (const CouenneObject *obj, const **OsiBranchingInformation** *info, expression *&var, double *&brpts, double *&brDist, int &way)

  set up branching object by evaluating many branching points for each expression's arguments
- virtual bool isCuttable (CouenneProblem *problem, int index) const

  can this expression be further linearized or are we on its concave ("bad") side

Additional Inherited Members

6.68.1 Detailed Description

Power of an expression (binary operator) with even exponent, $f(x)^k$ with $k \in \mathbb{Z}$ constant even.

Definition at line 26 of file CouenneExprEvenPow.hpp.

6.68.2 Member Function Documentation

6.68.2.1 CouNumber Couenne::exprEvenPow::operator() ( ) [inline], [virtual]

function for the evaluation of the expression

compute power

Reimplemented from Couenne::exprPow.

Definition at line 90 of file CouenneExprEvenPow.hpp.

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

- CouenneExprEvenPow.hpp

6.69 Couenne::exprExp Class Reference

class for the exponential, $e^{f(x)}$

`#include <CouenneExprExp.hpp>`

Inheritance diagram for Couenne::exprExp:

Collaboration diagram for Couenne::exprExp:

Public Member Functions

- exprExp (expression ∗al)
  Constructor.
- expression ∗ clone (Domain ∗d=NULL) const
  Cloning method.
- unary_function F ()
  The operator's function.
- std::string printOp () const
  Print operator.
- CouNumber gradientNorm (const double ∗x)
  return l-2 norm of gradient at given point
- expression ∗ differentiate (int index)
  Differentiation.
- void getBounds (expression ∗&, expression ∗&)
  Get lower and upper bound of an expression (if any)
- virtual void getBounds (CouNumber &lb, CouNumber &ub)
  Get expression of lower and upper bound of an expression (if any)
- void generateCuts (expression ∗w, **OsiCuts** &cs, const CouenneCutGenerator ∗cg, t_chg_bounds ∗=NULL, int=-1, CouNumber=-COUENNE_INFINITY, CouNumber=COUENNE_INFINITY)
  Generate convexification cuts for this expression.
- virtual enum expr_type code ()
  Code for comparisons.
- bool impliedBound (int, CouNumber ∗, CouNumber ∗, t_chg_bounds ∗, enum auxSign=expression::AUX_EQ)
  Implied bound processing.
- virtual CouNumber selectBranch (const CouenneObject ∗obj, const **OsiBranchingInformation** ∗info, expression ∗&var, double ∗&brpts, double ∗&brDist, int &way)
  Set up branching object by evaluating many branching points for each expression's arguments.
- virtual bool isBijective () const
  return true if bijective
- virtual CouNumber inverse (expression ∗vardep) const
  inverse of exponential
- virtual bool isCuttable (CouenneProblem ∗problem, int index) const
  can this expression be further linearized or are we on its concave ("bad") side

Additional Inherited Members

6.69.1 Detailed Description

class for the exponential, $e^{f(x)}$

Definition at line 22 of file CouenneExprExp.hpp.

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

- CouenneExprExp.hpp

6.70 Couenne::exprFloor Class Reference

class floor, $\lfloor f(x) \rfloor$

`#include <CouenneExprFloor.hpp>`

Inheritance diagram for Couenne::exprFloor:

Collaboration diagram for Couenne::exprFloor:

Public Member Functions

- exprFloor (expression *arg)

  constructor, destructor
- expression * clone (Domain *d=NULL) const

  cloning method
- unary_function F ()

  the operator itself (e.g. sin, log...)
- std::string printOp () const

  print operator
- CouNumber gradientNorm (const double *x)

  return l-2 norm of gradient at given point
- expression * differentiate (int index)

  obtain derivative of expression
- void getBounds (expression *&, expression *&)

  Get lower and upper bound of an expression (if any)
- void getBounds (CouNumber &lb, CouNumber &ub)

  Get value of lower and upper bound of an expression.
- void generateCuts (expression *w, **OsiCuts** &cs, const CouenneCutGenerator *cg, t_chg_bounds *=NULL, int=-1, CouNumber=-COUENNE_INFINITY, CouNumber=COUENNE_INFINITY)

  *generate equality between *this and *w*
- virtual enum expr_type code ()

  code for comparisons
- bool impliedBound (int index, CouNumber *l, CouNumber *u, t_chg_bounds *chg, enum auxSign=expression↩ ::AUX_EQ)

  implied bound processing
- virtual CouNumber selectBranch (const CouenneObject *obj, const **OsiBranchingInformation** *info, expression *&var, double *&brpts, double *&brDist, int &way)

  Set up branching object by evaluating many branching points for each expression's arguments.
- virtual void closestFeasible (expression *varind, expression *vardep, CouNumber &left, CouNumber &right) const

  closest feasible points in function in both directions
- virtual bool isCuttable (CouenneProblem *problem, int index) const

  can this expression be further linearized or are we on its concave ("bad") side?

Additional Inherited Members

6.70.1 Detailed Description

class floor, $\lfloor f(x) \rfloor$

Definition at line 20 of file CouenneExprFloor.hpp.

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

- CouenneExprFloor.hpp

6.71 Couenne::exprGroup Class Reference

class Group, with constant, linear and nonlinear terms: $a_0 + \sum_{i=1}^{n} a_i x_i$

`#include <CouenneExprGroup.hpp>`

Inheritance diagram for Couenne::exprGroup:

Collaboration diagram for Couenne::exprGroup:

Public Member Functions

- exprGroup (CouNumber, lincoeff &, expression **=NULL, int=0)
  Constructor.
- exprGroup (const exprGroup &src, Domain *d=NULL)
  Copy constructor.
- virtual ~exprGroup ()
  Destructor – needed to clear bounds.
- virtual expression * clone (Domain *d=NULL) const
  Cloning method.
- CouNumber getc0 ()
  return constant term
- lincoeff & lcoeff () const
  return linear term coefficients
- virtual void print (std::ostream &=std::cout, bool=false) const
  Print expression to iostream.
- virtual CouNumber operator() ()
  function for the evaluation of the expression
- virtual CouNumber gradientNorm (const double *x)
  return l-2 norm of gradient at given point
- virtual int DepList (std::set< int > &deplist, enum dig_type type=ORIG_ONLY)
  fill in the set with all indices of variables appearing in the expression
- virtual expression * differentiate (int index)
  differentiation
- virtual expression * simplify ()
  simplification
- virtual int Linearity ()
  get a measure of "how linear" the expression is:

- virtual void getBounds (expression *&, expression *&)

  Get lower and upper bound of an expression (if any)

- virtual void getBounds (CouNumber &, CouNumber &)

  Get lower and upper bound of an expression (if any)

- virtual void generateCuts (expression *, **OsiCuts** &, const CouenneCutGenerator *, t_chg_bounds *=NULL, int=-1, CouNumber=-COUENNE_INFINITY, CouNumber=COUENNE_INFINITY)

  special version for linear constraints

- virtual int compare (exprGroup &)

  only compare with people of the same kind

- virtual enum expr_type code ()

  code for comparisons

- virtual bool isInteger ()

  is this expression integer?

- virtual int rank ()

  used in rank-based branching variable choice

- virtual void fillDepSet (std::set< DepNode *, compNode > *, DepGraph *)

  update dependence set with index of this variable

- virtual void replace (exprVar *x, exprVar *w)

  replace variable x with new (aux) w

- virtual void realign (const CouenneProblem *p)

  redirect variables to proper variable vector

Static Public Member Functions

- static expression * genExprGroup (CouNumber, lincoeff &, expression **=NULL, int=0)

  Generalized (static) constructor: check parameters and return a constant, a single variable, or a real exprGroup.

Protected Attributes

- lincoeff lcoeff_

  coefficients and indices of the linear term

- CouNumber c0_

  constant term

Additional Inherited Members

6.71.1 Detailed Description

class Group, with constant, linear and nonlinear terms: $a_0 + \sum_{i=1}^{n} a_i x_i$

Definition at line 25 of file CouenneExprGroup.hpp.

6.71.2 Member Function Documentation

6.71.2.1 CouNumber Couenne::exprGroup::operator() ( ) [inline],[virtual]

function for the evaluation of the expression

compute sum of linear and nonlinear terms

Reimplemented from Couenne::exprSum.

Reimplemented in Couenne::exprQuad.

Definition at line 127 of file CouenneExprGroup.hpp.

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

- CouenneExprGroup.hpp

6.72 Couenne::ExprHess Class Reference

expression matrices.

`#include <CouenneExprHess.hpp>`

6.72.1 Detailed Description

expression matrices.

Used to evaluate the Hessian of the Lagrangian function at an optimal solution of the NLP

Definition at line 21 of file CouenneExprHess.hpp.

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

- CouenneExprHess.hpp

6.73 Couenne::exprIf Class Reference

Inheritance diagram for Couenne::exprIf:

Collaboration diagram for Couenne::exprIf:

Additional Inherited Members

6.73.1 Detailed Description

Definition at line 18 of file CouenneExprIf.hpp.

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

- CouenneExprIf.hpp

6.74 Couenne::exprInv Class Reference

class inverse: $1/f(x)$

`#include <CouenneExprInv.hpp>`

Inheritance diagram for Couenne::exprInv:

Collaboration diagram for Couenne::exprInv:

Public Member Functions

- exprInv (expression *al)

  Constructors, destructor.
- expression * clone (Domain *d=NULL) const

  cloning method
- unary_function F ()

  the operator's function
- virtual void print (std::ostream &out=std::cout, bool=false) const

  output "1/argument"
- CouNumber gradientNorm (const double *x)

  return l-2 norm of gradient at given point
- expression * differentiate (int index)

  differentiation
- virtual int Linearity ()

  get a measure of "how linear" the expression is (see CouenneTypes.h)
- void getBounds (expression *&, expression *&)

  Get lower and upper bound of an expression (if any)
- void getBounds (CouNumber &lb, CouNumber &ub)

  Get value of lower and upper bound of an expression (if any)
- void generateCuts (expression *w, **OsiCuts** &cs, const CouenneCutGenerator *cg, t_chg_bounds *=NULL, int=-1, CouNumber=-COUENNE_INFINITY, CouNumber=COUENNE_INFINITY)

  *generate equality between *this and *w*
- virtual enum expr_type code ()

  code for comparisons
- bool impliedBound (int, CouNumber *, CouNumber *, t_chg_bounds *, enum auxSign=expression::AUX_EQ)

  implied bound processing
- virtual CouNumber selectBranch (const CouenneObject *obj, const **OsiBranchingInformation** *info, expression *&var, double *&brpts, double *&brDist, int &way)

  set up branching object by evaluating many branching points for each expression's arguments
- virtual bool isBijective () const

  return true if bijective
- virtual CouNumber inverse (expression *vardep) const

  return inverse of y=f(x)=1/x, i.e., x=1/y
- virtual bool isCuttable (CouenneProblem *problem, int index) const

  can this expression be further linearized or are we on its concave ("bad") side

Additional Inherited Members

6.74.1 Detailed Description

class inverse: $1/f(x)$

Definition at line 35 of file CouenneExprInv.hpp.

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

- CouenneExprInv.hpp

6.75 Couenne::exprIVar Class Reference

variable-type operator.

`#include <CouenneExprIVar.hpp>`

Inheritance diagram for Couenne::exprIVar:

Collaboration diagram for Couenne::exprIVar:

Public Member Functions

- exprIVar (int varIndex, Domain *d=NULL)

  Constructor.
- exprIVar (const exprIVar &e, Domain *d=NULL)

  Copy constructor – must go.
- virtual exprVar * clone (Domain *d=NULL) const

  Cloning method.
- virtual void print (std::ostream &out=std::cout, bool=false) const

  Print.
- virtual bool isDefinedInteger ()

  is this expression defined as an integer?
- virtual bool isInteger ()

  Is this expression integer?

Additional Inherited Members

6.75.1 Detailed Description

variable-type operator.

All variables of the expression must be objects of this class

Definition at line 25 of file CouenneExprIVar.hpp.

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

- CouenneExprIVar.hpp

6.76 Couenne::ExprJac Class Reference

Jacobian of the problem (computed through Couenne expression classes).

`#include <CouenneExprJac.hpp>`

6.76.1 Detailed Description

Jacobian of the problem (computed through Couenne expression classes).

Definition at line 21 of file CouenneExprJac.hpp.

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

- CouenneExprJac.hpp

6.77 Couenne::exprLBCos Class Reference

class to compute lower bound of a cosine based on the bounds of its arguments

`#include <CouenneExprBCos.hpp>`

Inheritance diagram for Couenne::exprLBCos:

Collaboration diagram for Couenne::exprLBCos:

Public Member Functions

- exprLBCos (expression *lb, expression *ub)

  Constructors, destructor.
- expression * clone (Domain *d=NULL) const

  cloning method
- CouNumber operator() ()

  function for the evaluation of the expression
- enum pos printPos () const

  print position (PRE, INSIDE, POST)
- std::string printOp () const

  print operator

Additional Inherited Members

6.77.1 Detailed Description

class to compute lower bound of a cosine based on the bounds of its arguments

Definition at line 27 of file CouenneExprBCos.hpp.

6.77.2 Member Function Documentation

6.77.2.1 CouNumber Couenne::exprLBCos::operator() ( ) [inline], [virtual]

function for the evaluation of the expression

compute sum

Implements Couenne::expression.

Definition at line 58 of file CouenneExprBCos.hpp.

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

- CouenneExprBCos.hpp

6.78 Couenne::exprLBDiv Class Reference

class to compute lower bound of a fraction based on the bounds of both numerator and denominator

`#include <CouenneExprBDiv.hpp>`

Inheritance diagram for Couenne::exprLBDiv:

Collaboration diagram for Couenne::exprLBDiv:

Public Member Functions

- exprLBDiv (expression **al, int n)
  Constructors, destructor.
- expression * clone (Domain *d=NULL) const
  cloning method
- CouNumber operator() ()
  function for the evaluation of the expression
- enum pos printPos () const
  print position (PRE, INSIDE, POST)
- std::string printOp () const
  print operator

Additional Inherited Members

6.78.1 Detailed Description

class to compute lower bound of a fraction based on the bounds of both numerator and denominator

Definition at line 37 of file CouenneExprBDiv.hpp.

6.78.2 Member Function Documentation

6.78.2.1 CouNumber Couenne::exprLBDiv::operator() ( ) [inline], [virtual]

function for the evaluation of the expression

compute sum

Implements Couenne::expression.

Definition at line 64 of file CouenneExprBDiv.hpp.

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

- CouenneExprBDiv.hpp

6.79 Couenne::exprLBMul Class Reference

class to compute lower bound of a product based on the bounds of both factors

`#include <CouenneExprBMul.hpp>`

Inheritance diagram for Couenne::exprLBMul:

Collaboration diagram for Couenne::exprLBMul:

Public Member Functions

- exprLBMul (expression **al, int n)
  Constructors, destructor.
- expression * clone (Domain *d=NULL) const
  cloning method
- CouNumber operator() ()
  function for the evaluation of the expression
- enum pos printPos () const
  print position (PRE, INSIDE, POST)
- std::string printOp () const
  print operator

Additional Inherited Members

6.79.1 Detailed Description

class to compute lower bound of a product based on the bounds of both factors

Definition at line 40 of file CouenneExprBMul.hpp.

6.79.2 Member Function Documentation

6.79.2.1 CouNumber Couenne::exprLBMul::operator() ( ) [inline], [virtual]

function for the evaluation of the expression

compute sum

Implements Couenne::expression.

Definition at line 67 of file CouenneExprBMul.hpp.

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

- CouenneExprBMul.hpp

6.80 Couenne::exprLBQuad Class Reference

class to compute lower bound of a fraction based on the bounds of both numerator and denominator

`#include <CouenneExprBQuad.hpp>`

Inheritance diagram for Couenne::exprLBQuad:

Collaboration diagram for Couenne::exprLBQuad:

Public Member Functions

- exprLBQuad (exprQuad *ref)

  Constructor.
- exprLBQuad (const exprLBQuad &src, Domain *d=NULL)

  copy constructor
- ~exprLBQuad ()

  destructor
- expression * clone (Domain *d=NULL) const

  cloning method
- CouNumber operator() ()

  function for the evaluation of the expression
- virtual void print (std::ostream &s=std::cout, bool descend=false) const

  I/O.

Additional Inherited Members

6.80.1 Detailed Description

class to compute lower bound of a fraction based on the bounds of both numerator and denominator

Definition at line 22 of file CouenneExprBQuad.hpp.

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

- CouenneExprBQuad.hpp

6.81 Couenne::exprLBSin Class Reference

class to compute lower bound of a sine based on the bounds on its arguments

`#include <CouenneExprBSin.hpp>`

Inheritance diagram for Couenne::exprLBSin:

Collaboration diagram for Couenne::exprLBSin:

Public Member Functions

- exprLBSin (expression *lb, expression *ub)
  Constructors, destructor.
- expression * clone (Domain *d=NULL) const
  cloning method
- CouNumber operator() ()
  function for the evaluation of the expression
- enum pos printPos () const
  print position (PRE, INSIDE, POST)
- std::string printOp () const
  print operator

Additional Inherited Members

6.81.1 Detailed Description

class to compute lower bound of a sine based on the bounds on its arguments

Definition at line 27 of file CouenneExprBSin.hpp.

6.81.2 Member Function Documentation

6.81.2.1 CouNumber Couenne::exprLBSin::operator() ( ) [inline], [virtual]

function for the evaluation of the expression

compute sum

Implements Couenne::expression.

Definition at line 58 of file CouenneExprBSin.hpp.

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

- CouenneExprBSin.hpp

6.82 Couenne::exprLog Class Reference

class logarithm, $\log f(x)$

`#include <CouenneExprLog.hpp>`

Inheritance diagram for Couenne::exprLog:

Collaboration diagram for Couenne::exprLog:

Public Member Functions

- exprLog (expression *al)
  Constructors, destructor.
- expression * clone (Domain *d=NULL) const

cloning method

- unary_function F ()

  the operator's function

- std::string printOp () const

  print operator

- CouNumber gradientNorm (const double *x)

  return l-2 norm of gradient at given point

- expression * differentiate (int index)

  differentiation

- void getBounds (expression *&, expression *&)

  Get lower and upper bound of an expression (if any)

- void getBounds (CouNumber &lb, CouNumber &ub)

  Get value of lower and upper bound of an expression (if any)

- void generateCuts (expression *w, **OsiCuts** &cs, const CouenneCutGenerator *cg, t_chg_bounds *=NULL, int=-1, CouNumber=-COUENNE_INFINITY, CouNumber=COUENNE_INFINITY)

  *generate equality between *this and *w*

- virtual enum expr_type code ()

  code for comparisons

- bool impliedBound (int, CouNumber *, CouNumber *, t_chg_bounds *, enum auxSign=expression::AUX_EQ)

  implied bound processing

- virtual CouNumber selectBranch (const CouenneObject *obj, const **OsiBranchingInformation** *info, expression *&var, double *&brpts, double *&brDist, int &way)

  set up branching object by evaluating many branching points for each expression's arguments

- virtual bool isBijective () const

  return true if feasible

- virtual CouNumber inverse (expression *vardep) const

  inverse of this operator

- virtual bool isCuttable (CouenneProblem *problem, int index) const

  can this expression be further linearized or are we on its concave ("bad") side

Additional Inherited Members

6.82.1 Detailed Description

class logarithm, $\log f(x)$

Definition at line 21 of file CouenneExprLog.hpp.

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

- CouenneExprLog.hpp

6.83 Couenne::exprLowerBound Class Reference

These are bound expression classes.

`#include <CouenneExprBound.hpp>`

Inheritance diagram for Couenne::exprLowerBound:

Collaboration diagram for Couenne::exprLowerBound:

Public Member Functions

- enum nodeType Type () const

  Node type.
- exprLowerBound (int varIndex, Domain *d=NULL)

  Constructor.
- exprLowerBound (const exprLowerBound &src, Domain *d=NULL)

  Copy constructor.
- exprLowerBound * clone (Domain *d=NULL) const

  cloning method
- void print (std::ostream &out=std::cout, bool=false) const

  Print to iostream.
- CouNumber operator() ()

  return the value of the variable
- expression * differentiate (int)

  differentiation
- int dependsOn (int *, int, enum dig_type type=STOP_AT_AUX)

  dependence on variable set
- virtual int Linearity ()

  get a measure of "how linear" the expression is:
- virtual enum expr_type code ()

  code for comparisons

Additional Inherited Members

6.83.1 Detailed Description

These are bound expression classes.

They are used in the parametric convexification part to obtain lower/upper bounds of an expression as a function of the expression itself.

For example, the lower and upper bounds to expression (x1 - exp (x2)) are (l1 - exp (u2)) and (u1 - exp (l2)), respectively, where l1 (l2) is the lower bound of x1 (x2) and u1 (u2) is the upper bound of x1 (x2).

A lower/upper bound of an expression is a function of all bounds in the expression and is known only when all variables bounds are known. lower bound

Definition at line 38 of file CouenneExprBound.hpp.

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

- CouenneExprBound.hpp

6.84 Couenne::exprMax Class Reference

Inheritance diagram for Couenne::exprMax:

Collaboration diagram for Couenne::exprMax:

Public Member Functions

- cloning method exprMax ∗ clone (Domain ∗d=NULL) const

  Cloning method.
- print position enum pos printPos () const

  print position (PRE, INSIDE, POST)
- function for the evaluation of the expression CouNumber operator() ()

  null function for evaluating the expression
- differentiation expression ∗ differentiate (int)

  differentiation
- simplification expression ∗ simplify ()

  simplification
- void getBounds (expression ∗&, expression ∗&)

  Get lower and upper bound of an expression (if any)
- code for virtual comparisons enum expr_type code ()

  return code to classify type of expression

Additional Inherited Members

6.84.1 Detailed Description

Definition at line 20 of file CouenneExprMax.hpp.

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

- CouenneExprMax.hpp

6.85 Couenne::exprMin Class Reference

Inheritance diagram for Couenne::exprMin:

Collaboration diagram for Couenne::exprMin:

Public Member Functions

- Cloning method exprMin ∗ clone (Domain ∗d=NULL) const

  Cloning method.
- Function for the evaluation of the expression CouNumber operator() ()

  null function for evaluating the expression
- Differentiation expression ∗ differentiate (int)

  differentiation
- Simplification expression ∗ simplify ()

  simplification
- void getBounds (expression ∗&, expression ∗&)

  Get lower and upper bound of an expression (if any)
- Code for virtual comparisons enum expr_type code ()

  return code to classify type of expression

Additional Inherited Members

6.85.1 Detailed Description

Definition at line 27 of file CouenneExprMin.hpp.

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

- CouenneExprMin.hpp

6.86 Couenne::exprMul Class Reference

class for multiplications, $\prod_{i=1}^{n} f_i(x)$

`#include <CouenneExprMul.hpp>`

Inheritance diagram for Couenne::exprMul:

Collaboration diagram for Couenne::exprMul:

Public Member Functions

- exprMul (expression **, int)
  Constructor.
- exprMul (expression *, expression *)
  Constructor with two arguments.
- virtual expression * clone (Domain *d=NULL) const
  Cloning method.
- std::string printOp () const
  Print operator.
- CouNumber operator() ()
  Method to evaluate the expression.
- virtual CouNumber gradientNorm (const double *x)
  return l-2 norm of gradient at given point
- expression * differentiate (int index)
  differentiation
- expression * simplify ()
  simplification
- virtual int Linearity ()
  get a measure of "how linear" the expression is:
- virtual void getBounds (expression *&, expression *&)
  Get lower and upper bound of an expression (if any)
- virtual void getBounds (CouNumber &lb, CouNumber &ub)
  Get value of lower and upper bound of an expression (if any)
- virtual exprAux * standardize (CouenneProblem *p, bool addAux=true)
  reduce expression in standard form, creating additional aux variables (and constraints)
- virtual void generateCuts (expression *w, **OsiCuts** &cs, const CouenneCutGenerator *cg, t_chg_bounds *=NULL, int=-1, CouNumber=-COUENNE_INFINITY, CouNumber=COUENNE_INFINITY)
  *generate equality between *this and *w*

- virtual enum expr_type code ()

  code for comparison
- virtual bool impliedBound (int, CouNumber *, CouNumber *, t_chg_bounds *, enum Couenne::expression::auxSign=Couenne::expression::AUX_EQ)

  implied bound processing
- virtual CouNumber selectBranch (const CouenneObject *obj, const **OsiBranchingInformation** *info, expression *&var, double *&brpts, double *&brDist, int &way)

  set up branching object by evaluating many branching points for each expression's arguments
- virtual void closestFeasible (expression *varind, expression *vardep, CouNumber &left, CouNumber &right) const

  compute y^{lv} and y^{uv} for Violation Transfer algorithm

Protected Member Functions

- int impliedBoundMul (CouNumber wl, CouNumber wu, std::vector< CouNumber > &xl, std::vector< CouNumber > &xu, std::vector< std::pair< int, CouNumber > > &nl, std::vector< std::pair< int, CouNumber > > &nu)

  inferring bounds on factors of a product
- CouNumber balancedMul (const **OsiBranchingInformation** *info, int index, int wind)

  balanced strategy for branching point selection in products
- virtual bool isCuttable (CouenneProblem *problem, int index) const

  can this expression be further linearized or are we on its concave ("bad") side

Additional Inherited Members

6.86.1 Detailed Description

class for multiplications, $\prod_{i=1}^{n} f_i(x)$

Definition at line 24 of file CouenneExprMul.hpp.

6.86.2 Member Function Documentation

6.86.2.1 CouNumber Couenne::exprMul::operator() ( ) [inline], [virtual]

Method to evaluate the expression.

compute multiplication

Implements Couenne::expression.

Definition at line 118 of file CouenneExprMul.hpp.

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

- CouenneExprMul.hpp

6.87 Couenne::exprMultiLin Class Reference

another class for multiplications, $\prod_{i=1}^{n} f_i(x)$

`#include <CouenneExprMultiLin.hpp>`

Inheritance diagram for Couenne::exprMultiLin:

Collaboration diagram for Couenne::exprMultiLin:

Public Member Functions

- exprMultiLin (expression **, int)

  Constructor.
- exprMultiLin (expression *, expression *)

  Constructor with two arguments.
- CouNumber gradientNorm (const double *x)

  return l-2 norm of gradient at given point
- expression * differentiate (int index)

  differentiation
- expression * simplify ()

  simplification
- virtual int Linearity ()

  get a measure of "how linear" the expression is:
- virtual void getBounds (expression *&, expression *&)

  Get lower and upper bound of an expression (if any)
- virtual void getBounds (CouNumber &lb, CouNumber &ub)

  Get value of lower and upper bound of an expression (if any)
- virtual exprAux * standardize (CouenneProblem *p, bool addAux=true)

  reduce expression in standard form, creating additional aux variables (and constraints)
- void generateCuts (expression *w, **OsiCuts** &cs, const CouenneCutGenerator *cg, t_chg_bounds *=NULL, int=-1, CouNumber=-COUENNE_INFINITY, CouNumber=COUENNE_INFINITY)

  *generate equality between *this and *w*
- virtual enum expr_type code ()

  code for comparison
- bool impliedBound (int, CouNumber *, CouNumber *, t_chg_bounds *, enum Couenne::expression::auxSign=Couenne::expression::AUX_EQ)

  implied bound processing
- virtual CouNumber selectBranch (const CouenneObject *obj, const **OsiBranchingInformation** *info, expression *&var, double *&brpts, double *&brDist, int &way)

  set up branching object by evaluating many branching points for each expression's arguments
- virtual void closestFeasible (expression *varind, expression *vardep, CouNumber &left, CouNumber &right) const

  compute y^{lv} and y^{uv} for Violation Transfer algorithm

Protected Member Functions

- int impliedBoundMul (CouNumber wl, CouNumber wu, std::vector< CouNumber > &xl, std::vector< CouNumber > &xu, std::vector< std::pair< int, CouNumber > > &nl, std::vector< std::pair< int, CouNumber > > &nu)

  inferring bounds on factors of a product
- CouNumber balancedMul (const **OsiBranchingInformation** *info, int index, int wind)

  balanced strategy for branching point selection in products
- virtual bool isCuttable (CouenneProblem *problem, int index) const

  can this expression be further linearized or are we on its concave ("bad") side

Additional Inherited Members

6.87.1 Detailed Description

another class for multiplications, $\prod_{i=1}^{n} f_i(x)$

Definition at line 21 of file CouenneExprMultiLin.hpp.

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

- CouenneExprMultiLin.hpp

6.88 Couenne::exprNorm Class Reference

Class for p-norms, $||f(x)||_p = (\sum_{i=1}^{n} f_i(x)^p)^{\frac{1}{p}}$.

`#include <CouenneExprNorm.hpp>`

Inheritance diagram for Couenne::exprNorm:

Collaboration diagram for Couenne::exprNorm:

Additional Inherited Members

6.88.1 Detailed Description

Class for p-norms, $||f(x)||_p = (\sum_{i=1}^{n} f_i(x)^p)^{\frac{1}{p}}$.

Definition at line 20 of file CouenneExprNorm.hpp.

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

- CouenneExprNorm.hpp

6.89 Couenne::exprOddPow Class Reference

Power of an expression (binary operator), $f(x)^k$ with k constant.

`#include <CouenneExprOddPow.hpp>`

Inheritance diagram for Couenne::exprOddPow:

Collaboration diagram for Couenne::exprOddPow:

Public Member Functions

- exprOddPow (expression **al, int n=2)
  Constructor.
- exprOddPow (expression *arg0, expression *arg1)
  Constructor with only two arguments.
- expression * clone (Domain *d=NULL) const
  cloning method

- std::string printOp () const

  print operator
- CouNumber operator() ()

  function for the evaluation of the expression
- void getBounds (expression ∗&, expression ∗&)

  Get lower and upper bound of an expression (if any)
- void getBounds (CouNumber &lb, CouNumber &ub)

  Get value of lower and upper bound of an expression (if any)
- exprAux ∗ standardize (CouenneProblem ∗p, bool addAux=true)

  reduce expression in standard form, creating additional aux variables (and constraints)
- void generateCuts (expression ∗w, **OsiCuts** &cs, const CouenneCutGenerator ∗cg, t_chg_bounds ∗=NULL, int=-1, CouNumber=-COUENNE_INFINITY, CouNumber=COUENNE_INFINITY)

  generate equality between ∗this and ∗w
- expression ∗ getFixVar ()

  return an index to the variable's argument that is better fixed in a branching rule for solving a nonconvexity gap
- virtual enum expr_type code ()

  code for comparison
- bool impliedBound (int, CouNumber ∗, CouNumber ∗, t_chg_bounds ∗, enum auxSign=expression::AUX_EQ)

  implied bound processing
- virtual CouNumber selectBranch (const CouenneObject ∗obj, const **OsiBranchingInformation** ∗info, expression ∗&var, double ∗&brpts, double ∗&brDist, int &way)

  set up branching object by evaluating many branching points for each expression's arguments
- virtual bool isCuttable (CouenneProblem ∗problem, int index) const

  can this expression be further linearized or are we on its concave ("bad") side

Additional Inherited Members

6.89.1 Detailed Description

Power of an expression (binary operator), $f(x)^k$ with k constant.

Definition at line 22 of file CouenneExprOddPow.hpp.

6.89.2 Member Function Documentation

6.89.2.1 CouNumber Couenne::exprOddPow::operator() ( ) [inline], [virtual]

function for the evaluation of the expression

compute power

Reimplemented from Couenne::exprPow.

Definition at line 90 of file CouenneExprOddPow.hpp.

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

- CouenneExprOddPow.hpp

6.90 Couenne::exprOp Class Reference

general n-ary operator-type expression: requires argument list.

`#include <CouenneExprOp.hpp>`

Inheritance diagram for Couenne::exprOp:

Collaboration diagram for Couenne::exprOp:

Public Member Functions

- virtual enum nodeType Type () const
  Node type.
- exprOp (expression **arglist, int nargs)
  Constructor.
- exprOp (expression *arg0, expression *arg1)
  Constructor with two arguments (for convenience)
- virtual ~exprOp ()
  Destructor.
- exprOp (const exprOp &e, Domain *d=NULL)
  Copy constructor: only allocate space for argument list, which will be copied with clonearglist()
- expression ** ArgList () const
  return argument list
- virtual void ArgList (expression **al)
  set arglist (used in deleting nodes without deleting children)
- int nArgs () const
  return number of arguments
- virtual void print (std::ostream &out=std::cout, bool=false) const
  I/O.
- virtual enum pos printPos () const
  print position (PRE, INSIDE, POST)
- virtual std::string printOp () const
  print operator
- virtual int DepList (std::set< int > &deplist, enum dig_type type=ORIG_ONLY)
  fill in the set with all indices of variables appearing in the expression
- virtual expression * simplify ()
  simplification
- expression ** clonearglist (Domain *d=NULL) const
  clone argument list (for use with clone method)
- int shrink_arglist (CouNumber, CouNumber)
  compress argument list
- virtual int Linearity ()
  get a measure of "how linear" the expression is (see CouenneTypes.h)
- virtual exprAux * standardize (CouenneProblem *, bool addAux=true)
  generate auxiliary variable
- virtual enum expr_type code ()
  return code to classify type of expression
- virtual bool isInteger ()

is this expression integer?

- virtual int compare (exprOp &)

  compare with other generic exprOp

- virtual int rank ()

  used in rank-based branching variable choice

- virtual void fillDepSet (std::set< DepNode *, compNode > *dep, DepGraph *g)

  fill in dependence structure update dependence set with index of this variable

- virtual void replace (exprVar *, exprVar *)

  replace variable with other

- virtual void realign (const CouenneProblem *p)

  empty function to redirect variables to proper variable vector

Protected Attributes

- expression ** arglist_

  argument list is an array of pointers to other expressions

- int nargs_

  number of arguments (cardinality of arglist)

Additional Inherited Members

6.90.1 Detailed Description

general n-ary operator-type expression: requires argument list.

All non-unary and non-leaf operators, i.e., sum, subtraction, multiplication, power, division, max, min, etc. are derived from this class.

Definition at line 31 of file CouenneExprOp.hpp.

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

- CouenneExprOp.hpp

6.91 Couenne::exprOpp Class Reference

class opposite, $-f(x)$

`#include <CouenneExprOpp.hpp>`

Inheritance diagram for Couenne::exprOpp:

Collaboration diagram for Couenne::exprOpp:

Public Member Functions

- exprOpp (expression *al)

  Constructors, destructor.

- expression * clone (Domain *d=NULL) const

  cloning method

- unary_function F ()

  the operator's function
- void print (std::ostream &out, bool descend) const

  Output.
- CouNumber gradientNorm (const double *x)

  return l-2 norm of gradient at given point
- expression * differentiate (int index)

  differentiation
- virtual expression * simplify ()

  simplification
- int Linearity ()

  get a measure of "how linear" the expression is (see CouenneTypes.h)
- void getBounds (expression *&, expression *&)

  Get lower and upper bound of an expression (if any)
- void getBounds (CouNumber &, CouNumber &)

  Get value of lower and upper bound of an expression (if any)
- virtual void generateCuts (expression *, **OsiCuts** &, const CouenneCutGenerator *, t_chg_bounds *=NULL, int=-1, CouNumber=-COUENNE_INFINITY, CouNumber=COUENNE_INFINITY)

  special version for linear constraints
- virtual enum expr_type code ()

  code for comparisons
- bool isInteger ()

  is this expression integer?
- bool impliedBound (int, CouNumber *, CouNumber *, t_chg_bounds *, enum auxSign=expression::AUX_EQ)

  implied bound processing
- exprAux * standardize (CouenneProblem *, bool addAux=true)

  standardization (to deal with complex arguments)

Additional Inherited Members

6.91.1 Detailed Description

class opposite, $-f(x)$

Definition at line 27 of file CouenneExprOpp.hpp.

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

- CouenneExprOpp.hpp

6.92 Couenne::exprPow Class Reference

Power of an expression (binary operator), $f(x)^k$ with k constant.

`#include <CouenneExprPow.hpp>`

Inheritance diagram for Couenne::exprPow:

Collaboration diagram for Couenne::exprPow:

Public Member Functions

- exprPow (expression **al, int n=2, bool signpower=false)
  Constructor.
- exprPow (expression *arg0, expression *arg1, bool signpower=false)
  Constructor with only two arguments.
- expression * clone (Domain *d=NULL) const
  cloning method
- virtual enum pos printPos () const
  print operator positioning
- virtual std::string printOp () const
  print operator
- virtual CouNumber operator() ()
  function for the evaluation of the expression
- virtual CouNumber gradientNorm (const double *x)
  return l-2 norm of gradient at given point
- virtual expression * differentiate (int index)
  differentiation
- virtual expression * simplify ()
  simplification
- virtual int Linearity ()
  get a measure of "how linear" the expression is
- virtual bool isInteger ()
  is this expression integer?
- virtual void getBounds (expression *&, expression *&)
  Get lower and upper bound of an expression (if any)
- virtual void getBounds (CouNumber &lb, CouNumber &ub)
  Get value of lower and upper bound of an expression (if any)
- virtual exprAux * standardize (CouenneProblem *p, bool addAux=true)
  reduce expression in standard form, creating additional aux variables (and constraints)
- virtual void generateCuts (expression *w, **OsiCuts** &cs, const CouenneCutGenerator *cg, t_chg_bounds *=N↩ ULL, int=-1, CouNumber=-COUENNE_INFINITY, CouNumber=COUENNE_INFINITY)
  *generate equality between *this and *w*
- virtual expression * getFixVar ()
  return an index to the variable's argument that is better fixed in a branching rule for solving a nonconvexity gap
- virtual enum expr_type code ()
  code for comparison
- virtual bool impliedBound (int, CouNumber *, CouNumber *, t_chg_bounds *, enum auxSign=expression::AU↩ X_EQ)
  implied bound processing
- virtual CouNumber selectBranch (const CouenneObject *obj, const **OsiBranchingInformation** *info, expression *&var, double *&brpts, double *&brDist, int &way)
  set up branching object by evaluating many branching points for each expression's arguments
- virtual void closestFeasible (expression *varind, expression *vardep, CouNumber &left, CouNumber &right) const
  compute y^{lv} and y^{uv} for Violation Transfer algorithm
- virtual bool isCuttable (CouenneProblem *problem, int index) const
  can this expression be further linearized or are we on its concave ("bad") side
- virtual bool isSignpower () const
  return whether this expression corresponds to a signed integer power

Additional Inherited Members

6.92.1 Detailed Description

Power of an expression (binary operator), $f(x)^k$ with k constant.

Definition at line 30 of file CouenneExprPow.hpp.

6.92.2 Member Function Documentation

6.92.2.1 CouNumber Couenne::exprPow::operator() ( ) [inline], [virtual]

function for the evaluation of the expression

compute power

Implements Couenne::expression.

Reimplemented in Couenne::exprEvenPow, and Couenne::exprOddPow.

Definition at line 173 of file CouenneExprPow.hpp.

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

- CouenneExprPow.hpp

6.93 Couenne::exprPWLinear Class Reference

Inheritance diagram for Couenne::exprPWLinear:

Collaboration diagram for Couenne::exprPWLinear:

Additional Inherited Members

6.93.1 Detailed Description

Definition at line 18 of file CouenneExprPWLinear.hpp.

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

- CouenneExprPWLinear.hpp

6.94 Couenne::exprQuad Class Reference

class exprQuad, with constant, linear and quadratic terms

`#include <CouenneExprQuad.hpp>`

Inheritance diagram for Couenne::exprQuad:

Collaboration diagram for Couenne::exprQuad:

Public Types

- typedef std::vector< std::pair< exprVar *, CouNumber > > sparseQcol

  matrix

Protected Attributes

Q matrix storage

Sparse implementation: given expression of the form $\sum_{i\in N, j\in N} q_{ij}x_ix_j$, qindexI_ and qindexJ_ contain respectively entries i and j for which q_{ij} is nonzero in $q_{ij}x_ix_j$:

- sparseQ **matrix_**

Convexification data structures

These are filled by alphaConvexify, which implements the alpha-convexification method described in the LaGO paper by Nowak and Vigerske – see also Adjiman and Floudas.

- std::vector< std::pair< CouNumber, std::vector< std::pair< exprVar *, CouNumber > > > > eigen_

  eigenvalues and eigenvectors
- std::map< exprVar *, std::pair< CouNumber, CouNumber > > bounds_

  current bounds (checked before re-computing eigenvalues/vectors)
- int nqterms_

  number of non-zeroes in Q
- exprQuad (CouNumber c0, std::vector< std::pair< exprVar *, CouNumber > > &lcoeff, std::vector< quadElem > &qcoeff, expression **al=NULL, int n=0)

  Constructor.
- exprQuad (const exprQuad &src, Domain *d=NULL)

  Copy constructor.
- sparseQ & **getQ** () const
- int **getnQTerms** ()
- virtual expression * clone (Domain *d=NULL) const

  cloning method
- virtual void print (std::ostream &=std::cout, bool=false) const

  Print expression to an iostream.
- virtual CouNumber operator() ()

  Function for the evaluation of the expression.
- CouNumber gradientNorm (const double *x)

  return l-2 norm of gradient at given point
- virtual expression * differentiate (int index)

  Compute derivative of this expression with respect to variable whose index is passed as argument.
- virtual expression * simplify ()

  Simplify expression.
- virtual int Linearity ()

  Get a measure of "how linear" the expression is.
- virtual void getBounds (expression *&, expression *&)

  Get lower and upper bound of an expression (if any)
- virtual void getBounds (CouNumber &, CouNumber &)

Get lower and upper bound of an expression (if any)

- virtual void generateCuts (expression *w, **OsiCuts** &cs, const CouenneCutGenerator *cg, t_chg_bounds *=NULL, int=-1, CouNumber=-COUENNE_INFINITY, CouNumber=COUENNE_INFINITY)

  Generate cuts for the quadratic expression, which are supporting hyperplanes of the concave upper envelope and the convex lower envelope.

- virtual bool alphaConvexify (const CouenneProblem *)

  Compute data for α-convexification of a quadratic form (fills in dCoeff_ and dIndex_ for the convex underestimator)

- void quadCuts (expression *w, **OsiCuts** &cs, const CouenneCutGenerator *cg)

  method exprQuad::quadCuts

- virtual int compare (exprQuad &)

  Compare two exprQuad.

- virtual enum expr_type code ()

  Code for comparisons.

- virtual int rank ()

  Used in rank-based branching variable choice.

- virtual bool isInteger ()

  is this expression integer?

- virtual int DepList (std::set< int > &deplist, enum dig_type type=ORIG_ONLY)

  fill in the set with all indices of variables appearing in the expression

- virtual CouNumber selectBranch (const CouenneObject *obj, const **OsiBranchingInformation** *info, expression *&var, double *&brpts, double *&brDist, int &way)

  Set up branching object by evaluating many branching points for each expression's arguments.

- virtual void fillDepSet (std::set< DepNode *, compNode > *dep, DepGraph *g)

  Fill dependence set of the expression associated with this auxiliary variable.

- virtual void replace (exprVar *x, exprVar *w)

  replace variable x with new (aux) w

- virtual void realign (const CouenneProblem *p)

  replace variable x with new (aux) w

- virtual bool impliedBound (int, CouNumber *, CouNumber *, t_chg_bounds *, enum auxSign=expression::AUX_EQ)

  implied bound processing

- CouNumber computeQBound (int sign)

  method to compute the bound based on sign: -1 for lower, +1 for upper

- virtual void closestFeasible (expression *varind, expression *vardep, CouNumber &left, CouNumber &right) const

  compute y^{lv} and y^{uv} for Violation Transfer algorithm

- void computeQuadFiniteBound (CouNumber &qMin, CouNumber &qMax, CouNumber *l, CouNumber *u, int &indInfLo, int &indInfUp)

  return lower and upper bound of quadratic expression

- virtual bool isCuttable (CouenneProblem *problem, int index) const

  can this expression be further linearized or are we on its concave ("bad") side

Additional Inherited Members

6.94.1 Detailed Description

class exprQuad, with constant, linear and quadratic terms

It represents an expression of the form $a_0 + \sum_{i \in I} b_i x_i + x^T Q x + \sum_{i \in J} h_i(x)$, with $a_0 + \sum_{i \in I} b_i x_i$ an affine term, $x^T Q x$ a quadratic term, and a nonlinear sum $\sum_{i \in J} h_i(x)$. Standardization checks possible quadratic or linear terms in the latter and includes them in the former parts.

If $h_i(x)$ is a product of two nonlinear, nonquadratic functions $h'(x)h''(x)$, two auxiliary variables $w' = f'(x)$ and $w'' = h''(x)$ are created and the product $w'w''$ is included in the quadratic part of the exprQuad. If $h(x)$ nonquadratic, nonlinear function, an auxiliary variable $w = h(x)$ is created and included in the linear part.

Definition at line 44 of file CouenneExprQuad.hpp.

6.94.2 Member Function Documentation

6.94.2.1 CouNumber Couenne::exprQuad::operator() ( ) [inline], [virtual]

Function for the evaluation of the expression.

Compute sum of linear and nonlinear terms.

Reimplemented from Couenne::exprGroup.

Definition at line 293 of file CouenneExprQuad.hpp.

6.94.2.2 virtual void Couenne::exprQuad::generateCuts (expression * *w*, OsiCuts & *cs*, const CouenneCutGenerator * *cg*, t_chg_bounds * = NULL, int = -1, CouNumber = -COUENNE_INFINITY, CouNumber = COUENNE_INFINITY) [virtual]

Generate cuts for the quadratic expression, which are supporting hyperplanes of the concave upper envelope and the convex lower envelope.

Reimplemented from Couenne::exprGroup.

6.94.2.3 void Couenne::exprQuad::quadCuts (expression * *w*, OsiCuts & *cs*, const CouenneCutGenerator * *cg*)

method exprQuad::quadCuts

Based on the information (dIndex_, dCoeffLo_, dCoeffUp_) created/modified by alphaConvexify(), create convexification cuts for this expression.

The original constraint is :

$$\eta = a_0 + a^T x + x^T Q x$$

where η is the auxiliary corresponding to this expression and w_j are the auxiliaries corresponding to the other non-linear terms contained in the expression.

The under-estimator of $x^T Q x$ is given by

$$x^T Q x + \sum \lambda_{\min,i} (x_i - l_i)(u_i - x_i)$$

and its over-estimator is given by

$$x^T Q x + \sum \lambda_{\max,i} (x_i - l_i)(u_i - x_i)$$

(where $\lambda_{\min,i} = \frac{\lambda_{\min}}{w_i^2}$, $\lambda_{\max,i} = \frac{\lambda_{\max}}{w_i^2}$, and $w_i = u_i - l_i$), where $\lambda_{\max}$ ($\lambda_{\max}$) is the minimum (maximum) eigenvalue of the matrix $A = \mathrm{Diag}(\mathbf{u} - \mathbf{l})Q\mathrm{Diag}(\mathbf{u} - \mathbf{l})$, obtained by pre- and post-multiplying Q by the diagonal matrix whose i-th element is $u_i - l_i$.

Let $\tilde{a}_0(\lambda)$, $\tilde{a}(\lambda)$ and $\tilde{Q}(\lambda)$ be

$$\tilde{a}_0(\lambda) = a_0 - \sum_{i=1}^{n} \lambda_i l_i u_i$$

$$\tilde{a}(\lambda) = a + \begin{bmatrix} \lambda_1(u_1 + l_1) \\ \vdots \\ \lambda_n(u_n + l_n) \end{bmatrix},$$

$$\tilde{Q}(\lambda) = Q - \begin{pmatrix} \lambda_1 & & 0 \\ & \ddots & \\ 0 & & \lambda_n \end{pmatrix}.$$

The convex relaxation of the initial constraint is then given by the two constraints

$$\eta \geq \tilde{a}_0(\lambda_{\min}) + \tilde{a}(\lambda_{\min})^T x + x^T \tilde{Q}(\lambda_{\min}) x$$

$$\eta \leq \tilde{a}_0(\lambda_{\max}) + \tilde{a}(\lambda_{\max})^T x + x^T \tilde{Q}(\lambda_{\max}) x$$

The cut is computed as follow. Let (x^*, η^*) be the solution at hand. The two outer-approximation cuts are:

$$\eta \geq \tilde{a}_0(\lambda_{\min}) + \tilde{a}(\lambda_{\min})^T x + {x^*}^T \tilde{Q}(\lambda_{\min})(2x - x^*)$$

and

$$\eta \leq \tilde{a}_0(\lambda_{\max}) + \tilde{a}(\lambda_{\max})^T x + {x^*}^T \tilde{Q}(\lambda_{\max})(2x - x^*);$$

grouping coefficients, we get:

$${x^*}^T \tilde{Q}(\lambda_{\min}) x^* - \tilde{a}_0(\lambda_{\min}) \geq (\tilde{a}(\lambda_{\min}) + 2\tilde{Q}(\lambda_{\min}) x^*)^T x - \eta$$

and

$${x^*}^T \tilde{Q}(\lambda_{\max}) x^* - \tilde{a}_0(\lambda_{\max}) \leq (\tilde{a}(\lambda_{\max}) + 2\tilde{Q}(\lambda_{\max}) x^*)^T x - \eta$$

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

- CouenneExprQuad.hpp

6.95 Couenne::exprSignPow Class Reference

Power of an expression (binary operator), $f(x)|f(x)|^{k-1}$ with $k \in \mathbb{R}$ constant.

`#include <CouenneExprSignPow.hpp>`

Public Member Functions

- exprSignPow (expression **al, int n=2)

  Constructor.
- exprSignPow (expression *arg0, expression *arg1)

  Constructor with only two arguments.
- expression * clone (Domain *d=NULL) const

  cloning method
- CouNumber operator() ()

  function for the evaluation of the expression
- void getBounds (expression *&, expression *&)

  Get lower and upper bound of an expression (if any)
- void getBounds (CouNumber &lb, CouNumber &ub)

  Get value of lower and upper bound of an expression (if any)
- exprAux * standardize (CouenneProblem *p, bool addAux=true)

  reduce expression in standard form, creating additional aux variables (and constraints)
- void generateCuts (expression *w, **OsiCuts** &cs, const CouenneCutGenerator *cg, t_chg_bounds *=NULL, int=-1, CouNumber=-COUENNE_INFINITY, CouNumber=COUENNE_INFINITY)

  *generate equality between *this and *w*
- virtual enum expr_type code ()

  code for comparison
- bool impliedBound (int, CouNumber *, CouNumber *, t_chg_bounds *, enum auxSign=expression::AUX_EQ)

  implied bound processing
- virtual CouNumber selectBranch (const CouenneObject *obj, const **OsiBranchingInformation** *info, expression *&var, double *&brpts, double *&brDist, int &way)

  set up branching object by evaluating many branching points for each expression's arguments
- virtual bool isCuttable (CouenneProblem *problem, int index) const

  can this expression be further linearized or are we on its concave ("bad") side

6.95.1 Detailed Description

Power of an expression (binary operator), $f(x)|f(x)|^{k-1}$ with $k \in \mathbb{R}$ constant.

Definition at line 25 of file CouenneExprSignPow.hpp.

6.95.2 Member Function Documentation

6.95.2.1 CouNumber Couenne::exprSignPow::operator() ( ) [inline]

function for the evaluation of the expression

compute power

Definition at line 85 of file CouenneExprSignPow.hpp.

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

- CouenneExprSignPow.hpp

6.96 Couenne::exprSin Class Reference

class for $\sin f(x)$

`#include <CouenneExprSin.hpp>`

Inheritance diagram for Couenne::exprSin:

Collaboration diagram for Couenne::exprSin:

Public Member Functions

- exprSin (expression *al)

  Constructors, destructor.
- expression * clone (Domain *d=NULL) const

  cloning method
- unary_function F ()

  the operator itself (e.g. sin, log...)
- std::string printOp () const

  print operator
- CouNumber gradientNorm (const double *x)

  return l-2 norm of gradient at given point
- expression * differentiate (int index)

  differentiation
- void getBounds (expression *&, expression *&)

  Get lower and upper bound of an expression (if any)
- void getBounds (CouNumber &lb, CouNumber &ub)

  Get value of lower and upper bound of an expression (if any)
- void generateCuts (expression *w, **OsiCuts** &cs, const CouenneCutGenerator *cg, t_chg_bounds *=NULL, int=-1, CouNumber=-COUENNE_INFINITY, CouNumber=COUENNE_INFINITY)

  *generate equality between *this and *w*
- virtual enum expr_type code ()

  code for comparisons
- bool impliedBound (int index, CouNumber *l, CouNumber *u, t_chg_bounds *chg, enum auxSign=expression↩::AUX_EQ)

  implied bound processing
- virtual CouNumber selectBranch (const CouenneObject *obj, const **OsiBranchingInformation** *info, expression *&var, double *&brpts, double *&brDist, int &way)

  Set up branching object by evaluating many branching points for each expression's arguments.
- virtual void closestFeasible (expression *varind, expression *vardep, CouNumber &left, CouNumber &right) const

  closest feasible points in function in both directions
- virtual bool isCuttable (CouenneProblem *problem, int index) const

  can this expression be further linearized or are we on its concave ("bad") side

Additional Inherited Members

6.96.1 Detailed Description

class for $\sin f(x)$

Definition at line 47 of file CouenneExprSin.hpp.

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

- CouenneExprSin.hpp

6.97 Couenne::exprStore Class Reference

storage class for previously evaluated expressions

`#include <CouenneExprStore.hpp>`

Inheritance diagram for Couenne::exprStore:

Collaboration diagram for Couenne::exprStore:

Public Member Functions

- exprStore (expression *copy)
  Constructor.
- exprStore (const exprStore &e, Domain *d=NULL)
  Store constructor – Must go.
- virtual ~exprStore ()
  Destructor.
- virtual void print (std::ostream &out=std::cout, bool descend=false) const
  Printing.
- virtual expression * clone (Domain *d=NULL) const
  Cloning method.
- virtual CouNumber operator() ()
  function for evaluating the expression – returns value of exprCopy pointed to, which returns a value stored from a previous evaluation

Protected Attributes

- CouNumber value_
  Value of the (previously evaluated) expression.

Additional Inherited Members

6.97.1 Detailed Description

storage class for previously evaluated expressions

Definition at line 23 of file CouenneExprStore.hpp.

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

- CouenneExprStore.hpp

6.98 Couenne::exprSub Class Reference

class for subtraction, $f(x) - g(x)$

`#include <CouenneExprSub.hpp>`

Inheritance diagram for Couenne::exprSub:

Collaboration diagram for Couenne::exprSub:

Public Member Functions

- exprSub (expression **al, int n=2)

  Constructor.
- exprSub (expression *arg0, expression *arg1)

  Constructor with two explicit elements.
- expression * clone (Domain *d=NULL) const

  Cloning method.
- std::string printOp () const

  print operator
- CouNumber operator() ()

  Function for the evaluation of the difference.
- expression * differentiate (int index)

  Differentiation.
- expression * simplify ()

  Simplification.
- virtual int Linearity ()

  Get a measure of "how linear" the expression is (see CouenneTypes.h)
- void getBounds (expression *&, expression *&)

  Get lower and upper bound of an expression (if any)
- void getBounds (CouNumber &lb, CouNumber &ub)

  Get value of lower and upper bound of an expression (if any)
- virtual exprAux * standardize (CouenneProblem *p, bool addAux=true)

  Reduce expression in standard form, creating additional aux variables (and constraints)
- virtual void generateCuts (expression *, **OsiCuts** &, const CouenneCutGenerator *, t_chg_bounds *=NULL, int=-1, CouNumber=-COUENNE_INFINITY, CouNumber=COUENNE_INFINITY)

  Special version for linear constraints.
- virtual enum expr_type code ()

  Code for comparisons.
- bool impliedBound (int, CouNumber *, CouNumber *, t_chg_bounds *, enum auxSign=expression::AUX_EQ)

  Implied bound processing.

Additional Inherited Members

6.98.1 Detailed Description

class for subtraction, $f(x) - g(x)$

Definition at line 22 of file CouenneExprSub.hpp.

6.98.2 Member Function Documentation

6.98.2.1 CouNumber Couenne::exprSub::operator() ( ) [inline], [virtual]

Function for the evaluation of the difference.

Compute difference.

Implements Couenne::expression.

Definition at line 88 of file CouenneExprSub.hpp.

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

- CouenneExprSub.hpp

6.99 Couenne::exprSum Class Reference

class sum, $\sum_{i=1}^{n} f_i(x)$

`#include <CouenneExprSum.hpp>`

Inheritance diagram for Couenne::exprSum:

Collaboration diagram for Couenne::exprSum:

Public Member Functions

- exprSum (expression **=NULL, int=0)
  Constructors, destructor.
- exprSum (expression *, expression *)
  Constructor with two elements.
- virtual ~exprSum ()
  Empty destructor.
- virtual expression * clone (Domain *d=NULL) const
  Cloning method.
- std::string printOp () const
  Print operator.
- virtual CouNumber operator() ()
  Function for the evaluation of the expression.
- virtual expression * differentiate (int index)
  Differentiation.
- virtual expression * simplify ()
  Simplification.

- virtual int Linearity ()

  Get a measure of "how linear" the expression is:

- virtual void getBounds (expression *&, expression *&)

  Get lower and upper bound of an expression (if any)

- virtual void getBounds (CouNumber &, CouNumber &)

  Get lower and upper bound of an expression (if any)

- virtual exprAux * standardize (CouenneProblem *p, bool addAux=true)

  Reduce expression in standard form, creating additional aux variables (and constraints)

- virtual void generateCuts (expression *, **OsiCuts** &, const CouenneCutGenerator *, t_chg_bounds *=NULL, int=-1, CouNumber=-COUENNE_INFINITY, CouNumber=COUENNE_INFINITY)

  Special version for linear constraints.

- virtual enum expr_type code ()

  Code for comparison.

- virtual bool impliedBound (int, CouNumber *, CouNumber *, t_chg_bounds *, enum auxSign=expression::AUX_EQ)

  Implied bound.

- exprAux * createQuadratic (CouenneProblem *)

  Checks for quadratic terms in the expression and returns an exprQuad if there are enough to create something that can be convexified.

Protected Member Functions

- int impliedBoundSum (CouNumber wl, CouNumber wu, std::vector< CouNumber > &xl, std::vector< CouNumber > &xu, std::vector< std::pair< int, CouNumber > > &nl, std::vector< std::pair< int, CouNumber > > &nu)

  inferring bounds on factors of a product

Additional Inherited Members

6.99.1 Detailed Description

class sum, $\sum_{i=1}^{n} f_i(x)$

Definition at line 22 of file CouenneExprSum.hpp.

6.99.2 Member Function Documentation

6.99.2.1 CouNumber Couenne::exprSum::operator() ( ) [inline], [virtual]

Function for the evaluation of the expression.

compute sum

Implements Couenne::expression.

Reimplemented in Couenne::exprQuad, and Couenne::exprGroup.

Definition at line 118 of file CouenneExprSum.hpp.

6.99.2.2 virtual bool Couenne::exprSum::impliedBound (int , CouNumber * , CouNumber * , t_chg_bounds * , enum *auxSign* = `expression::AUX_EQ`) [virtual]

Implied bound.

An expression

$w = a0 + \sum_{i \in I1} a_i x_i + \sum_{i \in I2} a_i x_i$

is given such that all a_i are positive for $i \in I1$ and negative for $i \in I2$. If the bounds on $w \in [l, u]$, implied bounds on all $x_i, i \in I1 \cup I2$ are as follows:

$\forall i \in I1\ x_i \geq (l - a0 - \sum_{j \in I1 | j \neq i} a_j u_j - \sum_{j \in I2} a_j l_j)/a_i\ x_i \leq (u - a0 - \sum_{j \in I1 | j \neq i} a_j l_j - \sum_{j \in I2} a_j u_j)/a_i$

$\forall i \in I2\ x_i \geq (u - a0 - \sum_{j \in I1} a_j u_j - \sum_{j \in I2 | j \neq i} a_j l_j)/a_i\ x_i \leq (l - a0 - \sum_{j \in I1} a_j l_j - \sum_{j \in I2 | j \neq i} a_j u_j)/a_i,$

where l_i and u_i are lower and upper bound, respectively, of x_i. We also have to check if some of these bounds are infinite.

Reimplemented from Couenne::expression.

Reimplemented in Couenne::exprQuad.

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

- CouenneExprSum.hpp

6.100 Couenne::exprTrilinear Class Reference

class for multiplications

`#include <CouenneExprTrilinear.hpp>`

Inheritance diagram for Couenne::exprTrilinear:

Collaboration diagram for Couenne::exprTrilinear:

Public Member Functions

- exprTrilinear (expression **, int)

  Constructor.
- exprTrilinear (expression *, expression *, expression *)

  Constructor with two arguments.
- expression * clone (Domain *d=NULL) const

  Cloning method.
- CouNumber gradientNorm (const double *x)

  return l-2 norm of gradient at given point
- virtual void getBounds (expression *&, expression *&)

  Get lower and upper bound of an expression (if any)
- virtual void getBounds (CouNumber &lb, CouNumber &ub)

  Get value of lower and upper bound of an expression (if any)
- void generateCuts (expression *w, **OsiCuts** &cs, const CouenneCutGenerator *cg, t_chg_bounds *=NULL, int=-1, CouNumber=-COUENNE_INFINITY, CouNumber=COUENNE_INFINITY)

  *generate equality between *this and *w*
- virtual enum expr_type code ()

code for comparison

- bool impliedBound (int, CouNumber ∗, CouNumber ∗, t_chg_bounds ∗, enum Couenne::expression::auxSign=Couenne::expression::AUX_EQ)

  implied bound processing

- virtual CouNumber selectBranch (const CouenneObject ∗obj, const **OsiBranchingInformation** ∗info, expression ∗&var, double ∗&brpts, double ∗&brDist, int &way)

  set up branching object by evaluating many branching points for each expression's arguments

- virtual void closestFeasible (expression ∗varind, expression ∗vardep, CouNumber &left, CouNumber &right) const

  compute y^{lv} and y^{uv} for Violation Transfer algorithm

Additional Inherited Members

6.100.1 Detailed Description

class for multiplications

Definition at line 21 of file CouenneExprTrilinear.hpp.

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

- CouenneExprTrilinear.hpp

6.101 Couenne::exprUBCos Class Reference

class to compute lower bound of a cosine based on the bounds of its arguments

`#include <CouenneExprBCos.hpp>`

Inheritance diagram for Couenne::exprUBCos:

Collaboration diagram for Couenne::exprUBCos:

Public Member Functions

- exprUBCos (expression ∗lb, expression ∗ub)

  Constructors, destructor.

- expression ∗ clone (Domain ∗d=NULL) const

  cloning method

- CouNumber operator() ()

  function for the evaluation of the expression

- std::string printOp () const

  print operator

- enum pos printPos () const

  print position (PRE, INSIDE, POST)

Additional Inherited Members

6.101.1 Detailed Description

class to compute lower bound of a cosine based on the bounds of its arguments

Definition at line 80 of file CouenneExprBCos.hpp.

6.101.2 Member Function Documentation

6.101.2.1 CouNumber Couenne::exprUBCos::operator() ( ) [inline], [virtual]

function for the evaluation of the expression

compute sum

Implements Couenne::expression.

Definition at line 111 of file CouenneExprBCos.hpp.

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

- CouenneExprBCos.hpp

6.102 Couenne::exprUBDiv Class Reference

class to compute upper bound of a fraction based on the bounds of both numerator and denominator

`#include <CouenneExprBDiv.hpp>`

Inheritance diagram for Couenne::exprUBDiv:

Collaboration diagram for Couenne::exprUBDiv:

Public Member Functions

- exprUBDiv (expression **al, int n)
  Constructors, destructor.
- expression * clone (Domain *d=NULL) const
  cloning method
- CouNumber operator() ()
  function for the evaluation of the expression
- enum pos printPos () const
  print position (PRE, INSIDE, POST)
- std::string printOp () const
  print operator

Additional Inherited Members

6.102.1 Detailed Description

class to compute upper bound of a fraction based on the bounds of both numerator and denominator

Definition at line 85 of file CouenneExprBDiv.hpp.

6.102.2 Member Function Documentation

6.102.2.1 CouNumber Couenne::exprUBDiv::operator() ( ) [inline],[virtual]

function for the evaluation of the expression

compute sum

Implements Couenne::expression.

Definition at line 112 of file CouenneExprBDiv.hpp.

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

- CouenneExprBDiv.hpp

6.103 Couenne::exprUBMul Class Reference

class to compute upper bound of a product based on the bounds of both factors

`#include <CouenneExprBMul.hpp>`

Inheritance diagram for Couenne::exprUBMul:

Collaboration diagram for Couenne::exprUBMul:

Public Member Functions

- exprUBMul (expression **al, int n)

  Constructors, destructor.
- expression * clone (Domain *d=NULL) const

  cloning method
- CouNumber operator() ()

  function for the evaluation of the expression
- enum pos printPos () const

  print position (PRE, INSIDE, POST)
- std::string printOp () const

  print operator

Additional Inherited Members

6.103.1 Detailed Description

class to compute upper bound of a product based on the bounds of both factors

Definition at line 93 of file CouenneExprBMul.hpp.

6.103.2 Member Function Documentation

6.103.2.1 CouNumber Couenne::exprUBMul::operator() ( ) [inline], [virtual]

function for the evaluation of the expression

compute sum

Implements Couenne::expression.

Definition at line 120 of file CouenneExprBMul.hpp.

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

- CouenneExprBMul.hpp

6.104 Couenne::exprUBQuad Class Reference

class to compute upper bound of a fraction based on the bounds of both numerator and denominator

`#include <CouenneExprBQuad.hpp>`

Inheritance diagram for Couenne::exprUBQuad:

Collaboration diagram for Couenne::exprUBQuad:

Public Member Functions

- exprUBQuad (exprQuad *ref)
  Constructor.
- exprUBQuad (const exprUBQuad &src, Domain *d=NULL)
  copy constructor
- ~exprUBQuad ()
  destructor
- expression * clone (Domain *d=NULL) const
  cloning method
- CouNumber operator() ()
  function for the evaluation of the expression
- virtual void print (std::ostream &s=std::cout, bool descend=false) const
  I/O.

Additional Inherited Members

6.104.1 Detailed Description

class to compute upper bound of a fraction based on the bounds of both numerator and denominator

Definition at line 60 of file CouenneExprBQuad.hpp.

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

- CouenneExprBQuad.hpp

6.105 Couenne::exprUBSin Class Reference

class to compute lower bound of a sine based on the bounds on its arguments

`#include <CouenneExprBSin.hpp>`

Inheritance diagram for Couenne::exprUBSin:

Collaboration diagram for Couenne::exprUBSin:

Public Member Functions

- exprUBSin (expression *lb, expression *ub)
  Constructors, destructor.
- expression * clone (Domain *d=NULL) const
  cloning method
- CouNumber operator() ()
  function for the evaluation of the expression
- std::string printOp () const
  print operator
- enum pos printPos () const
  print position (PRE, INSIDE, POST)

Additional Inherited Members

6.105.1 Detailed Description

class to compute lower bound of a sine based on the bounds on its arguments

Definition at line 80 of file CouenneExprBSin.hpp.

6.105.2 Member Function Documentation

6.105.2.1 CouNumber Couenne::exprUBSin::operator() ( ) [inline], [virtual]

function for the evaluation of the expression

compute sum

Implements Couenne::expression.

Definition at line 111 of file CouenneExprBSin.hpp.

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

- CouenneExprBSin.hpp

6.106 Couenne::exprUnary Class Reference

expression class for unary functions (sin, log, etc.)

`#include <CouenneExprUnary.hpp>`

Inheritance diagram for Couenne::exprUnary:

Collaboration diagram for Couenne::exprUnary:

Public Member Functions

- virtual enum nodeType Type () const
  node type
- exprUnary (expression *argument)
  Constructor.
- virtual unary_function F ()
  the operator itself (e.g. sin, log...)
- virtual ~exprUnary ()
  Destructor.
- int nArgs () const
  return number of arguments
- virtual expression * Argument () const
  return argument
- virtual expression ** ArgPtr ()
  return pointer to argument
- virtual void print (std::ostream &out=std::cout, bool=false) const
  print this expression to iostream
- virtual enum pos printPos () const
  print position (PRE, INSIDE, POST)
- virtual std::string printOp () const
  print operator
- virtual CouNumber operator() ()
  compute value of unary operator
- virtual int DepList (std::set< int > &deplist, enum dig_type type=ORIG_ONLY)
  fill in the set with all indices of variables appearing in the expression
- expression * simplify ()
  simplification
- virtual int Linearity ()
  get a measure of "how linear" the expression is (see CouenneTypes.h) for general univariate functions, return nonlinear.
- virtual exprAux * standardize (CouenneProblem *, bool addAux=true)
  reduce expression in standard form, creating additional aux variables (and constraints)
- virtual enum expr_type code ()
  type of operator
- virtual bool isInteger ()
  is this expression integer?
- virtual int compare (exprUnary &)
  compare two unary functions
- virtual int rank ()
  used in rank-based branching variable choice
- virtual void fillDepSet (std::set< DepNode *, compNode > *dep, DepGraph *g)
  fill in dependence structure
- virtual void replace (exprVar *, exprVar *)
  replace variable with other
- virtual void realign (const CouenneProblem *p)
  empty function to redirect variables to proper variable vector

Protected Attributes

- expression * argument_
  single argument taken by this expression

Additional Inherited Members

6.106.1 Detailed Description

expression class for unary functions (sin, log, etc.)

univariate operator-type expression: requires single argument. All unary functions are derived from this base class, which has a lot of common methods that need not be re-implemented by any univariate class.

Definition at line 33 of file CouenneExprUnary.hpp.

6.106.2 Member Function Documentation

6.106.2.1 virtual int Couenne::exprUnary::Linearity ( ) [inline], [virtual]

get a measure of "how linear" the expression is (see CouenneTypes.h) for general univariate functions, return nonlinear.

Reimplemented from Couenne::expression.

Reimplemented in Couenne::exprInv, and Couenne::exprOpp.

Definition at line 96 of file CouenneExprUnary.hpp.

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

- CouenneExprUnary.hpp

6.107 Couenne::exprUpperBound Class Reference

upper bound

`#include <CouenneExprBound.hpp>`

Inheritance diagram for Couenne::exprUpperBound:

Collaboration diagram for Couenne::exprUpperBound:

Public Member Functions

- enum nodeType Type () const
  Node type.
- exprUpperBound (int varIndex, Domain *d=NULL)
  Constructor.
- exprUpperBound (const exprUpperBound &src, Domain *d=NULL)
  Copy constructor.
- exprUpperBound * clone (Domain *d=NULL) const
  cloning method
- void print (std::ostream &out=std::cout, bool=false) const

Print to iostream.

- CouNumber operator() ()

  return the value of the variable

- expression * differentiate (int)

  differentiation

- int dependsOn (int *, int, enum dig_type type=STOP_AT_AUX)

  dependence on variable set

- virtual int Linearity ()

  get a measure of "how linear" the expression is:

- virtual enum expr_type code ()

  code for comparisons

Additional Inherited Members

6.107.1 Detailed Description

upper bound

Definition at line 87 of file CouenneExprBound.hpp.

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

- CouenneExprBound.hpp

6.108 Couenne::exprVar Class Reference

variable-type operator

`#include <CouenneExprVar.hpp>`

Inheritance diagram for Couenne::exprVar:

Collaboration diagram for Couenne::exprVar:

Public Member Functions

- virtual enum nodeType Type () const

  Node type.

- exprVar (int varIndex, Domain *d=NULL)

  Constructor.

- virtual ~exprVar ()

  destructor

- exprVar (const exprVar &e, Domain *d=NULL)

  Copy constructor.

- virtual exprVar * clone (Domain *d=NULL) const

  Cloning method.

- int Index () const

  Get variable index in problem.

- virtual expression * Lb ()

Get lower bound expression.

- virtual expression * Ub ()

  Get upper bound expression.

- virtual CouNumber & lb ()

  Get/set lower bound value.

- virtual CouNumber & ub ()

  Get/set upper bound value.

- virtual void print (std::ostream &out=std::cout, bool=false) const

  print

- virtual CouNumber operator() ()

  return the value of the variable

- virtual CouNumber gradientNorm (const double *x)

  return l-2 norm of gradient at given point

- virtual expression * differentiate (int index)

  differentiation

- virtual int DepList (std::set< int > &deplist, enum dig_type type=ORIG_ONLY)

  fill in the set with all indices of variables appearing in the expression

- virtual void crossBounds ()

  set bounds depending on both branching rules and propagated bounds.

- virtual expression * simplify ()

  simplify

- virtual int Linearity ()

  get a measure of "how linear" the expression is (see CouenneTypes.hpp)

- virtual bool isDefinedInteger ()

  is this expression defined as an integer?

- virtual bool isInteger ()

  is this variable integer?

- virtual void getBounds (expression *&, expression *&)

  Get expressions of lower and upper bound of an expression (if any)

- virtual void getBounds (CouNumber &lb, CouNumber &ub)

  Get value of lower and upper bound of an expression (if any)

- virtual void generateCuts (**OsiCuts** &, const CouenneCutGenerator *, t_chg_bounds *=NULL, int=-1, CouNumber=-COUENNE_INFINITY, CouNumber=COUENNE_INFINITY)

  Get values of lower and upper bound of an expression (if any)

- virtual void generateCuts (expression *w, **OsiCuts** &cs, const CouenneCutGenerator *cg, t_chg_bounds *=NULL, int=-1, CouNumber=-COUENNE_INFINITY, CouNumber=COUENNE_INFINITY)

  generate convexification cut for constraint w = this

- virtual enum expr_type code ()

  code for comparison

- virtual bool impliedBound (int, CouNumber *, CouNumber *, t_chg_bounds *, enum auxSign=expression::AUX_EQ)

  implied bound processing

- virtual int rank ()

  rank of an original variable is always one

- virtual void fillDepSet (std::set< DepNode *, compNode > *, DepGraph *)

  update dependence set with index of this variable

- virtual bool isFixed ()

is this variable fixed?

- virtual void linkDomain (Domain *d)

  link this variable to a domain

- virtual Domain * domain ()

  return pointer to variable domain

- virtual void zeroMult ()

  Disable variable (empty for compatibility with exprAux)

- virtual void setInteger (bool value)

  Set this variable as integer (empty for compatibility with exprAux)

- virtual enum convexity convexity () const

  either CONVEX, CONCAVE, AFFINE, or NONCONVEX

- virtual CouenneObject * properObject (CouenneCutGenerator *c, CouenneProblem *p, Bonmin::BabSetupBase *base, **JnlstPtr** jnlst_)

  return proper object to handle expression associated with this variable (NULL if this is not an auxiliary)

- virtual enum auxSign sign () const

  return its sign in the definition constraint

Protected Attributes

- int varIndex_

  The index of the variable.

- Domain * domain_

  Pointer to a descriptor of the current point/bounds.

Additional Inherited Members

6.108.1 Detailed Description

variable-type operator

All variables of the expression must be objects of this class or of the derived exprAux class

Definition at line 45 of file CouenneExprVar.hpp.

6.108.2 Member Function Documentation

6.108.2.1 virtual void Couenne::exprVar::crossBounds ( ) [inline], [virtual]

set bounds depending on both branching rules and propagated bounds.

To be used after standardization

Reimplemented in Couenne::exprAux.

Definition at line 118 of file CouenneExprVar.hpp.

6.108.2.2 virtual void Couenne::exprVar::generateCuts (OsiCuts & , const CouenneCutGenerator * , t_chg_bounds * = NULL, int = -1, CouNumber = -COUENNE_INFINITY, CouNumber = COUENNE_INFINITY) [inline],[virtual]

Get values of lower and upper bound of an expression (if any)

generate cuts for expression associated with this auxiliary

Reimplemented in Couenne::exprAux.

Definition at line 156 of file CouenneExprVar.hpp.

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

- CouenneExprVar.hpp

6.109 Couenne::funtriplet Class Reference

Inheritance diagram for Couenne::funtriplet:

Public Member Functions

- funtriplet ()
  Basic constructor.
- virtual ~funtriplet ()
  Destructor.

6.109.1 Detailed Description

Definition at line 22 of file CouenneFunTriplets.hpp.

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

- CouenneFunTriplets.hpp

6.110 Couenne::GlobalCutOff Class Reference

6.110.1 Detailed Description

Definition at line 19 of file CouenneGlobalCutOff.hpp.

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

- CouenneGlobalCutOff.hpp

6.111 Couenne::InitHeuristic Class Reference

A heuristic that stores the initial solution of the NLP.

`#include <BonInitHeuristic.hpp>`

Inheritance diagram for Couenne::InitHeuristic:

Collaboration diagram for Couenne::InitHeuristic:

Public Member Functions

- InitHeuristic (double objValue, const double ∗sol, CouenneProblem &cp)

  Constructor with model and **Ipopt** *problems.*
- InitHeuristic (const InitHeuristic &other)

  Copy constructor.
- virtual ∼InitHeuristic ()

  Destructor.
- virtual **CbcHeuristic** ∗ clone () const

  Clone.
- InitHeuristic & operator= (const InitHeuristic &rhs)

  Assignment operator.
- virtual int solution (double &objectiveValue, double ∗newSolution)

  Run heuristic, return 1 if a better solution than the one passed is found and 0 otherwise.

6.111.1 Detailed Description

A heuristic that stores the initial solution of the NLP.

This is computed before Cbc is started, and in this way we can tell Cbc about this.

Definition at line 24 of file BonInitHeuristic.hpp.

6.111.2 Constructor & Destructor Documentation

6.111.2.1 Couenne::InitHeuristic::InitHeuristic (double *objValue,* const double ∗ *sol,* CouenneProblem & *cp*)

Constructor with model and **Ipopt** problems.

6.111.2.2 Couenne::InitHeuristic::InitHeuristic (const InitHeuristic & *other*)

Copy constructor.

6.111.3 Member Function Documentation

6.111.3.1 virtual CbcHeuristic∗ Couenne::InitHeuristic::clone ( ) const [virtual]

Clone.

Implements **CbcHeuristic**.

6.111.3.2 virtual int Couenne::InitHeuristic::solution (double & *objectiveValue,* double * *newSolution*) [virtual]

Run heuristic, return 1 if a better solution than the one passed is found and 0 otherwise.

objectiveValue Best known solution in input and value of solution found in output newSolution Solution found by heuristic.

Implements **CbcHeuristic**.

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

- BonInitHeuristic.hpp

6.112 Couenne::kpowertriplet Class Reference

Inheritance diagram for Couenne::kpowertriplet:

Collaboration diagram for Couenne::kpowertriplet:

Public Member Functions

- kpowertriplet (CouNumber exponent, CouNumber k)
  Basic constructor.
- virtual ~kpowertriplet ()
  Destructor.

6.112.1 Detailed Description

Definition at line 103 of file CouenneFunTriplets.hpp.

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

- CouenneFunTriplets.hpp

6.113 less_than_str Struct Reference

6.113.1 Detailed Description

Definition at line 128 of file CouenneProblem.hpp.

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

- CouenneProblem.hpp

6.114 Couenne::LinMap Class Reference

Public Member Functions

- std::map< int, CouNumber > & Map ()
  public access

- void insert (int index, CouNumber coe)

  insert a pair <int,CouNumber> into a map for linear terms

6.114.1 Detailed Description

Definition at line 48 of file CouenneLQelems.hpp.

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

- CouenneLQelems.hpp

6.115 Couenne::MultiProdRel Class Reference

Identifies 5-ples of variables of the form.

`#include <CouenneCrossConv.hpp>`

Inheritance diagram for Couenne::MultiProdRel:

Collaboration diagram for Couenne::MultiProdRel:

6.115.1 Detailed Description

Identifies 5-ples of variables of the form.

x_k := x_i x_j x_l := x_i x_p

x_q := x_k x_p OR x_q := x_k / x_j x_r := x_k x_j x_r := x_l / x_p

and generates, ONLY ONCE, a cut

x_q = x_r (in both cases).

Definition at line 82 of file CouenneCrossConv.hpp.

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

- CouenneCrossConv.hpp

6.116 myclass Struct Reference

6.116.1 Detailed Description

Definition at line 122 of file CouenneProblem.hpp.

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

- CouenneProblem.hpp

6.117 myclass0 Struct Reference

6.117.1 Detailed Description

Definition at line 76 of file CouenneProblem.hpp.

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

- CouenneProblem.hpp

6.118 Nauty Class Reference

Public Member Functions

- std::vector< std::vector< int > > * getOrbits () const

  Returns the orbits in a "convenient" form.
- void setWriteAutoms (const std::string &afilename)

  Methods to classify orbits.

6.118.1 Detailed Description

Definition at line 23 of file Nauty.h.

6.118.2 Member Function Documentation

6.118.2.1 void Nauty::setWriteAutoms (const std::string & *afilename*)

Methods to classify orbits.

Not horribly efficient, but gets the job done

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

- Nauty.h

6.119 Couenne::CouenneInfo::NlpSolution Class Reference

Class for storing an Nlp Solution.

`#include <BonCouenneInfo.hpp>`

Inheritance diagram for Couenne::CouenneInfo::NlpSolution:

Collaboration diagram for Couenne::CouenneInfo::NlpSolution:

Accessor methods

- const double * **solution** () const
- double **objVal** () const
- int **nVars** () const

6.119.1 Detailed Description

Class for storing an Nlp Solution.

Definition at line 26 of file BonCouenneInfo.hpp.

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

- BonCouenneInfo.hpp

6.120 Couenne::NlpSolveHeuristic Class Reference

Inheritance diagram for Couenne::NlpSolveHeuristic:

Collaboration diagram for Couenne::NlpSolveHeuristic:

Public Member Functions

- NlpSolveHeuristic ()
  Default constructor.
- NlpSolveHeuristic (**CbcModel** &mip, Bonmin::OsiTMINLPInterface &nlp, bool cloneNlp=false, CouenneProblem *couenne=NULL)
  Constructor with model and **Ipopt** *problems.*
- NlpSolveHeuristic (const NlpSolveHeuristic &other)
  Copy constructor.
- virtual ~NlpSolveHeuristic ()
  Destructor.
- virtual **CbcHeuristic** * clone () const
  Clone.
- NlpSolveHeuristic & operator= (const NlpSolveHeuristic &rhs)
  Assignment operator.
- void setNlp (Bonmin::OsiTMINLPInterface &nlp, bool cloneNlp=true)
  Set the nlp solver.
- void setCouenneProblem (CouenneProblem *)
  set the couenne problem to use.
- virtual void resetModel (**CbcModel** *model)
  Does nothing.
- virtual int solution (double &objectiveValue, double *newSolution)
  Run heuristic, return 1 if a better solution than the one passed is found and 0 otherwise.
- void setMaxNlpInf (double value)
  set maxNlpInf.
- void setNumberSolvePerLevel (int value)
  set number of nlp's solved for each given level of the tree

Static Public Member Functions

- static void registerOptions (**Ipopt::SmartPtr**< Bonmin::RegisteredOptions >)
  initialize options

6.120.1 Detailed Description

Definition at line 28 of file BonNlpHeuristic.hpp.

6.120.2 Constructor & Destructor Documentation

6.120.2.1 Couenne::NlpSolveHeuristic::NlpSolveHeuristic ( )

Default constructor.

6.120.2.2 Couenne::NlpSolveHeuristic::NlpSolveHeuristic (CbcModel & *mip,* Bonmin::OsiTMINLPInterface & *nlp,* bool *cloneNlp* = false, CouenneProblem * *couenne* = NULL)

Constructor with model and **Ipopt** problems.

6.120.2.3 Couenne::NlpSolveHeuristic::NlpSolveHeuristic (const NlpSolveHeuristic & *other*)

Copy constructor.

6.120.3 Member Function Documentation

6.120.3.1 virtual CbcHeuristic* Couenne::NlpSolveHeuristic::clone ( ) const [virtual]

Clone.

Implements **CbcHeuristic**.

6.120.3.2 void Couenne::NlpSolveHeuristic::setNlp (Bonmin::OsiTMINLPInterface & *nlp,* bool *cloneNlp* = true)

Set the nlp solver.

6.120.3.3 void Couenne::NlpSolveHeuristic::setCouenneProblem (CouenneProblem *)

set the couenne problem to use.

6.120.3.4 virtual void Couenne::NlpSolveHeuristic::resetModel (CbcModel * *model*) [inline], [virtual]

Does nothing.

Implements **CbcHeuristic**.

Definition at line 53 of file BonNlpHeuristic.hpp.

6.120.3.5 virtual int Couenne::NlpSolveHeuristic::solution (double & *objectiveValue,* double * *newSolution*) [virtual]

Run heuristic, return 1 if a better solution than the one passed is found and 0 otherwise.

objectiveValue Best known solution in input and value of solution found in output newSolution Solution found by heuristic.

Implements **CbcHeuristic**.

6.120.3.6 void Couenne::NlpSolveHeuristic::setMaxNlpInf (double *value*) [inline]

set maxNlpInf.

Definition at line 61 of file BonNlpHeuristic.hpp.

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

- BonNlpHeuristic.hpp

6.121 Node Class Reference

6.121.1 Detailed Description

Definition at line 53 of file CouenneProblem.hpp.

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

- CouenneProblem.hpp

6.122 Couenne::powertriplet Class Reference

Inheritance diagram for Couenne::powertriplet:

Collaboration diagram for Couenne::powertriplet:

Public Member Functions

- powertriplet (CouNumber exponent, bool signpower=false)
  Basic constructor.
- virtual ~powertriplet ()
  Destructor.

6.122.1 Detailed Description

Definition at line 72 of file CouenneFunTriplets.hpp.

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

- CouenneFunTriplets.hpp

6.123 Couenne::PowRel Class Reference

Identifies 5-tuple of the form.

`#include <CouenneCrossConv.hpp>`

Inheritance diagram for Couenne::PowRel:

Collaboration diagram for Couenne::PowRel:

6.123.1 Detailed Description

Identifies 5-tuple of the form.

x_j := x_i ^ alpha x_p := x_i ^ beta

and generates cuts based on the relation

x_p = x_j ^ {beta/alpha}

Definition at line 125 of file CouenneCrossConv.hpp.

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

- CouenneCrossConv.hpp

6.124 Couenne::Qroot Class Reference

class that stores result of previous calls to rootQ into a map for faster access

`#include <CouenneRootQ.hpp>`

Collaboration diagram for Couenne::Qroot:

Public Member Functions

- Qroot ()

  Empty constructor – we only need the method to work on the static structure.
- ∼Qroot ()

  Empty destructor.
- CouNumber operator() (int k)

  Retrieve root of Q with order = k.

Static Protected Attributes

- static std::map< int, CouNumber > Qmap

  Maps an integer k with the root of $Q^k(x)$.

6.124.1 Detailed Description

class that stores result of previous calls to rootQ into a map for faster access

Definition at line 29 of file CouenneRootQ.hpp.

6.124.2 Constructor & Destructor Documentation

6.124.2.1 Couenne::Qroot::Qroot ( ) [inline]

Empty constructor – we only need the method to work on the static structure.

Definition at line 41 of file CouenneRootQ.hpp.

6.124.3 Member Function Documentation

6.124.3.1 CouNumber Couenne::Qroot::operator() (int *k*) [inline]

Retrieve root of Q with order = k.

If no such computation has been performed yet, do it here

Definition at line 49 of file CouenneRootQ.hpp.

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

- CouenneRootQ.hpp

6.125 Couenne::quadElem Class Reference

6.125.1 Detailed Description

Definition at line 20 of file CouenneLQelems.hpp.

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

- CouenneLQelems.hpp

6.126 Couenne::QuadMap Class Reference

Public Member Functions

- std::map< std::pair< int, int >, CouNumber > & Map ()

  public access
- void insert (int indI, int indJ, CouNumber coe)

  insert a pair <<int,int>,CouNumber> into a map for quadratic terms

6.126.1 Detailed Description

Definition at line 75 of file CouenneLQelems.hpp.

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

- CouenneLQelems.hpp

6.127 Couenne::simpletriplet Class Reference

Inheritance diagram for Couenne::simpletriplet:

Collaboration diagram for Couenne::simpletriplet:

Public Member Functions

- simpletriplet (unary_function f=NULL, unary_function fp=NULL, unary_function fpp=NULL, unary_function fp↩ l=NULL)

  Basic constructor.
- virtual ∼simpletriplet ()

  Destructor.

6.127.1 Detailed Description

Definition at line 40 of file CouenneFunTriplets.hpp.

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

- CouenneFunTriplets.hpp

6.128 Couenne::SmartAsl Class Reference

Inheritance diagram for Couenne::SmartAsl:

Collaboration diagram for Couenne::SmartAsl:

6.128.1 Detailed Description

Definition at line 33 of file BonCouenneSetup.hpp.

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

- BonCouenneSetup.hpp

6.129 Couenne::SumLogAuxRel Class Reference

Identifies 5-ples of variables of the form.

`#include <CouenneCrossConv.hpp>`

Inheritance diagram for Couenne::SumLogAuxRel:

Collaboration diagram for Couenne::SumLogAuxRel:

6.129.1 Detailed Description

Identifies 5-ples of variables of the form.

x_3 := log x_1 x_4 := log x_2 x_5 := x_1 x_2 in [l,u]

and generates a cut

x_3 + x_4 in [max {0, log l}, max {0, log u}].

This has to be repeatedly generated, even when l=u (l and/or u could change in other nodes).

Definition at line 58 of file CouenneCrossConv.hpp.

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

- CouenneCrossConv.hpp

6.130 Couenne::t_chg_bounds Class Reference

status of lower/upper bound of a variable, to be checked/modified in bound tightening

`#include <CouenneTypes.hpp>`

6.130.1 Detailed Description

status of lower/upper bound of a variable, to be checked/modified in bound tightening

Definition at line 66 of file CouenneTypes.hpp.

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

- CouenneTypes.hpp

Chapter 7

File Documentation

7.1 cons_rowcuts.h File Reference

constraint handler for rowcuts constraints enables separation of convexification cuts during SCIP solution procedure

```
#include "CouenneCutGenerator.hpp"
#include "OsiSolverInterface.hpp"
```

Include dependency graph for cons_rowcuts.h:

Index